bpm'online

Bpm'online
Development Guide

Simplify the future

Table of Contents

Getting started with the bpm’online platform 11
Architecture 11
Application infrastructure 11-15
Components 15-18
Packages, schemas, modules 18-22
Application interface and structure 22
Main menu 22-23
Sections 23-25
Section lists 25-28
Section analytics 28-29
Section actions 29-30
Filters 30-32
Tags 32-33
Record edit page 33-34
Details 34-36
Mini-page 36-37
Modal windows 37-38
Communication panel 38-39
Command line 39-40
Action dashboard 40-41
How to start the development 42-43
Development process organization 43-45
Organizing a development environment 45-46
Recommended development sequence 46-49
Development rules 49-50
How to deploy bpm'online on-site 50-51
Deploying the bpm'online cloud application 51-52
Create repository in SVN server 52-55
Working with packages 55-56
Package structure and contents 56-58

Package dependencies. Basic application packages 58-63
Package [Custom] 63-65
Creating and installing a package for development 65-69
Committing a package to repository 69-71
Installing packages from repository 71-75
Updating package from repository 75-76
Exporting packages from the application interface 76-79
Creating a package in the file system development mode 79-86
Binding data to packages 86-92
Transferring changes between the working environments 92
Exporting packages from the application interface 92-95
Installing marketplace applications from a zip archive 95-100
Transferring changes using schema export and import 100-101
Transferring changes using SVN 101-103
Transferring changes using WorkspaceConsole 103-106
Creating a custom client module schema 106-111
Creating the entity schema 111-120
Creating the [Source code] schema 120-121
Development resources 122
Built-in development tools 122
The [Configuration] section 122-129
The [Configuration] section. The [Data] tab 129-132
Source code and metadata viewport 132-134
Designers of configuration items 134
Workspace of the Object Designer 134-136
Module designer 136-138
Source code designer 138-141
Process designer workspace 141-142
User task designer workspace 142-143
Workspace of image list designer 143-145
Report designer 145
Setting up the report designer connection with server 145-147

Report designer workspace 147-151
Report designer features 151-155
Development in the file system 155-158
Visual Studio settings for development in the file system 158-163
Working with the server side source code in Visual Studio 163-168
Working with the client code in the file system 168-172
Working with SVN in the file system 172-175
Creating a package in the file system development mode 175-182
How to install an SVN package in the file system development mode 182-188
How to bind existing package to SVN 188-199
Updating and committing changes to the SVN from the file system 199-202
Creation of the package and switching to the file system development mode 202-210
Developing the configuration server code in the user project 210-218
Automatic displaying of changes in the development of the custom logic 218-222
Packages file content 222-225
Localization of the file content 225-228
How to create Unit-tests via NUnit and Visual Studio 228-234
How to use TypeScript when developing custom functions 234-239
Working with WorkspaceConsole 239-240
WorkspaceConsole settings 240-241
WorkspaceConsole parameters 241-246
Exporting packages from database 246-248
Saving packages to the database 248-251
Saving SVN packages 251-254
Client code debugging 254-258
Server code debugging 258-267
Bpm'online development cases 268
Section business logic 268
Creating a new section 268-271
Adding an action to the list 271-273
How to add a section action: handling the selection of a single record 273-276

How to add a section action: handling the selection of several records 276-279
Handling the selection of several records. Examples 280-285
How to add a button to a section 285-289
How to highlight a record in the list in color 289-291
Adding quick filter block to a section 291-294
Page configuration 294-295
Setting the edit page fields using business rules 295-299
The FILTRATION rule use case 299-304
The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition 304-308
The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition 308-310
The BINDPARAMETER rule. How to make a field required based on a specific condition 310-314
Business rules created via wizards 314-317
Adding an action to the edit page 317-322
Control elements 322
Adding a new field to the edit page 323-328
Adding a button to the edit page 328-330
How to add a button to an edit page in the new record add mode 330-333
How to add the button on the edit page in the combined mode 333-339
How to add a field with an image to the edit page 339-347
How to add the color select button to the edit page 347-351
How to add multi-currency field 351-357
How to add custom logic to the existing controls 357-361
Adding calculated fields 361-365
How to set a default value for a field 365-370
How to add the field validation 370-376
Using filtration for lookup fields. Examples 377-380
Adding an action panel 380-384
Adding a new channel to the action panel 384-390
Displaying contact's time zone 390-394
How to display the difference between dates on edit page fields 394-395

How to block fields of the edit page 395-397
Adding details 397-398
Adding an edit page detail 398-407
Adding a detail with an editable list 407-415
Creating a detail with selection from lookup 415-424
Adding multiple records to a detail 424-427
Creating a custom detail with fields 427-432
Advanced settings of a custom detail with fields 432-437
Creating a detail in wizards 437-441
Adding the [Attachments] detail 441-450
Displaying additional columns on the [Attachments] tab 450-452
How to hide menu commands of the detail with list 452-454
Business processes 454
How to add auto-numbering to the edit page field 454-461
Process launch from a client module 461-467
Creating custom [User task] process element 467-476
How to customize notifications for the [User task] process element 476-481
How to run bpm'online processes via web service 481-490
How to save the record without closing the edit page which is opened by the business process 490-496
Typical customizations 496
Creating pop-up summaries (mini pages) 496-504
Adding pop-up summaries (mini pages) to a module 504-508
Creating a pop-up summary (mini page) for adding records 508-514
Adding pop-up hints 514-521
How to modify sales pipeline calculations 521-526
How to enable additional filtering in a sales pipeline 526-528
Adding a custom dashboard widget 528-534
The Terrasoft.AlignableContainer custom element 534-543
Adding a duplicate search rule 543-546
Junk case custom filtering 546-547
How to display custom implementation of approving in the section wizard 547-548

How to create custom reminders and notifications 548-556
How to create the [Timeline] tab tiles bound to custom section 556-566
Adding multi-language email templates to a custom section 566-572
Analytics 572
How to create macros for a custom report in Word 572-582
Working with data 582-583
CRUD-operations in configuration 583
The use of EntitySchemaQuery implementation on client 583
Building of paths to columns relative to root schema 583-584
Adding columns to a query 585-588
Getting query result 588-590
EntitySchemaQuery filters handling 590-593
CRUD-operations on server side 593-594
Composing add data queries 594-595
The use of EntitySchemaQuery for creation of queries in database 595-606
Composing modify data queries 606-607
Composing delete data queries 607
Web-services in configuration 608
How to create custom configuration service 608-612
How to call configuration services with ServiceHelper 612-616
Creating anonymous web service 616-618
How to call configuration services using Postman 618-623
Reading multilingual data with EntitySchemaQuery 623-626
Views localization 626-631
Working with the localized data via Entity 631-635
Adding a multilingual terminator to an object schema 635-637
Using the DBExecutor for working with the database 637-638
Sales products customization 638-639
How to change the calculation for the "Closed" column in the [Forecasts] section. 639-644
Configuration of the editable columns on the product selection page 644-647
Service products customization 647

Adding a new rule for calculating case deadline 647-652
Adding a macro handler in email templates 652-655
Creating Web-to-Case landing pages 655-660
How to hide feed area in the agent desktop 660-662
Adding floating icons for internal case feed posts 662-665
Lending product customization 665
How to create custom verification action page 665-669
Using the EntityMapper schema 669-675
Marketing product customization 675
Adding a custom campaign element 675-686
Prediction 686
How to implement custom prediction model 686-690
Integration with bpm'online and public API 691
Choosing the method of integration with bpm'online 691-698
Authenticating external requests to bpm'online services 698-701
The AuthService.svc authentication service 701-704
Protection from CSRF attacks during integration with bpm'online 704-705
DataService web service 705-706
DataService. Adding records 706-713
DataService. Reading records 713-722
DataService. Data filtering 722-729
DataService. Using macros 729-733
DataService. Updating records 733-738
DataService. Deleting records 738-742
DataService. Batch queries 742-746
OData 746-747
Possibilities for the bpm'online integration over the OData protocol 747-749
Working with bpm'online objects over the OData protocol using Http request 749-757
Working with bpm'online objects over the OData protocol WCF-client 757-762
Examples of requests for filter selection 762-767
Executing OData queries using Fiddler 767-779

Integration of third-party sites via iframe 779-783
Web-To-Object. Using landings and web-forms 783-785
The ProcessEngineService.svc web service 785-786
Platform description 787
System Settings 787
Setting user session timeout 787-790
Working with data structure 790
Configuration localizable resources 790-793
Localizable resource structure and use 793-795
Localization tables 796-797
Bound data structure 797-798
User interface 798-799
AMD concept Modules 799-801
Modular development principles in bpm'online 801-807
Client Modules 807-812
Client view model schemas 812-813
Mixins. The "mixins" property 813-814
Attributes. The "attributes" property 814-816
Messages. The "messages" property 816-818
Methods. The "methods" property 818
Rules. The "rules" property 818-819
Business rules. The businessRules property 819-820
Modules. The "modules" property 820-821
The "diff" array 821-823
Alias mechanism 823-826
Schema formatting requirements for compatibility with wizards 826-831
Handling a data context loss 831-835
Properties. The "properties" property 835-836
Automatically generated view model properties 836-837
Sandbox. Module message exchange 837-843
Sandbox. Bidirectional messages 843-847
Sandbox. Loading and unloading modules 847-851

New bindTo format at setting connection between view and viewModel 851-852
Controls 852
Controls. Introduction 852-853
Details 853-862
The [Connected entity profile] control 862-871
SourceCodeEditMixin class description and work examples. 871-875
Blocking edit page fields 875-876
Dashboard widgets 876-879
Charts 879-881
Metrics 881-882
Gauge 882-884
Lists 884-885
Web-page 885
Sales pipeline 885-886
Scheduler setup 886
Recommendations on scheduler setup 886-889
Quartz policies for the processing of overdue tasks 889-891
Integration 891
Phone integration 891-894
Oktell 894-897
Webitel 897-898
Asterisk 898-900
Email integration 900-901
Working with email threads 901-903
Self-service Portal 903-904
ClientMessageBridge 904
ClientMessageBridge. Message history save mechanism 904-906
ClientMessageBridge. API description 906-908
ClientMessageBridge. The client-side WebSocket message handler 908-913
Sync Engine synchronization mechanism 913
Bpm'online synchronization with external storages 913-921
Synchronizing metadata in bpm'online 921-924

Synchronizing tasks with MS Exchange 924-926
Synchronizing email with MS Exchange 926-928
Synchronizing contacts with MS Exchange 928-931
Synchronizing appointments with MS Exchange 931-934
Data Enrichment and Prediction 934
Contact data enrichment from emails 934-937
Machine learning service 937-943
Creating data queries for the machine learning model 943-946
Bpm'online lending 946
Terrasoft.Configuration.EntityMapper class 946-949
Bpm'online marketing 949
Campaign elements 949-951
Bpm'online service 951
PortalMessagePublisherExtensions mixin. Portal messages in SectionActionDashboard 951-953
DataManager class description and use cases 953-957
Feature Toggle. Mechanism of enabling and disabling functions 957-960
The MoneyUtilsMixin mixin 960-964
The DecimalUtils module 964-968
Basic macros in the MS Word printables 968-971
Web-to-Case 971-972
Separate query pool 972-975
Development recommendations for Right-To-Left mode 975-976
Client static content in the file system 976-979
Record deactivation 979-980
Monitoring of private properties overriding. The Terrasoft.PrivateMemberWatcher class 980-982
The [Timeline] tab 982-986
Server content in the file system 986-987
Logging in bpm’online. Log4net 988-989

Getting started with the bpm’online platform

Contents

- **Architecture**
- **Application interface and structure**

Architecture

Contents

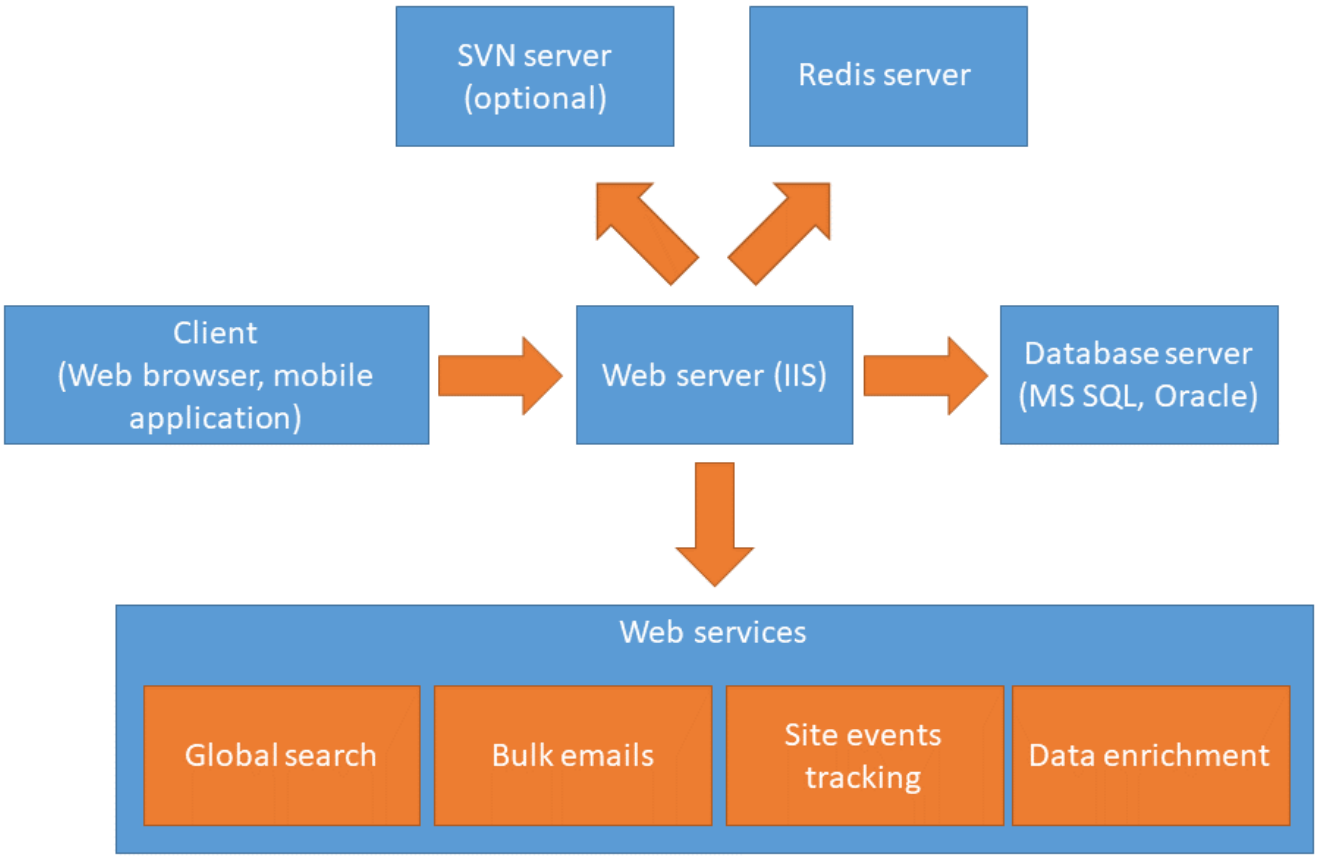
- **Application infrastructure**
- **Components**
- **Packages, schemas, modules**

Application infrastructure

Overview

From the server infrastructure view, the bpm’online is a three-tier architectural system with modifications (Fig. 1).

Fig. 1. Application infrastructure



The center of the infrastructure is the application server that runs under the Internet Information Services (IIS) version 7.0 or higher.

The next component of the architecture is the classic database server.

Client workplaces can be located on any available device (PC, laptop, mobile device). All requests to the server are performed via the web browser or mobile application.

In addition to the three main components, there are:

- database session server (Redis)
- version control server (optional)
- cloud web services.

Application server

Application server consists of two parts: WebAppLoader and WebApp.

WebAppLoader

The main purpose of WebAppLoader is authorization, authentication and redirect of users to the main application. The main functions of the WebAppLoader:

- user authorization
- user licensing and authentication
- starting the scheduler.

WebApp

After incoming requests were processed in the loader they are redirected to the WebApp. This part is responsible for the business logic of the system. This is an application that implements specific configuration and workplace in the system.

The database server

Database stores a data necessary for user or for the operation of the system itself. All configuration settings that define functionality of the products are also stored in the database.

Systems that can be used as database server:

- MS SQL Server 2012 SP3 or higher
- Oracle DBMS 11 g Release 2 and up (when deploying on-site)

NOTE

The MS SQL Server 2016 is used in the bpm’online cloud infrastructure.

Client

All requests to the server are performed via the web browser. Following web browsers are supported:

- Internet Explorer 11.0+,
- Firefox, the last official version on the bpm’online release date
- Chrome, the last official version on the bpm’online release date
- Safari, the last official version on the bpm’online release date.

The bpm’online mobile application is used to access the server with mobile devices.

The session storage server (Redis)

Redis main functions:

- storing the data of user sessions
- storing cached data
- Data exchange between web farm nodes.

Redis advantages:

- data is stored in RAM, it provides high performance of the system
- server can work under Unix OS

Version control system server (SVN).

This is an optional component that is enabled only when you need to start the development of custom configuration on the platform in parallel with system operation. Server functions:

- Transition of modifications between applications. Modifications are transferred with the packages.
- Storing the status of the configurations as a packages of specific version. Everything that is developed in the packages is stored in the version control system.

More information about the version control system can b e found in the “**Create repository in SVN server**” article.

Web services

This is an additional cloud services that can be accessed from the several bpm’online applications.

Global Search Service

The Global Search Service is created for integration of the ElasticSearch with the bpm'online and performs following functions:

1. Registration:

- Connects the client by creating an index in ElasticSearch, and stores the index application connection.

- Disconnects the client, upon request by deleting the index in the ElasticSearch.

2. Transition:

Participates in the indexing process. Takes the data from the application database and sends it to the ElasticSearch.

GSS consists of three components:

- *Service* – API for registration and access management to the global search
- *Worker* – exports data from the database and imports to the ElasticSearch index
- *Scheduler* – the scheduler of the Worker.

Bulk email service

Used for integration of bpm’online and bulk email services (ElasticEmail, UniOne, etc.). Enables to work with following bulk email types:

- Bulk emails.
- Transactional emails (immediate delivery to one recipient).

Access to the service is performed via public Web API.

Website event tracking service

Enables to track events from on the client site and pass them to the bpm’online. Application identification is preformed via unique API key by which the temporary event storage and the bpm’online instance for synchronization are defined. The JavaScript code embedded in the site pages used to perform tracking. The code sends events to the service.

The database enrichment service

Account data enrichment service uses different search technologies to find information about accounts and their communications from the open Internet sources.

The service for enriching contacts from email uses different search technologies to find information about contacts and their communications from the emails.

Deployment options

There are two deployment options:

1. On-site

2. Cloud.

On-site deployment

In On-site deployment, all costs associated with the organization of the server part (installation, configuration, maintenance, administration) are assigned to the customer.

One of the advantages of this deployment option is the simpler integration with the Active directory, since the domain controller is usually located in the enterprise LAN. Also, the on-site deployed application is better for development.

Disadvantages of this deployment options are that the customer bears constant costs to support this infrastructure (update, administration, maintenance costs).

How to deploy bpm'online on-site is described in the "Deploying bpm’online application on-site" article in the User Guide.

Cloud deployment

In the cloud deployment option, the application is deployed on the cloud servers (Amazon, Azure etc). All application server part is stored in the data centers and administrated by bpm’online employees. All issues related to administration, speed, scalability are solved by the bpm’online employees and client uses only the client part of the application.

Advantages of the Cloud deployment:

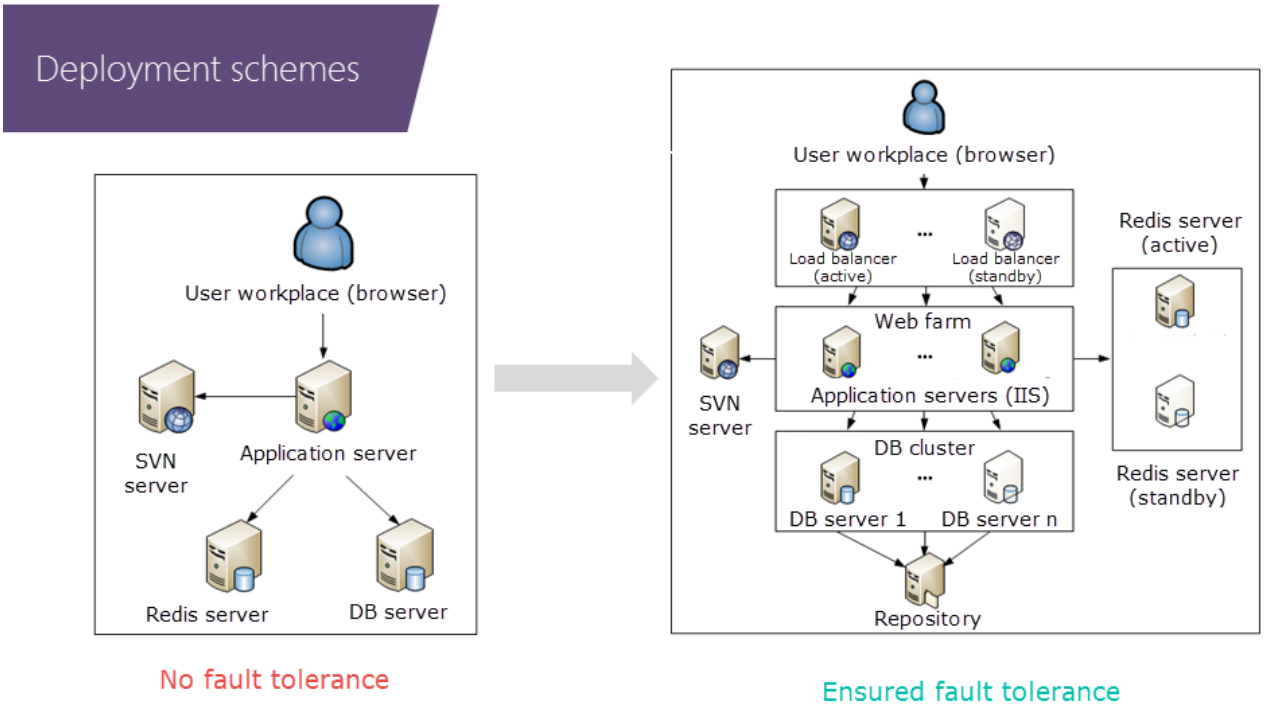
- timely updates
- maximal performance
- compliance with industry standards on data availability and security.

See "**Deploying the bpm'online cloud application**" article for more on restrictions related to bpm'online cloud.

On-site system installation schemas

Figure 2 contains two possible installation schemas of the bpm’online application.

Fig. 2. Installation schemas



Non fault tolerance system installation requires one application server, one Redis server, one database server and using SVN server for development.

Fault tolerance system requires:

- several workload balancers
- several web farm nodes
- several nodes of database cluster
- several nodes of the Redis cluster

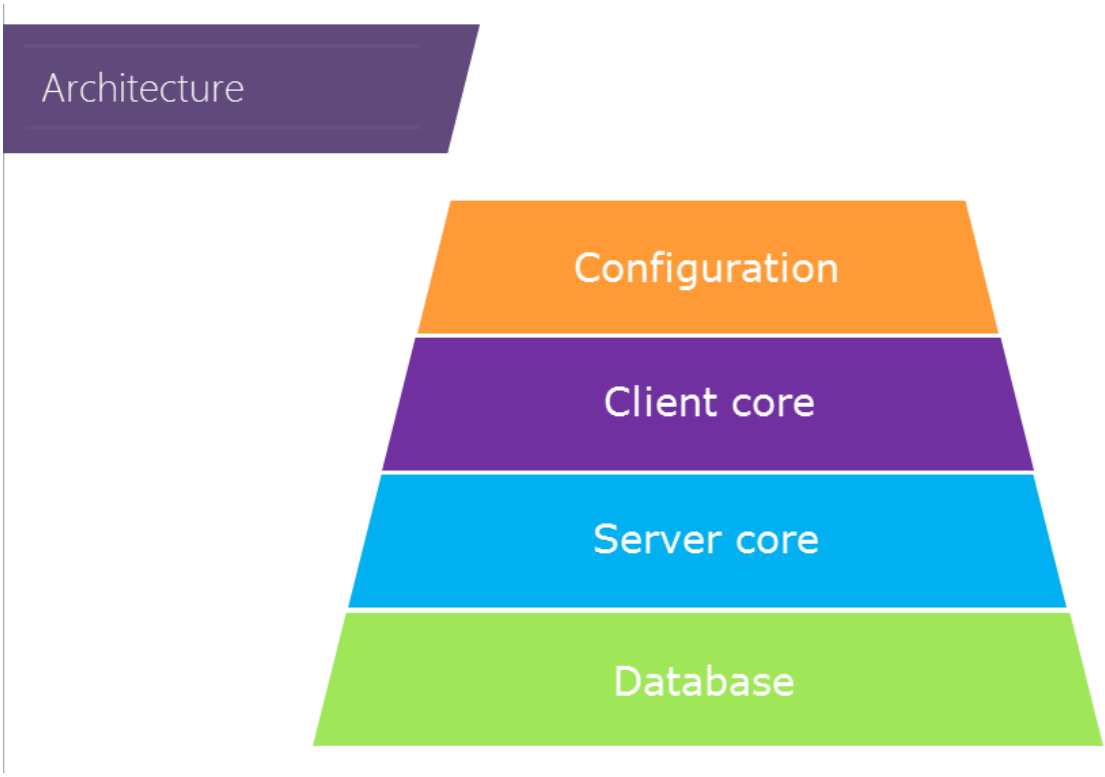
SVN server can be used for development, but it is not recommended on such complex fault tolerance systems.

Components

Overview

The bpm'online architecture is comprised of the following components (Fig. 1):

Fig. 1. Components



1. Database

Database stores user data, application settings and access rights settings at the physical data storage level.

Database primary functions:

- data storage
- data management
- configuration settings storage.

Database objects:

- tables
- views
- stored procedures
- indexes
- triggers in tables.

There is usually no need to work directly with the database objects during bpm'online development process. The system has tools that enable working with data directly from the UI.

Custom business logic can be implemented at the database level, with the help of views and stored procedures.

It may be faster and more rational to implement certain tasks at the database level. An example of such a task is the **setup of custom duplicate search rules**.

2. Server core

Server libraries are written in C# with the use of **.NET Framework classes ('.NET class libraries of platform core' in the on-line documentation)**.

The server core is a modifiable system component. Developers can create instances of server classes and use server libraries. Changes to these classes and libraries are restricted.

Server core primary components:

- ORM data model and its methods. It is recommended to use the object model for accessing data, although

direct database access is also implemented in the server core components.
- Packages and replacement mechanism.
- Server control element libraries. These elements include pages created using ASP.NET technology, for example, [Configuration] section pages.
- System web services.
- Functionality of designers and system sections.
- Libraries for integration with external services.
- Business process engine (ProcessEngine). This system component can execute algorithms that are set up as process diagrams.

3. Client core

The primary task of this level is to ensure the functioning of client modules. The **client core classes ('JavaScript API for platform core' in the on-line documentation)** are written in JavaScript with the use of various frameworks. They implement the UI and other business tasks on the browser side.

Client core primary components:

- Client framework external libraries. For example, the RequireJS library implements the mechanism for asynchronous loading of the client modules; the ExtJs framework implements the UI.
- **Sandbox** is a special client core component that ensures interaction between various client modules through message exchange.
- **Client modules** are JavaScript files that implement the functionality of primary system objects.

4. Configuration

A configuration is a set of functionalities available to users of a certain workspace. This includes:

- Server logic.
- Auto-generated classes, which are a product of system settings.
- Client logic, which includes pages, buttons, actions, reports, business processes and other customizable configuration elements.

Configurations are easily modifiable system components. Configurations consist of the following elements:

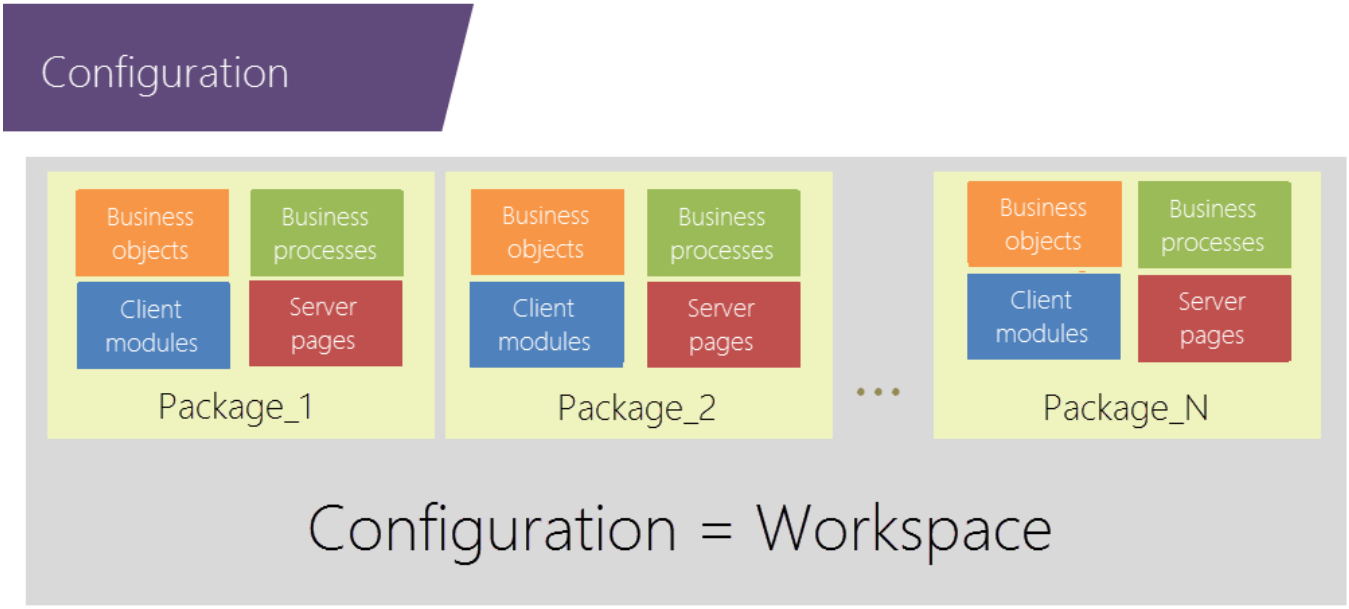
- Objects – entities that store data and connect a database table to a class on the server side.
- Business processes – customizable elements that are visual algorithms of user activities.
- Client modules.

All configuration elements are grouped in **packages**.

Packages are finite sets of functions that can be installed or uninstalled in configurations.

The final system functionality is formed based on the set of installed packages (Fig. 2).

Fig. 2. Bpm'online configuration



Packages, schemas, modules

Overview

Packages

A “package” is a combination of configuration elements (schemas, data, scripts, additional libraries) that implements specific functions.

The bpm'online package mechanism is based on the open/closed principle of object-oriented programming. According to this principle, all entities (classes, modules, functions) must be open for extension but closed for modification. This means that new functions must be implemented by adding new entities, rather than modifying the existing ones.

Each bpm'online product is a set of packages. To extend or modify system functions, a package with the corresponding changes must be installed.

There are two types of bpm'online packages:

- Base (pre-installed) packages include base functionality (such as Core, Base, Product packages), packages that extend system functions (such as phone integration packages) and packages created by third-party developers. Base packages are supplied with the system or can be installed as the **marketplace applications**.
- **Custom packages** are packages created by system users. They can be bound to the SVN storage.

Configuration elements from base packages cannot be modified. Any development of new functions and changes to existing functions are made in the custom packages only. A special replacement mechanism is used for this.

Package replacement mechanism

The mechanism replaces system objects in packages. If the behavior of an element from a base package must be

modified, a new inherited element is created in a custom package. This custom element is identified as a replacing one for its parent element from the base package. The replacement of elements is hierarchical. All changes that must be applied to the pre-installed element are implemented in a replacing custom package. As a result, the system will execute the logic of the replacing element instead of its parent base element.

NOTE

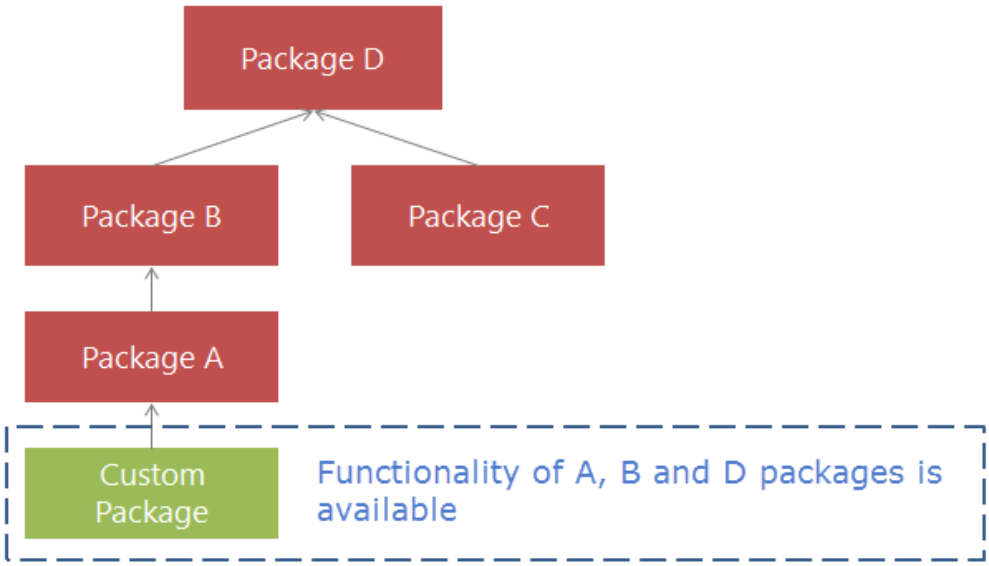
The replacement of a single base element can be implemented in several custom packages. The final implementation of a replacing element in a compiled configuration is determined by the hierarchy of all packages that contain replacing elements for the base package.

Package hierarchy

To use functionality from a different package, specify the dependency of the different package.

A dependent package extends or modifies the functionality of the package that it depends on. As a result, package dependency hierarchy is built. In the hierarchy, lower level packages can supplement or modify the functionality of any package that is higher in the hierarchy (Fig. 1).

Fig. 1. Package dependencies



A complete list of all packages that are installed in a workspace is displayed on the [Packages] tab of the [Configuration] section.

Packages can be installed from a ZIP archive (usually, those are pre-installed base packages) and version control system repository. The [Packages] tab also displays custom packages that were added in the current workspace.

Package composition:

1. Schemas – configuration elements of the system that define system functions.
2. External assemblies – third-party libraries that are required for development and integration with external systems. After installation, the libraries can be used in source code schemas.
3. SQL scripts – custom SQL scripts that are executed in the database during the package installation. SQL scripts may be required for transferring packages to other configurations if the package transfer requires database changes.
4. Data – section records, lookups and system setting values that are implemented in the current package may be required for transferring the package to other configurations if certain database records and values are connected to the current package.

For more information on working with packages, please refer to the "Development tools" articles.

Schema

A bpm'online configuration is a set of objects, processes, pages and modules.

The base element of a configuration is a schema. Configuration elements are schemas of different types. From the programming point of view, a schema is a core class inherited from the base Schema class.

Schema types:

Schema	Class	Purpose
Object schema	EntitySchema	These schemas can be used to manage database structure without needing to work with the database directly..
Client module schema	ClientUnitSchema	These schemas implement application client.
Source code schema	SourceCodeSchema	These schemas implement additional server logic of the application.
Business process schema	ProcessSchema	These schemas implement custom business processes.
Page schema	PageSchema	These schemas implement ASP.NET pages.
Business process task schema	ProcessUserTaskSchema	There schemas generate custom user tasks for business processes.
Report schema	ReportSchema	These schemas generate reports.
Image list schema	ImageListSchema	

Schemas are stored in the database as metadata. To edit schemas, various designers in the **[Configuration] section** are used: object designer, process designer, module designer, source code designer, etc.

Being inherited from the base Schema class, schemas of all types have a number of required properties and methods.

Required properties of schemas:

1. UId – unique identifier. When a new configuration element is added, its schema is created and assigned a unique identifier.
2. Name – schema name used for identification of the schema in program code.
3. Caption – schema title used for identification of the schema in the system interface.

Schema primary methods:

1. ReadMetaData – reads schema metadata from the database.
2. WriteMetaData – writes schema metadata into the database.
3. GetLocalizableValues – method that returns a collection of localized schema resources. These resources are used for storing and displaying names, captions, etc.

Collections of schema instances of different types are managed by special classes called “schema managers”.

Separate schema managers are used for different schema types.

Properties and methods of different classes are documented in the library of classes.

Object

The bpm'online data model is based on objects. An object is a business entity that declares a new ORM-model class on the server core level. On the database level, creating an object implies the creation of a new table with the same name and column composition as the created object. This means that in most cases each object is a representation of a actual table in the database.

There are base objects and custom objects.

- Base objects are non-editable and are stored in the base packages. They can be replaced in custom

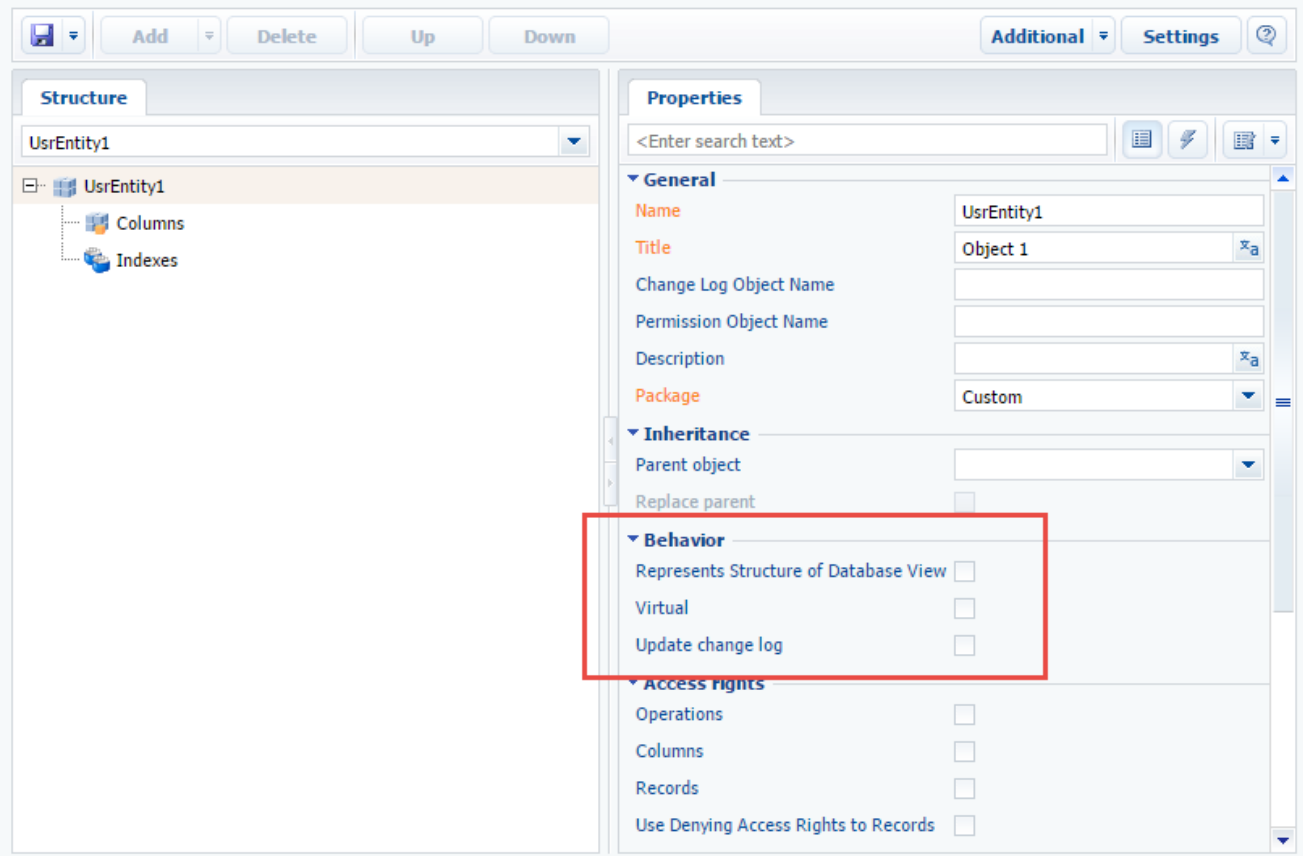
packages.
- Custom objects are objects created as part of configurations saved in custom packages.

There are 3 types of objects in bpm'online:

1. Objects connected to database tables.
2. Objects connected to database views.
3. Virtual objects used for creating hierarchies and implementing the inheritance mechanism (such as the BaseEntity entity).

Object type is set in the **object designer** (Fig. 2).

Fig. 2. Object types



A system object has three primary components:

1. Object schema – database table structure and properties. Object schema includes table columns (names and data types), indexes, access rights to object schema. Schema of an object is an instance of the *EntitySchema* class.

2. Object data – a data row of a table and methods for its processing. Each data row is an instance of the *Entity* class.

3. Embedded object process. Event model is implemented for each system object. Handling of object events is implemented through an embedded object process.

Module

Starting with version 7.0, the bpm'online client side has a module structure, which means that it is implemented as a set of functional blocks, each of which is implemented in a separate module. As part of the application operation process, loading of modules and their dependencies is done according to the Asynchronous Module Definition (AMD) approach.

The AMD approach declares the mechanism for determining and asynchronous loading of modules and their dependencies, which allows loading only the currently required data when working with the system. The AMD concept supports various JS frameworks. In bpm'online, the RequireJS loader is used for working with modules.

A *module* is a code fragment encapsulated in a separate block that can be loaded and executed independently.

The RequireJS loader provides the mechanism for declaring and loading modules, based on the AMD concept. General operational principles of the RequireJS loader mechanism:

1. Modules are declared in a special define() function, which registers fabric function for instantiating modules but does not immediately load the declared module when called.
2. Module dependencies are passed as an array of string values and not through the properties of the global object.
3. The loader loads all module dependencies passed as arguments to define(). Modules are loaded asynchronously, the load order is determined by the loader.
4. After all specified module dependencies are loaded, the factory function, which returns the module value, is called. Loaded dependency modules will be passed to the factory function as arguments.

Each client schema in bpm'online 7.x is characterized by at least one **client module**.

Client core provides mechanisms for working with modules:

- Provide API for accessing client modules.
- Determine the mechanism for message exchange and module loading.
- Provide access to base libraries, system enumerations and constants.
- Implement client mechanism to work with data.

Client module types

The following client module types are used in bpm'online:

1) Non-visual module

Non-visual modules implement system functionality, which, as a rule, is not connected to data binding or displaying data in the UI. Examples of non-visual models are business rules (BusinessRuleModule) and utility modules, which implement service functions. In the base version, non-visual modules have *Utilities, or *UtilitiesModule in their names.

2) View schema (visual module)

Visual modules implement the *View models* (ViewModel) according to MVVM template. These modules encapsulate data that is displayed in GUI control elements, as well as methods for working with them. Examples of visual modules are section, detail and page modules.

3) Extension module (replacing client module)

This type of module is designed for extending the functionality of base modules.

Application interface and structure

Contents

- **Main menu**
- **Sections**
- **Record edit page**
- **Details**
- **Mini-page**
- **Modal windows**
- **Communication panel**
- **Command line**
- **Action dashboard**

Main menu

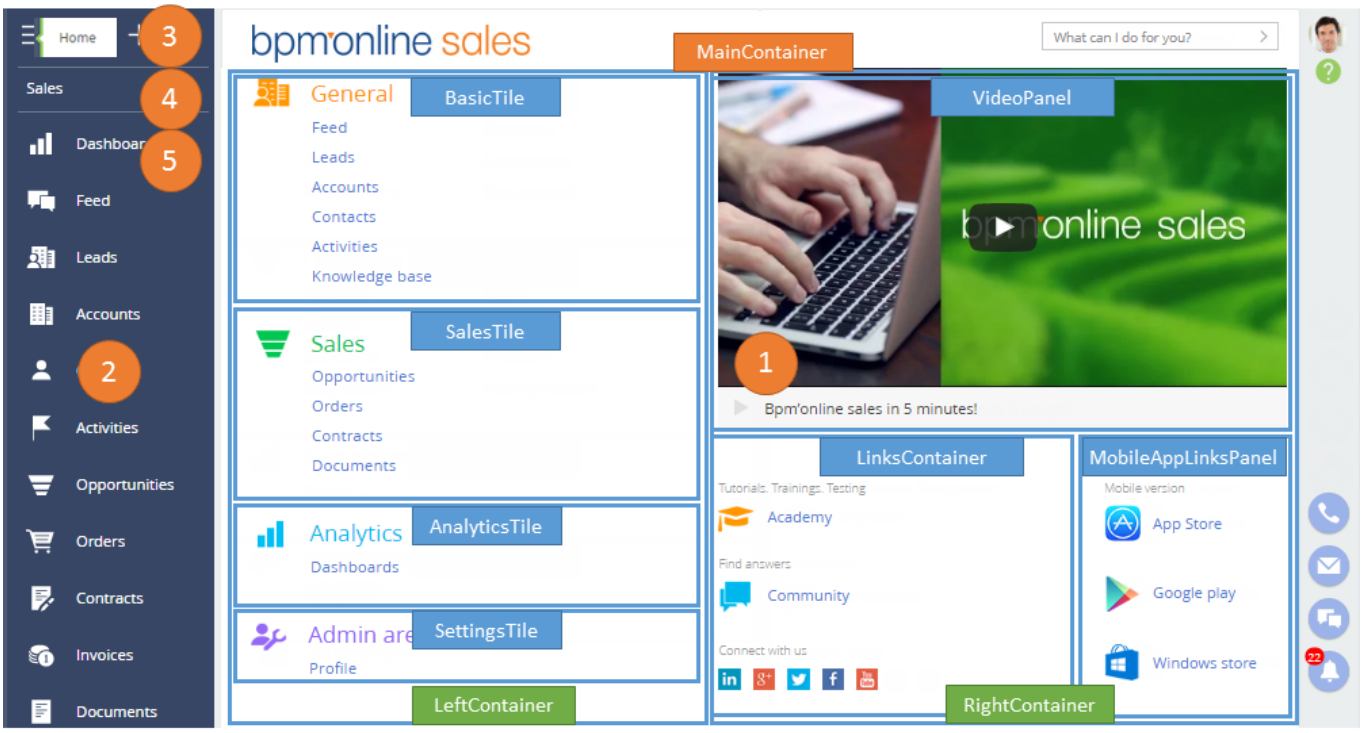
Difficulty level

Beginner Easy Medium Advanced

Overview

The main menu is displayed in the working area (1) of the UI after the application has been loaded (Fig. 1). The main menu can be opened using the "Menu" button located at the top (3) of the side panel (2).

Fig. 1. Main menu



Main menu commands used for opening system sections are also available in the section area (5) of the side panel (2). The list of available section navigation commands depends on the selected workplace.

Two schemas correspond to the main menu: the base schema of the *ApplicationMainMenu* business object and the product main menu schema inherited from the base product main menu schema *SimpleIntro*. For the *SalesEnterprise* product, the main menu schema is named *EnterpriseIntro*.

The element composition of the main menu UI depends on the product. All elements are placed in corresponding containers that are set up in the base or inherited schema of the main menu. The primary containers of the SalesEnterprise product include:

- **Menu main container** (*MainContainer*), which contains all main menu elements.
- **Section and setting container** (*LeftContainer*), which contains areas for commands that open sections and settings.
- **Resource container** (*RightContainer*), which contains areas with links to various resources.
- **Base functionality container** (*BasicTile*), which contains commands for opening sections that are available in all products.
- **Sales container** (*SalesTile*), which contains commands for opening sections of the Sales product family.
- **Analytics container** (*AnalyticsTile*), which contains command for opening the [Dashboards] section.
- **Settings container** (*SettingsTile*), which contains commands for opening the settings sections.
- **Video container** (*VideoPanel*), which contains video player and name of the linked video.
- **Link container** (*LinksContainer*), which contains links to training web resources and social networks.
- **Mobile app links container** (*MobileAppLinksPanel*), which contains links to bpm'online mobile app in various app stores.

Sections

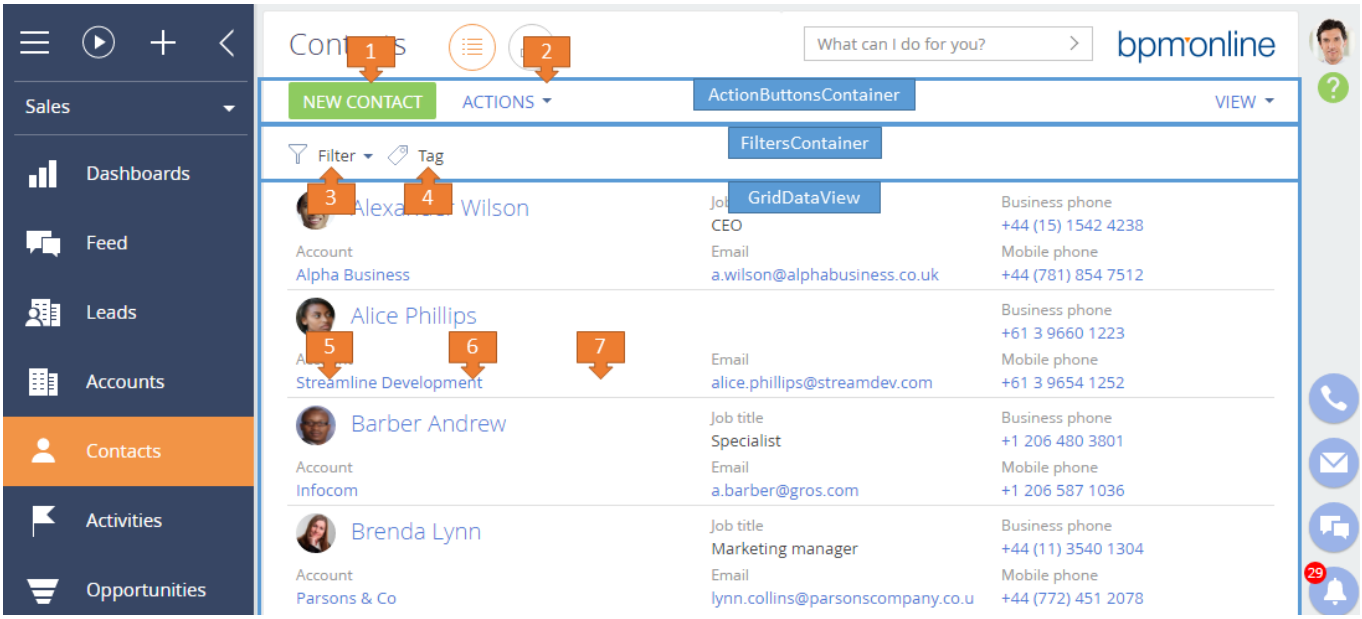
Difficulty level

Beginner Easy Medium Advanced

Overview

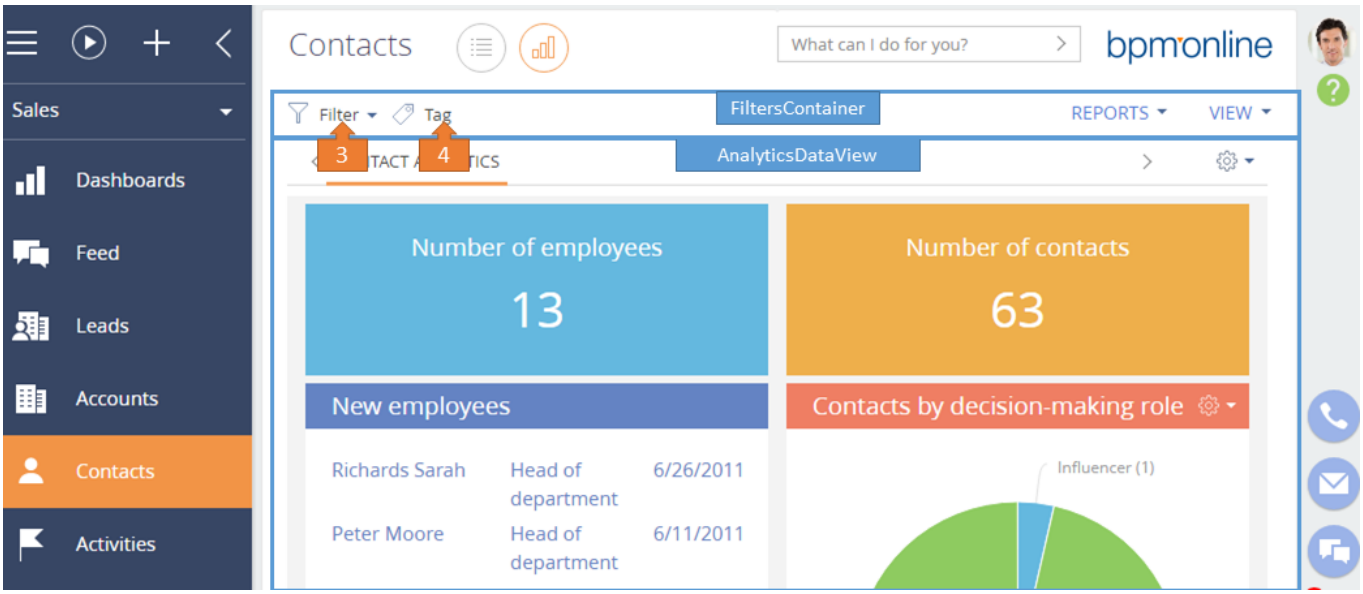
The "Section" element is displayed in the workspace of the user interface after selecting the appropriate menu item in the sidebar or on the main application page (Fig. 1).

Fig. 1. The [Contacts] section interface elements



As a rule, a section has two views: section list data display (Fig. 1) and section analytics display (Fig. 2).

Fig. 2. The [Contacts] section interface elements in the "Analytics" view



Each section corresponds to a certain schema. For example, the [Contacts] section is configured by the ContactSectionV2 schema. All sections are inherited from the parent BaseSectionV2 schema. More detailed information about schemata, their purpose and structure can be found in the "Schemata" article.

The user interface elements of the application relating to the section are placed in appropriate containers that are configured in the base or inherited section scheme. The main containers are:

- Action buttons container with a section action button (1) and a drop-down list (2)
- Filters container with filters (3) and tags (4)
- Section list data view container (GridDataView) with action buttons to edit (5), copy (6), and delete (7) the current record
- Section analytics data view container (AnalyticsDataView)

The order and content of the main section containers depending on the data display (Fig. 1) (Fig. 2)

The main interface elements and section functional elements are: list, section analytics, actions, filters and tags.

Section list displays records in tile or list view. Section list is displayed in the GridDataView container (Fig. 1). More information about lists can be found in the "**Section lists**" article.

Section analytics is used to display statistical data using diagrams, metrics or drop-down lists. Dashboards and user custom widgets are displayed in the in the container of the AnalyticsDataView analytics section (Fig. 2). More information about dashboards can be found in the "**Section analytics**" article.

Actions are functional section elements that are used to perform various operations with an active section list. Actions can be invoked with buttons (Fig. 1) in the ActionsButtons container or the active record container (Fig. 1). More information about actions can be found in the "**Section actions**" article.

Filter is used to search and filter records in the section. There are quick, standard, advanced filters and filter folders. The [Filter] buttons are located in the filters container (Fig.1 and Fig.2). More detailed information about filters can be found in the "**Filters**" article.

Tag is used to quickly search for section records by key words. The [Tag] buttons are located in the tags container (Fig.1 and Fig.2). More detailed information about tags can be found in the "**Tags**" article.

Section lists

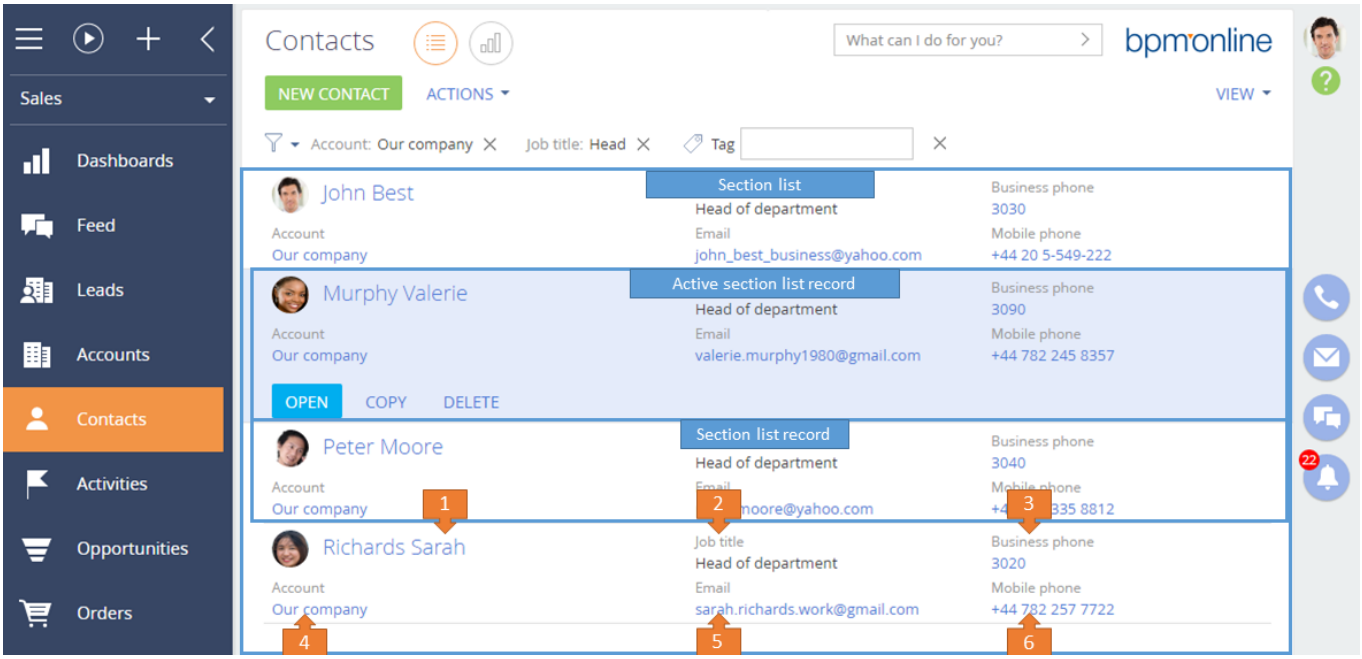
Overview

Section list is a list of records that can be displayed in one of two views.

A *tile view* displays the fields of each record in multiple lines. This is the default list view. In the [Contacts] section the following fields are displayed (Fig. 1)

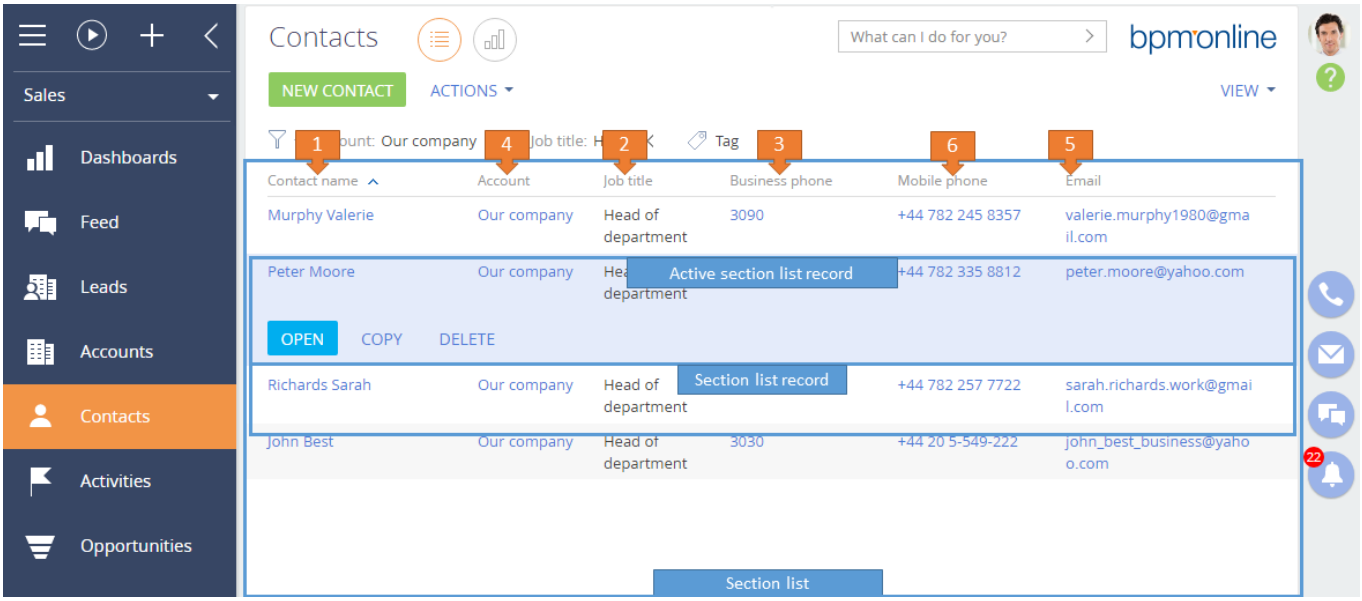
- Name (1)
- Position (2)
- Business phone (3)
- Account (4)
- Email (5)
- Mobile phone (6)

Fig. 1. The [Contacts] section list elements in the tile view



A *tile view* displays records as a simple table in which each record corresponds to one line (Fig. 2). The sequence of fields in the list view may not coincide with the sequence of fields in the tile view.

Fig. 2. The [Contacts] section interface elements in the list view



To avoid redundancy of the reported data, the list section displays only the most significant table columns. All data are displayed and edited on the section edit pages. Learn more about them in the "Edit page" article.

Each section has its own business object schema that describes the structure of a database table, which stores the data records. It also provides specific instructions for processing these data. From these data the list section is formed. Each table line corresponds to a section record. For example, the [Contacts] section corresponds to the Contact business object schema (Fig. 3) that contains Contact table properties (Fig. 4). The full list of model schema columns and their properties can be found using the object designer which is described in the "**Workspace of the Object Designer**".

Fig. 3. Contact object schema in the objects designer

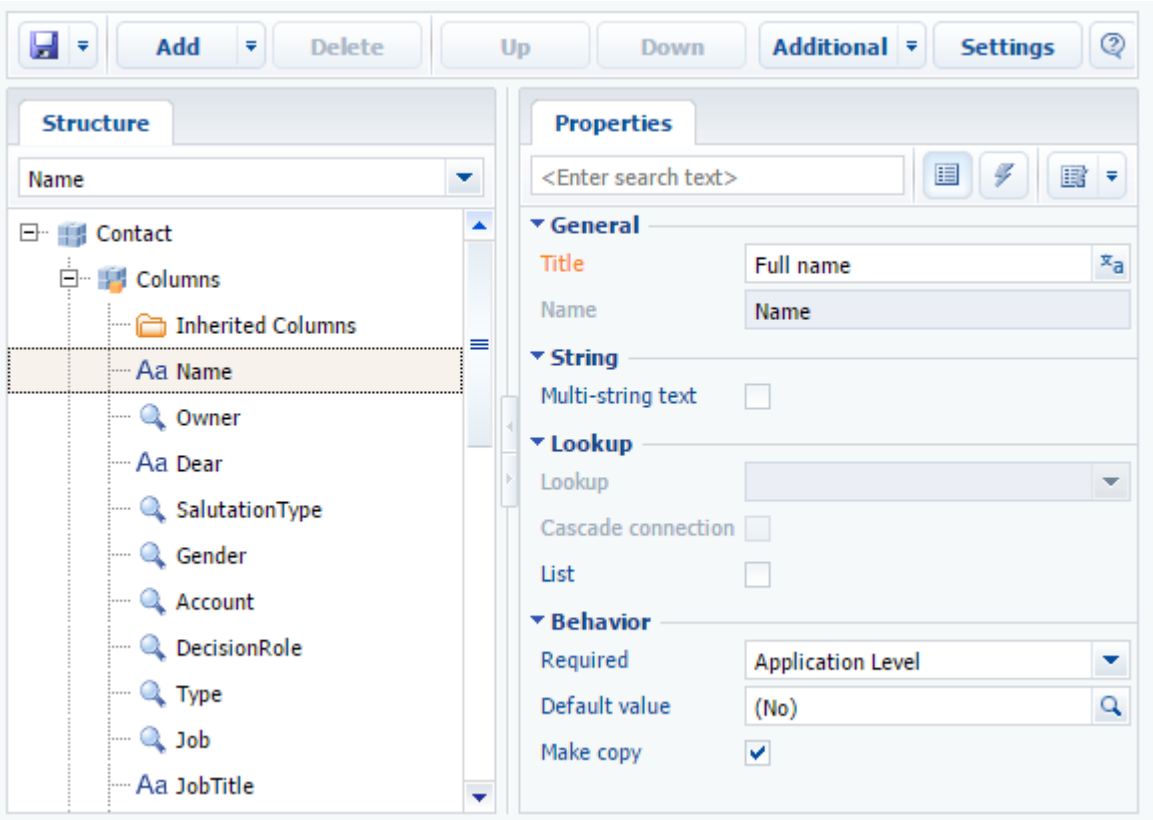


Fig. 4. Contact table

	Name	OwnerId	Dear	SalutationTypeId	GenderId	AccountId	DecisionRoleId
1	Supervisor	76929F8C-7E15-4C6...		7426FFB3-56E6-DF11-971B...	EEAC42EE-65B6-DF11-83...	E308B781-3C5B-4ECB-...	NULL
2	Alexander Wilson	76929F8C-7E15-4C6...	Wilson	F0B32E00-F46B-1410-AA84...	EEAC42EE-65B6-DF11-83...	8ECAB4A1-0CA3-4515-...	F71EE81D-0CAC-4
3	Alice Phillips	76929F8C-7E15-4C6...	Phillips	7526FFB3-56E6-DF11-971B...	FC2483F8-65B6-DF11-83...	B5E5BE36-F154-449F-...	NULL
4	Barber Andrew	76929F8C-7E15-4C6...	Barber	7426FFB3-56E6-DF11-971B...	EEAC42EE-65B6-DF11-83...	8EB03D91-EB4B-41A9-...	F71EE81D-0CAC-4
5	Brenda Lynn	76929F8C-7E15-4C6...	Collins	F4B32EE6-F36B-1410-AA84...	FC2483F8-65B6-DF11-83...	663D11E4-40D2-4F32-...	NULL
6	Caleb Jones	76929F8C-7E15-4C6...	Jones	F0B32E00-F46B-1410-AA84...	EEAC42EE-65B6-DF11-83...	E308B781-3C5B-4ECB-...	D2AB0E4E-57E6-D
7	Clayton Bruce	76929F8C-7E15-4C6...	Clayton	7426FFB3-56E6-DF11-971B...	EEAC42EE-65B6-DF11-83...	E1421CF4-3AF8-4BCE-...	F71EE81D-0CAC-4
8	Cook Regina	76929F8C-7E15-4C6...	Cook	7526FFB3-56E6-DF11-971B...	FC2483F8-65B6-DF11-83...	3906746F-3957-40BB-...	NULL
9	Grace Stewart	76929F8C-7E15-4C6...	Stewart	7526FFB3-56E6-DF11-971B...	FC2483F8-65B6-DF11-83...	8DDC6EA0-3EA3-48D2...	F71EE81D-0CAC-4
10	Henry Wayne	76929F8C-7E15-4C6...	Dr. Wayne	F0B32E00-F46B-1410-AA84...	EEAC42EE-65B6-DF11-83...	DDD2C965-CD43-4C9...	NULL
11	Hillam Jazlyn	76929F8C-7E15-4C6...		F0B32E00-F46B-1410-AA84...	FC2483F8-65B6-DF11-83...	7E5C0784-5CB9-4A08-...	NULL
12	James Smith	76929F8C-7E15-4C6...	Smith	7426FFB3-56E6-DF11-971B...	EEAC42EE-65B6-DF11-83...	FF7E089F-1FE9-4CA9-...	F71EE81D-0CAC-4
13	Jane Russel	76929F8C-7E15-4C6...	Russel	7526FFB3-56E6-DF11-971B...	FC2483F8-65B6-DF11-83...	96A7256B-7F4F-45A3-...	F71EE81D-0CAC-4
14	Jason Robinson	76929F8C-7E15-4C6...	Robinson	7426FFB3-56E6-DF11-971B...	EEAC42EE-65B6-DF11-83...	A0BF3E92-F36B-1410-...	NULL
15	John Best	76929F8C-7E15-4C6...	Best	7426FFB3-56E6-DF11-971B...	EEAC42EE-65B6-DF11-83...	E308B781-3C5B-4ECB-...	NULL
16	Johnson Diana	76929F8C-7E15-4C6...	Johnson	7426FFB3-56E6-DF11-971B...	FC2483F8-65B6-DF11-83...	B2AC3F26-A810-4B07-...	F71EE81D-0CAC-4
17	Jordan Anderson	76929F8C-7E15-4C6...	Anderson	F0B32E00-F46B-1410-AA84...	EEAC42EE-65B6-DF11-83...	8ECAB4A1-0CA3-4515-...	F71EE81D-0CAC-4
18	Kate Roberts	76929F8C-7E15-4C6...	Roberts	F4B32EE6-F36B-1410-AA84...	FC2483F8-65B6-DF11-83...	EAAE2286-F36B-1410-...	F71EE81D-0CAC-4

The layout and content of the displayed fields can be modified using the section wizard, or the drop-down list wizard available in the [View] button menu. More information on the section wizard can be found in the "Section wizard" article.

If you want to add a custom column in the business object schema and display it in the list, it can be done in two ways.

The first way is to use the section wizard. Create a replacing *Contact* object In the current custom package, which will inherit all the columns of the base *Contact* object from the *Base* package, to which a new custom column will be added. More information about the section wizard can be found in the "**Creating a new section**" article.

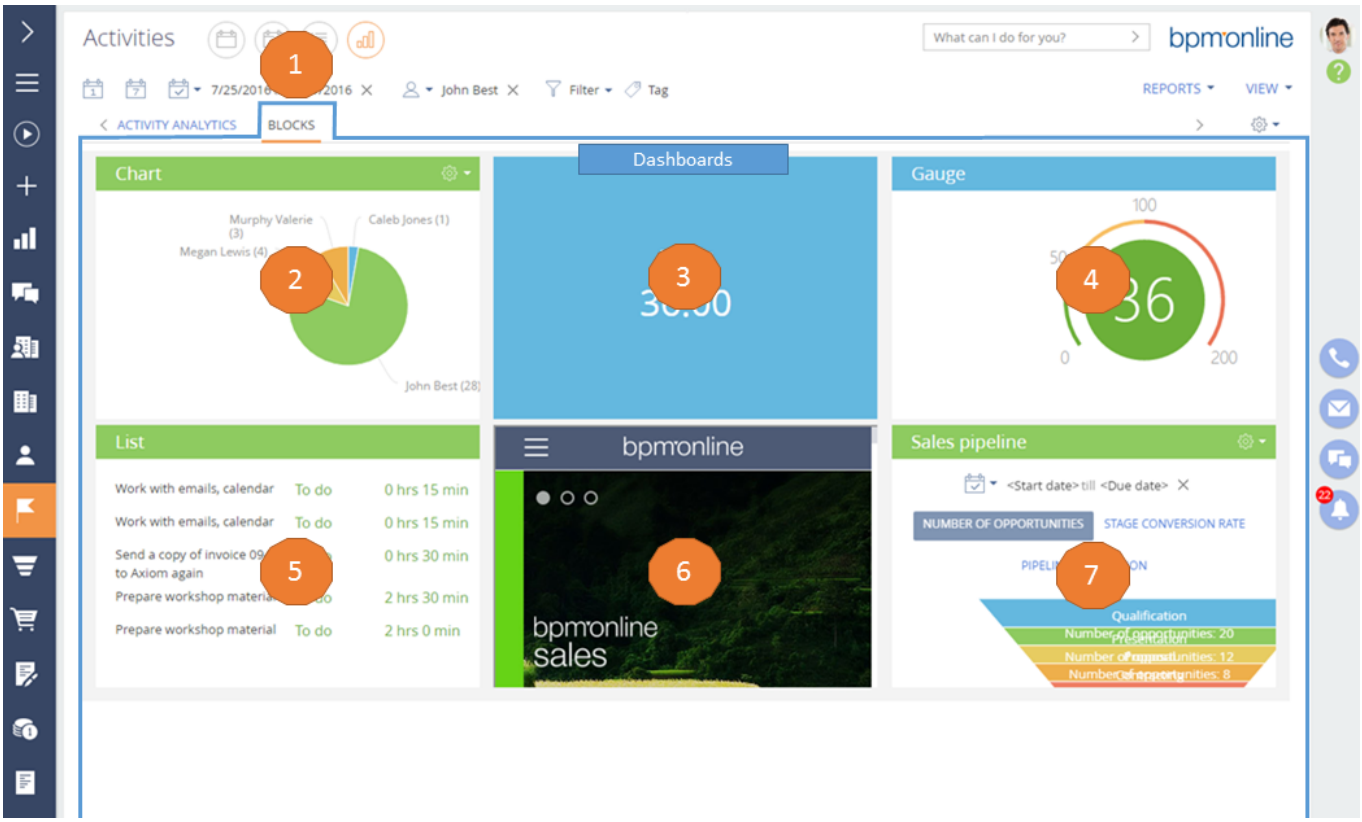
The second way is, using the object wizard, create a replacing *Contact* object in the custom package, which will inherit all the columns of the base *Contact* object from the *Base* package. Then, add the required columns to the replacing object and set up their properties. Next, using the section wizard or the drop-down list wizard, set up the

added columns display in the list. More information about the object wizard can be found in the "**Workspace of the Object Designer**".

Section analytics

Overview

Bpm'online analytics elements are used for gather statistics on section data. To open section analytics, go to the analytics (dashboards) view (Fig. 1) by clicking the corresponding button (1). To view analytics for all system sections, use the [Dashboards] section.



Analytics elements display information in special blocks called "dashboard blocks". The section area where the dashboard blocks are displayed is called the "dashboard panel". For more information on creating custom dashboard panels, please refer to the "Setting up dashboards" article.

The bpm'online application uses the following dashboard blocks (Fig. 1):

Chart (2). Charts are used to visually display data as graphs of various types or as a list of records. For more information on setting up charts, please refer to the "Setting up the “Chart” dashboard component" article in the User Guide.

Metric (3). A metric is used to display single numeric values, for example, the total number of current customers. For more information on setting up metrics, please refer to the "Setting up the “Indicator” dashboard component" article in the User Guide.

Gauge (4). A gauge displays data in relation to a scale.

List (5). A list displays filtered records. For more information on setting up lists, please refer to the "Setting up the “List” dashboard component" article in the User Guide.

Web page (6). This dashboard component displays web pages. This can be a search engine, currency converter page,

corporate website, etc

Sales pipeline (7). This dashboard component is used for sales stage dynamics analysis.

The widget dashboard component displays additional custom widgets.

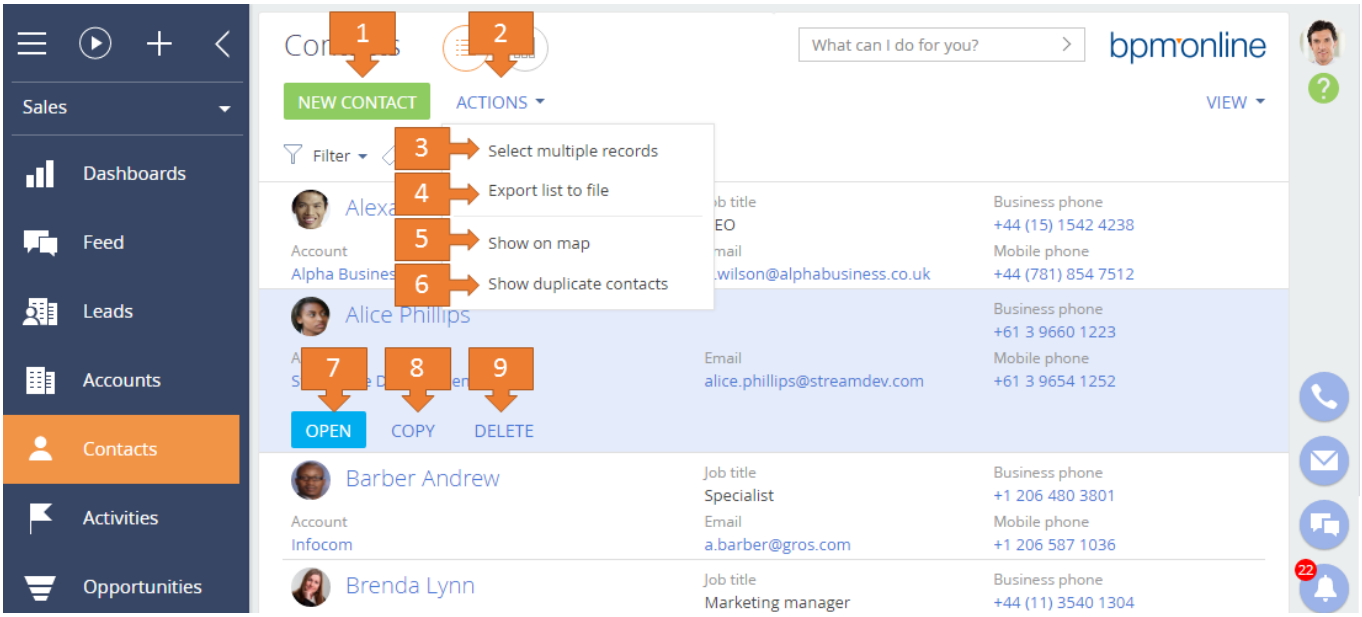
For more information on customizing analytics, please refer to the "Dashboard components setup" article.

Section actions

Overview

Actions are functional section elements that operate with one or multiple active list records. Actions can be invoked with buttons of different types, located in the action container of the current section and in the active record container (Fig. 1).

Fig. 1 Action interface elements of the [Contacts] section



User action interface elements are (Fig. 1):

- action buttons (1), (7)...(9);
- button with drop-down menu (2);
- action menu options (3)...(6).

There are standard section actions and additional actions that are unique for each section.

The standard actions are:

Add (1) — opens the edit page of a section and creates a new record.

Open (7) — opens the section edit page and populates it with the data from an active record.

Copy (8) — opens the section edit page, copies the data from an active record and creates a new record.

Delete (9) — deletes the active record.

Select multiple records (3)— sets the multiple selection mode of the registry entries.

Export list to file (4)— exports all list records of the current section to the *.csv file.

Additional actions are unique for each section of the application. For example, for the [Contacts] section, additional

actions are:

- Show on map (5).
- Go to contact duplicates (6).

Read more about additional actions in the "Bpm'online sales sections".

You can also create custom actions in bpm'online. Learn how to add custom actions in the "**Adding an action to the list**" article.

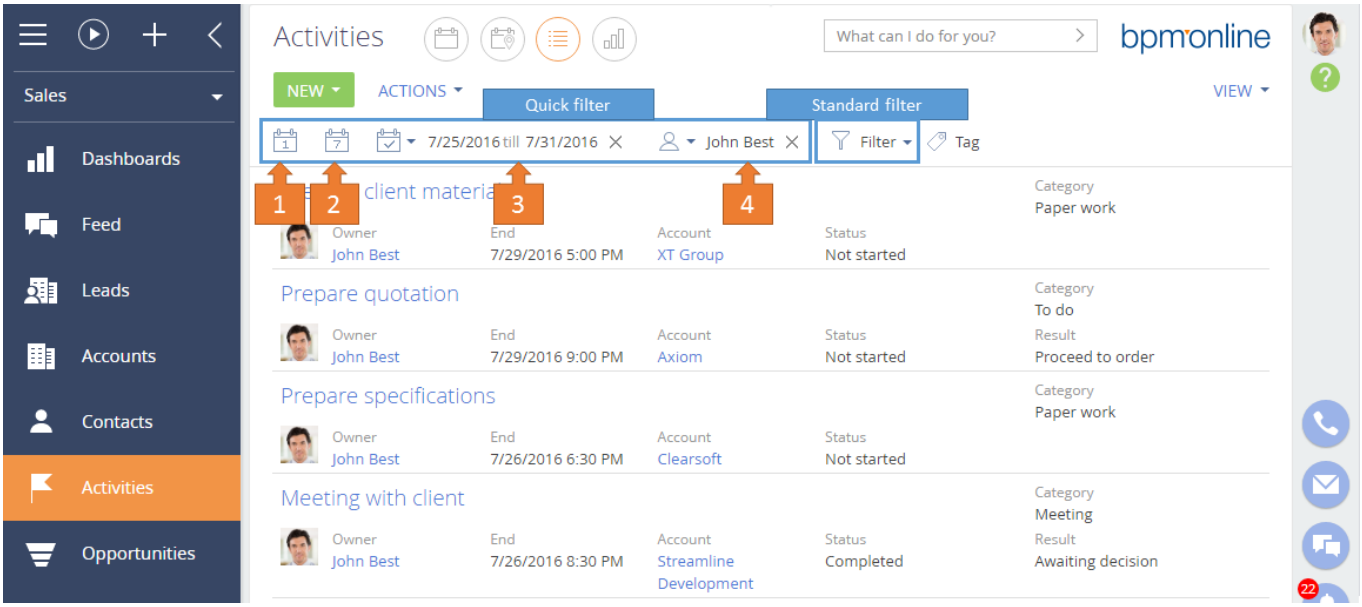
Filters

Overview

Filters are used to search and filter records in the sections. There are quick, standard and advanced filters and filter folders in bpm'online.

Filter management elements are displayed above the system sections list (pic 1). You can manage quick filters on the "Quick filter" dashboard, and standard and advanced filters and filter folders are set up in the "Filter" menu.

Fig. 1 Quick and standard filters of the [Activities] section



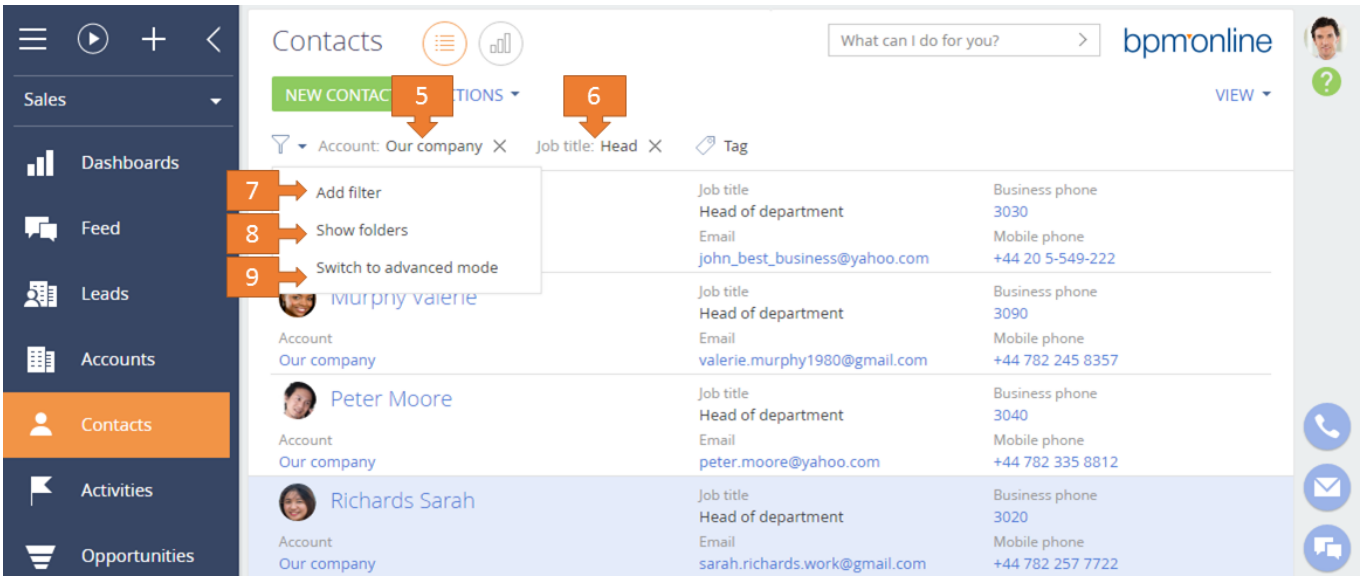
Quick filter is used to filter data based on most frequently used parameters. For example, activities of a single employee for a specified period of time are most often viewed in the [Activities] section. The following quick filters are designed exactly for this purpose (Pic 1):

- **Today** (1) filter displays records of the current day.
- **Current week** (2) filter displays records of the current week.
- **Select period** (3) filter displays records of the selected period, for example, "Yesterday", "Current week", etc. You can also set a custom period specifying the dates of its start and finish in the embedded calendar.
- **Select owner** (4) filter displays the activities of a single or multiple employees.

More detailed information about the filters is available in the "Quick filter" article.

A *standard filter* is used to search for records in the system sections based on specified values of one or more columns. For example, if you want to find all employees in the section (Pic. 2), you need to set up [Account] (5) and [Position] (6) filter fields.

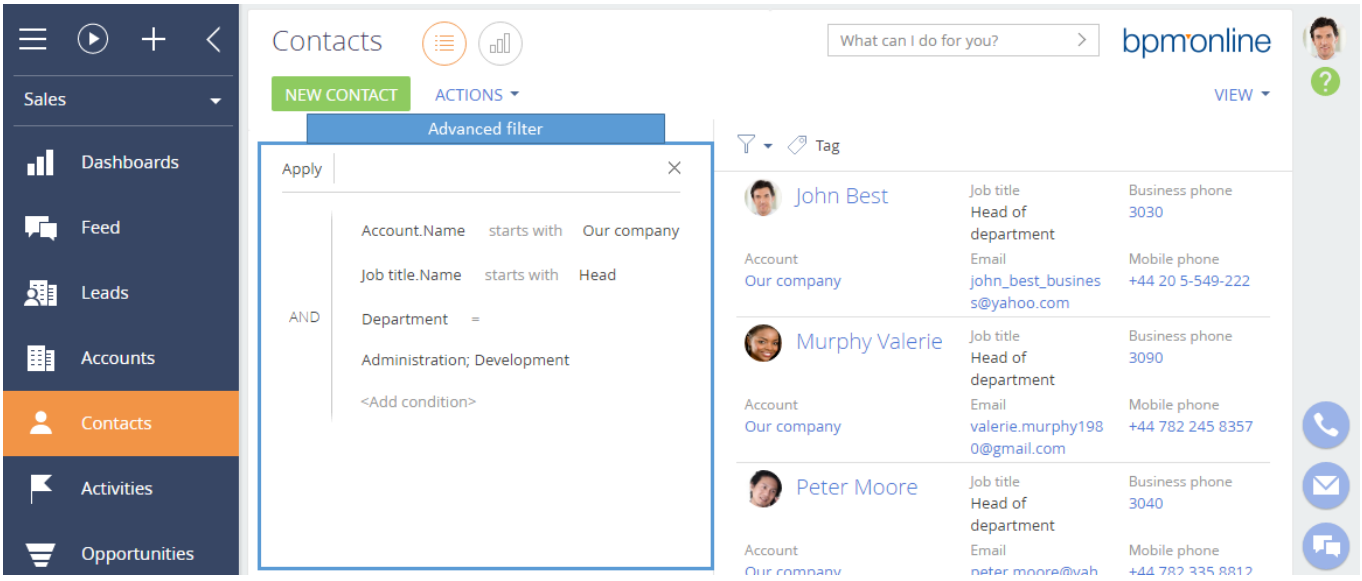
Fig. 2 [Contacts] section filter



You can set up standard filters by running the [Set condition] (7) command in the "Filter" menu. More detailed information about standard filter setup is available in the "Standard filter" article.

An *advanced filter* is used when you need to apply more complex filter consisting of several parameters and search criteria. For example, if you want all specialists to display only those who work in the departments "Development" and "Administration" (Fig. 3).

Fig. 3 Advanced [Contacts] section filter



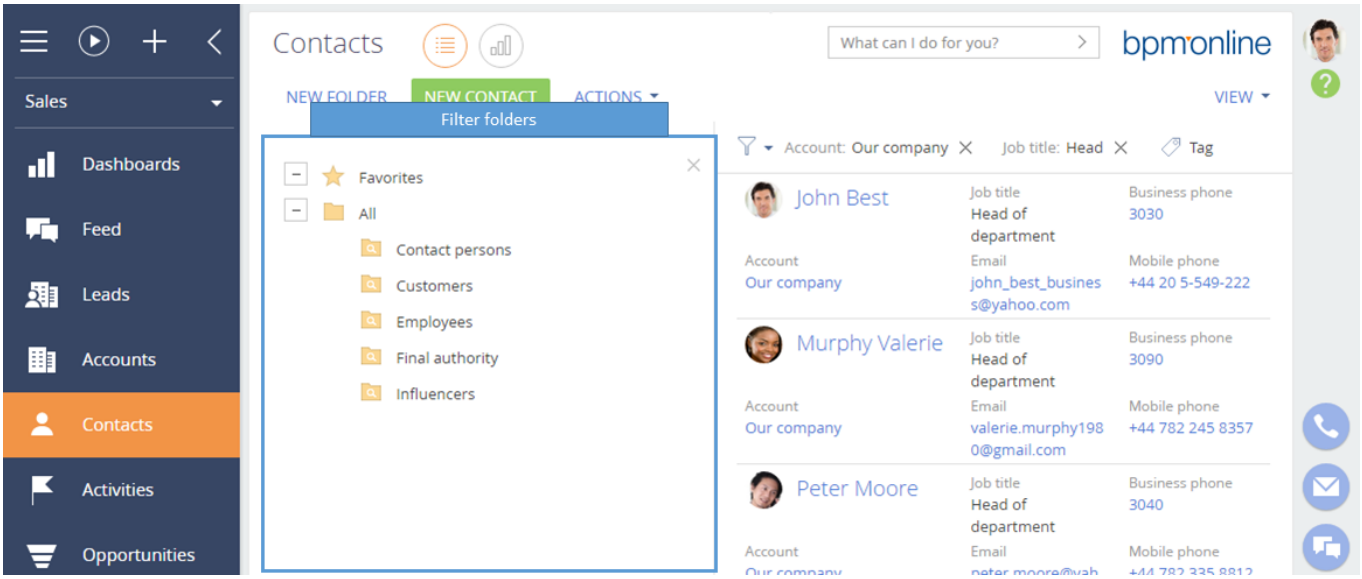
To set up the advanced filter, you must run the [Switch to extended mode] command (8, Fig. 2). More detailed information about advanced filter setup is available in the "Advanced filter" article.

Filter folders are used to segment records based on the specified filtering criteria. When selecting a folder, the section will display only those records that meet the filter folder conditions.

You cannot manually include or exclude records from filter folders. A record is automatically displayed in the folder if it meets the filter folder conditions. If a record no longer meets the filter folder criteria, it is excluded from the folder automatically.

To display filtered folders you need to run the [Show folders] command (8, Fig. 2). The existing folders will be displayed (Fig. 4) If necessary, you can create the required folder structure and define rules for their content.

Fig. 4 Filter folder of the [Contacts] section



More detailed information about filter folders is available in the "Working with folders" article.

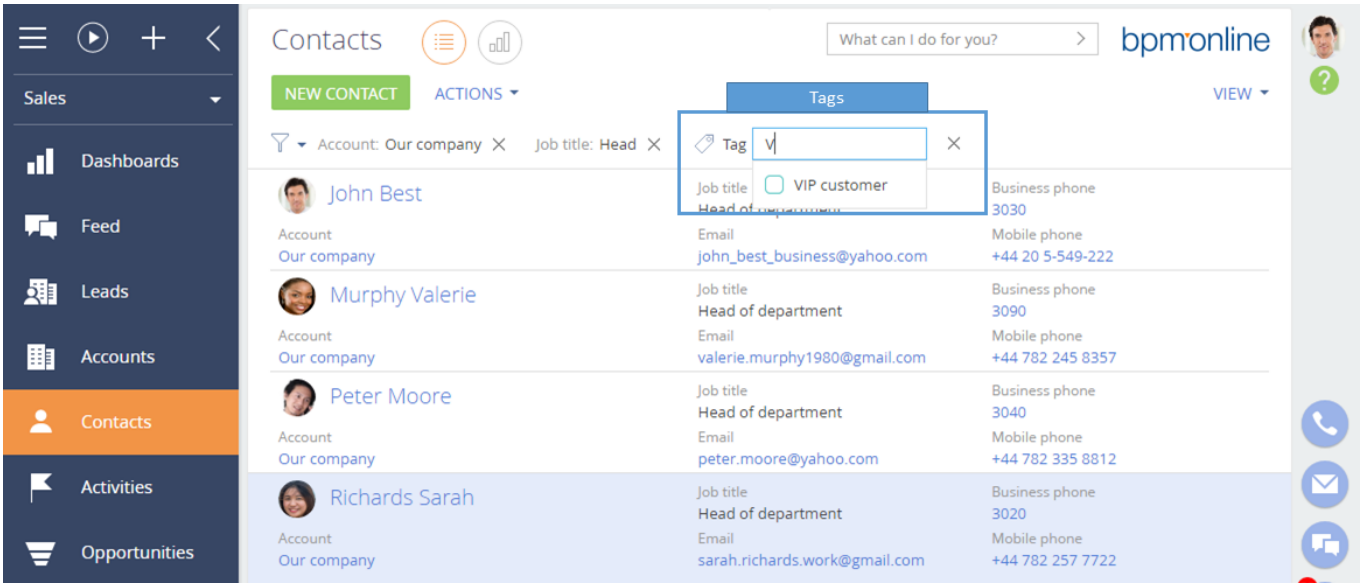
In bpm'online you can add user filters. More information about user quick filters is available in the "**Adding quick filter block to a section**" article.

Tags

Overview

Tags are used to quickly search for information by keywords. When you filter records by tags, the section will display only those records that have the selected tag.

Fig. 1 The [Contacts] section tags



Records are tagged manually. Each section of bpm'online sales has a separate list of tags.

Bpm'online sales contains the following tag types:

- **Personal** tags — can be seen and used only by users who created them. Neither system administrators nor managers can see the personal tags of employees. Personal tags are displayed in green.
- **Corporate** tags — displayed for all employees of the company. Any employee can set or clear a corporate tag. Any employee/role with access rights to perform "Corporate tag management" operation can create new corporate tags Corporate tags are displayed in blue.
- **Public** tags — displayed for all employees and for self-service users. Any employee can set or remove a public tag. Any employee/role with access rights to perform "Public tag management" operation can create new corporate tags Public tags are displayed in red.

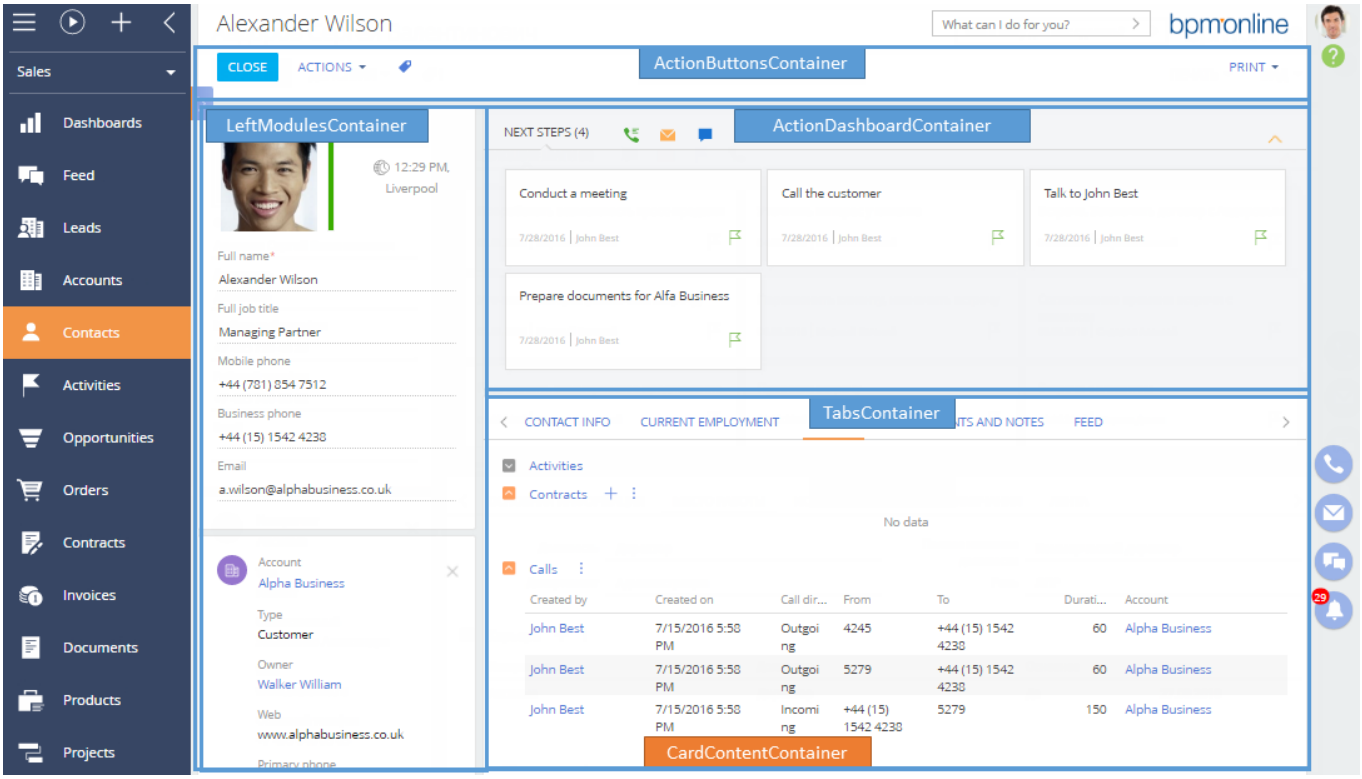
More information about creating and configuring tags can be found in the "Working with tags" article.

Record edit page

Overview

An edit page is a container with a set of fields for entering and editing the columns in the section object schema (see "**Section list**"). An edit page opens when you create or edit a section list record. Every section contains one or more edit pages.

Fig. 1 The [Contacts] section edit page interface



Every edit page has its own client module schema. For example, the [Contacts] section setup is performed in the *ContactPageV2* schema of the *UIv2* package. All edit page schemas are inherited from the parent *BasePageV2* schema of the *NUI* package. More information about packages, objects, and modules can be found in the "**Packages, schemas, modules**" article.

The user interface elements related to the edit page are located in the corresponding containers that are configured in the base or inherited the edit page schema. The main edit page containers include (Fig. 1):

- the container for the action buttons (ActionButtonsContainer);

- the container for the left side of the edit page (LeftModulesContainer), which contains the main input fields;
- the action dashboard container (ActionDashboardContainer) with the action panel and workflow bar;
- the tabs container (TabsContainer) with input fields, grouped by any attribute (Fig. 1).

If you need to add custom input fields to an edit page, it has to be replaced with a custom edit page. Read more about schemas replacement and inheritance in the article "Creating custom schemas. Replacement and inheritance". You can learn how to add various interface elements to the edit page in a series of articles in the "Page configuration" section.

Details

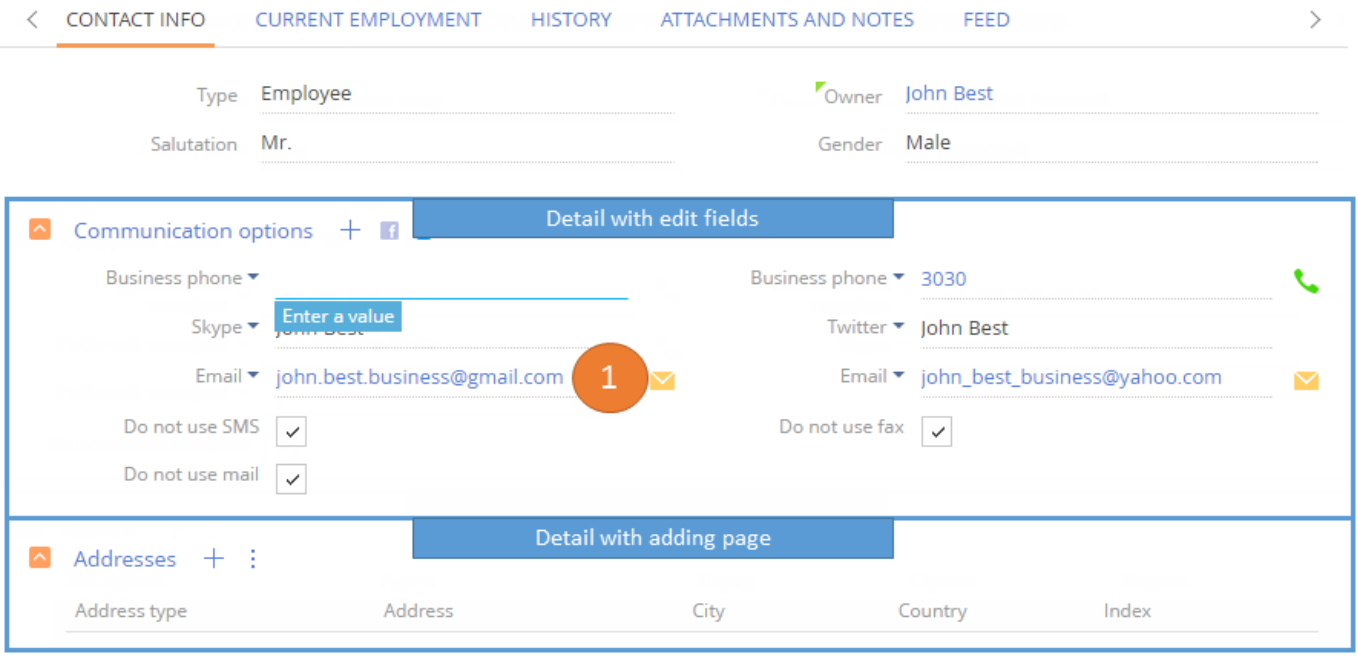
Overview

The purpose of details is to display supplemental data for a primary section object. The section details are displayed on the section edit page tabs in the tabs folder.

Depending on the method of entering and displaying data, there are following types of details.

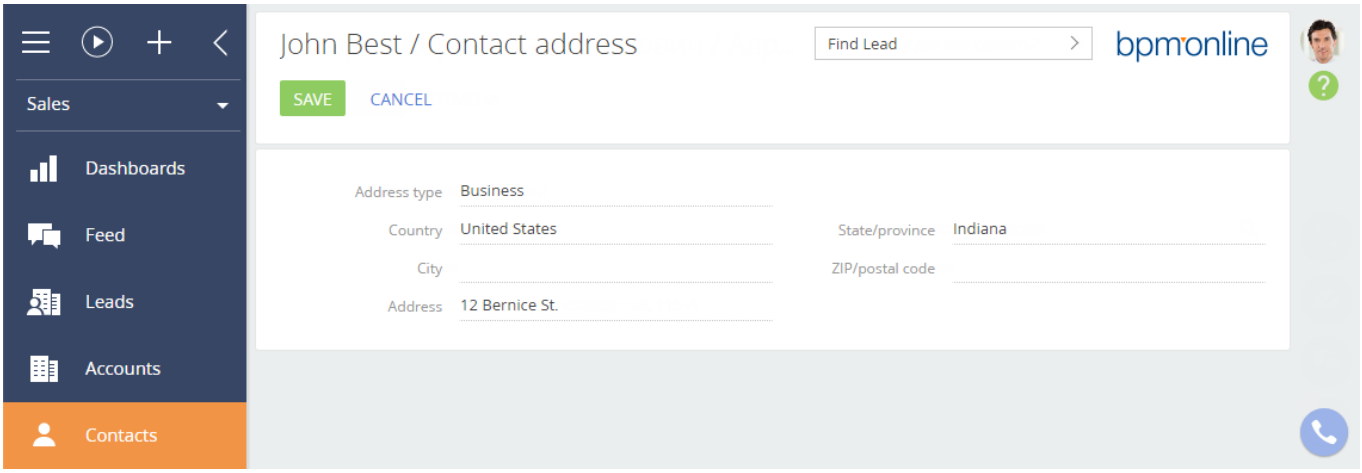
A *detail with edit fields* — data are filled in and edited in the detail data fields (Fig. 1). If necessary, you can add a new field to a detail (1). For example, the [Contact communication options] detail.

Fig. 1. The details with edit fields and the [Contacts] section data adding page



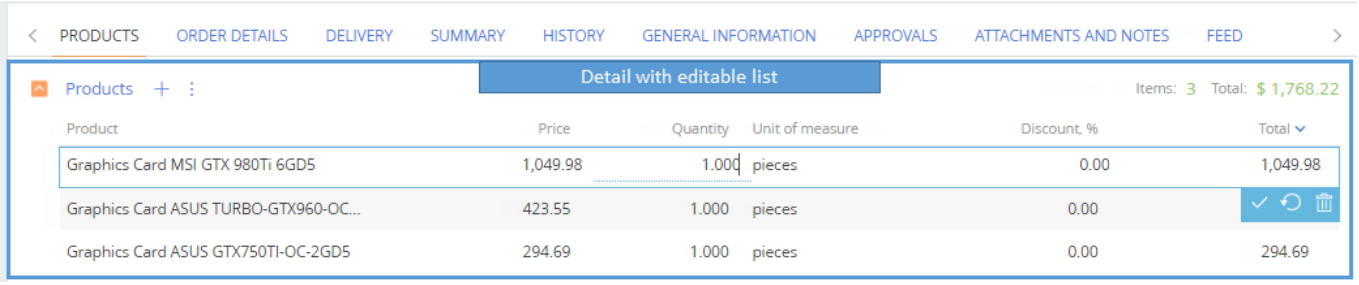
A *detail with adding page* — data are entered and edited on the detail edit page. For example, the [Contact address] detail (Fig. 1) — each address is entered and edited on the "Contact address" page (Fig. 2).

Fig. 2. The "Contact address" detail adding page



A *detail with editable list* — data are displayed as a list and are entered and edited directly in the list. For example, the [Order product] detail (Fig. 3)

Fig. 3. The [Product in order] detail with editable list



A *detail with selection from lookup* — data are selected from a lookup displayed in the modal window. For example, the [Lead product] detail (Fig. 4) — data are selected from a lookup in the modal dialog box "Select: Product" (Fig. 5).

Fig. 4. The [Lead product] detail with selection from lookup

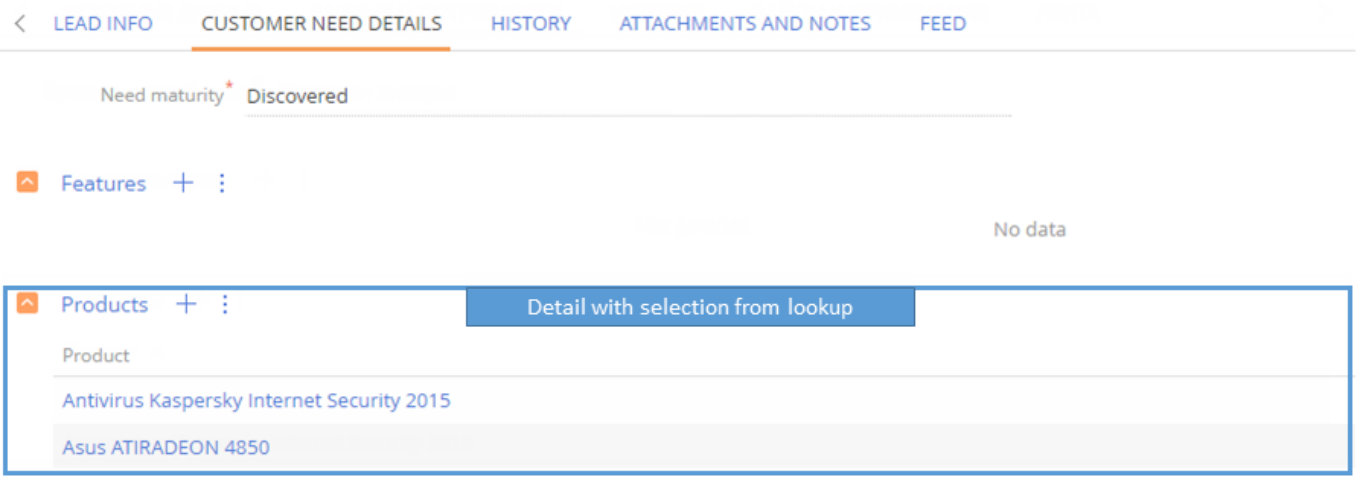
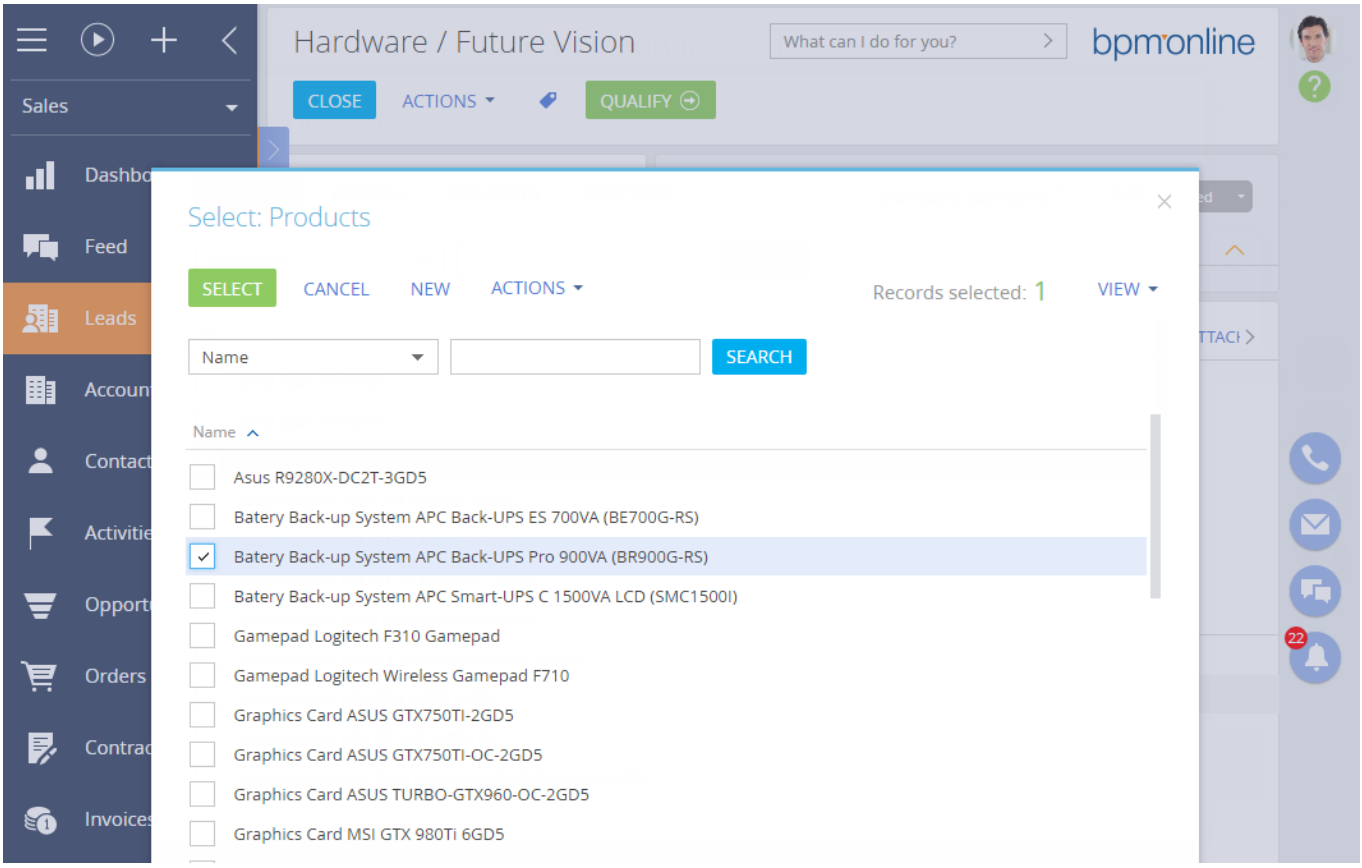


Fig. 5. Selecting products from the [Lead product] detail



Each detail corresponds to a business object schema connected to the object of the current section. For example, [Contact addresses] detail corresponds to "Contact addresses" (*ContactAddress*) object schema of the *Base* package. The connection is set up based on the mandatory [Contact] column of the detail object.

The content, location and behaviour of the user interface detail elements are configured by the detail schema. For example, the [Contact Addresses] detail is configured by the "Contact addresses detail" schema (ContactAddressDetailV2), that inherit "Base detail scheme with a list" (BaseAddressDetailV2) of the UIv2 package. Application details schemata are inherited from the base detail schema with a list (BaseGridDetailV2) and base detail schema (BaseDetailV2) of the NUI package.

A detail edit page is configured by the edit page schema. For example, the [Contact addresses] detail edit page properties are configured by the "Contact address page" schema (ContactAddressPageV2), which is inherited from the "Base address page" (BaseAddressPageV2) of the UIv2 package.

In bpm'online you can create custom details. More information about custom detail creation depending on its type can be found in the next articles:

- **Creating a detail in wizards**.
- **Adding an edit page detail**.
- **Adding a detail with an editable list**.
- **Creating a detail with selection from lookup**.
- **Creating a custom detail with fields**.
- **Adding the [Attachments] detail**.

Mini-page

Overview

A mini-page is designed to quickly view and add information about a section record without opening the edit page. Mini-page can be displayed by hovering the cursor over hyperlinks that lead to, for example, the contact edit page (Fig. 1) and account edit page (Fig. 2).

Fig. 1 Contact mini-page

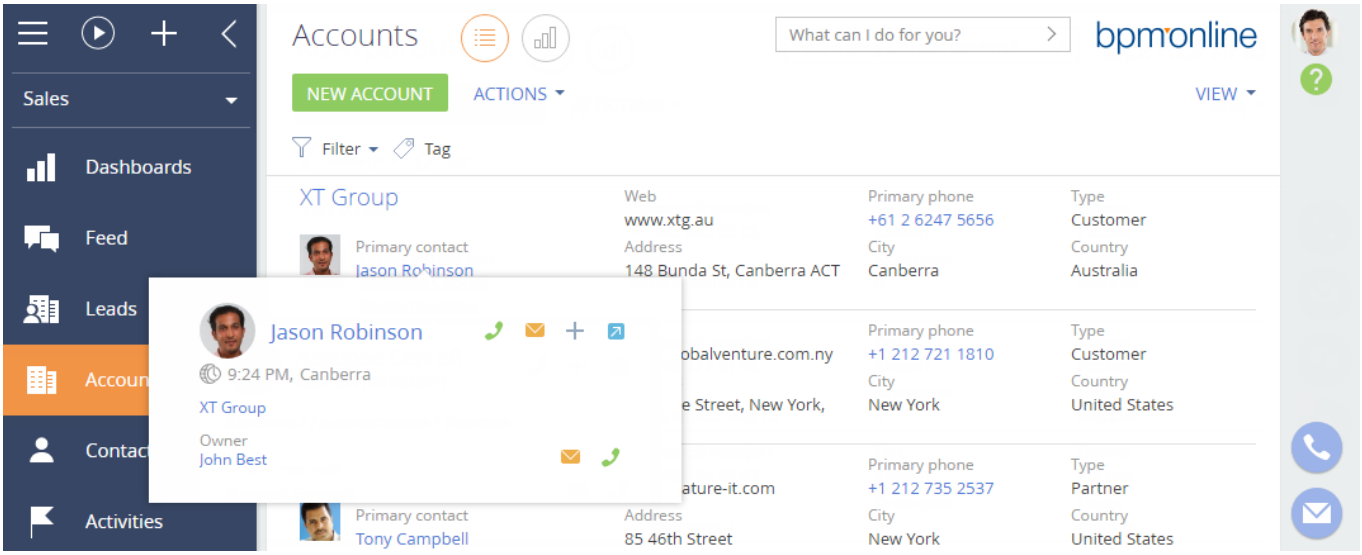
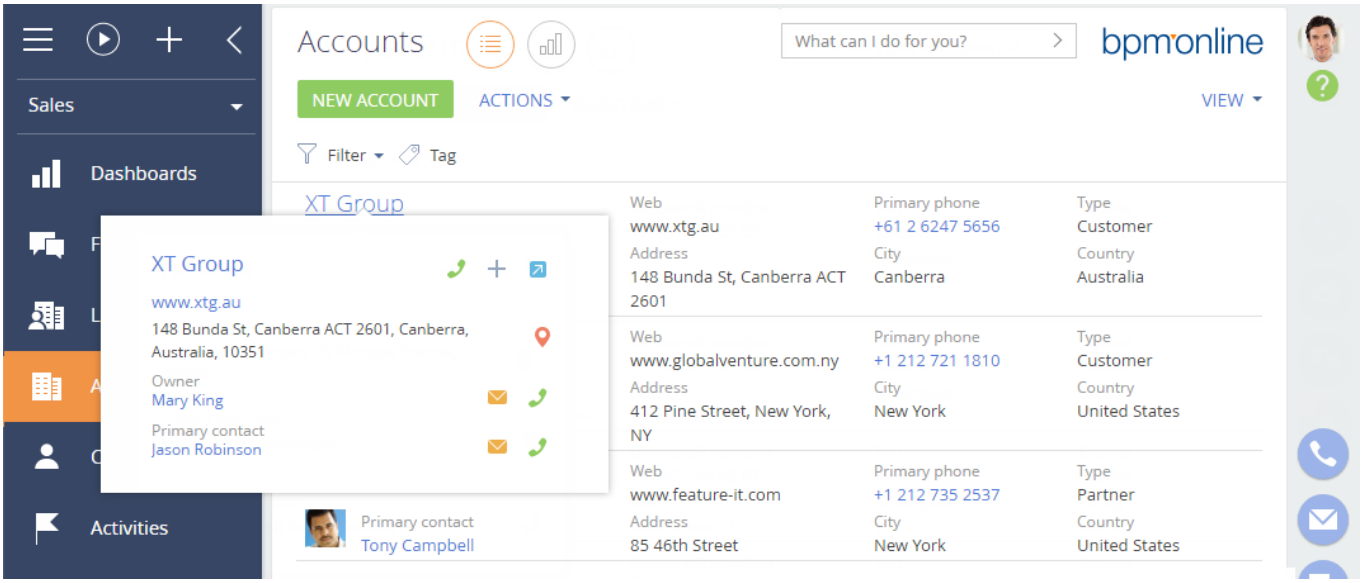


Fig. 2 Account mini-page



Using the mini-page, you can make calls, write and send emails, and create tasks or contacts. You can also view a location on the map. More information about mini-pages can be found in the "Mini-page" article.

The contents, location and behavior of user interface elements are configured in the schema of the mini-page view model schema. For example, the contact mini-page is configured through the *ContactMiniPage* schema, and the account mini-page through the *AccountMiniPage* schema of the *UIv2* package. The parent schema for all mini-pages schemas is the *BaseMiniPage* schema, part of the *NUI* schema.

If necessary, you can create a custom mini-page. An example of creating a custom mini-page for records in the [Knowledge Base] section is described in the "**Creating pop-up summaries (mini pages)**" article.

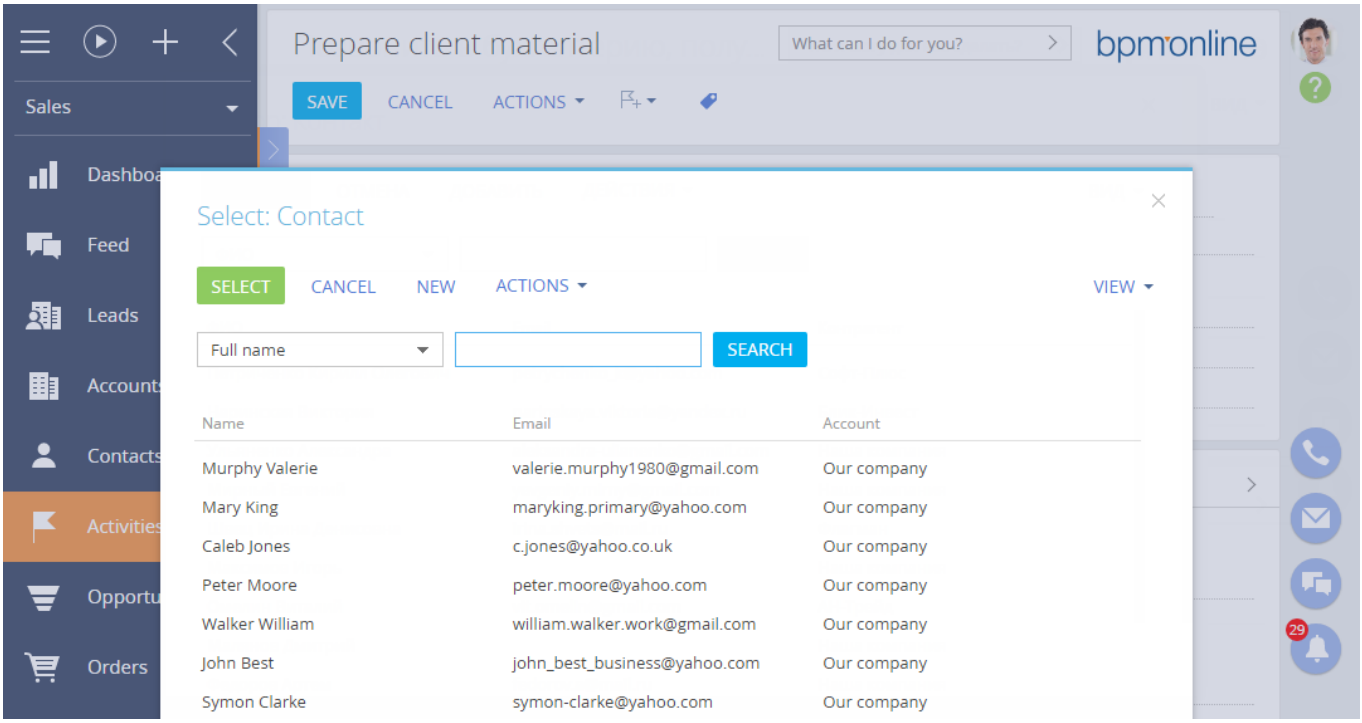
Modal windows

Overview

Modal windows display data in a pop-up dialog box. When a modal window opens, the page from which the modal window was opened does not close, and no new pages are opened in the process. Thus, the page that the modal window displays is not shown in the browser history.

The modal windows are used to display and select data from various lookups, for example, when selecting an activity assignee from the contact lookup (fig.1).

Fig. 1. Modal window for selecting activity assignee from the contacts lookup



General properties and behavior of modal windows are specified in the *ModalBox* and *ModalBoxSchemaModule* modules of the *NUI* package. The modal window for selecting data from lookups is called in the *LookupUtilitiesV2* module.

Communication panel

Overview

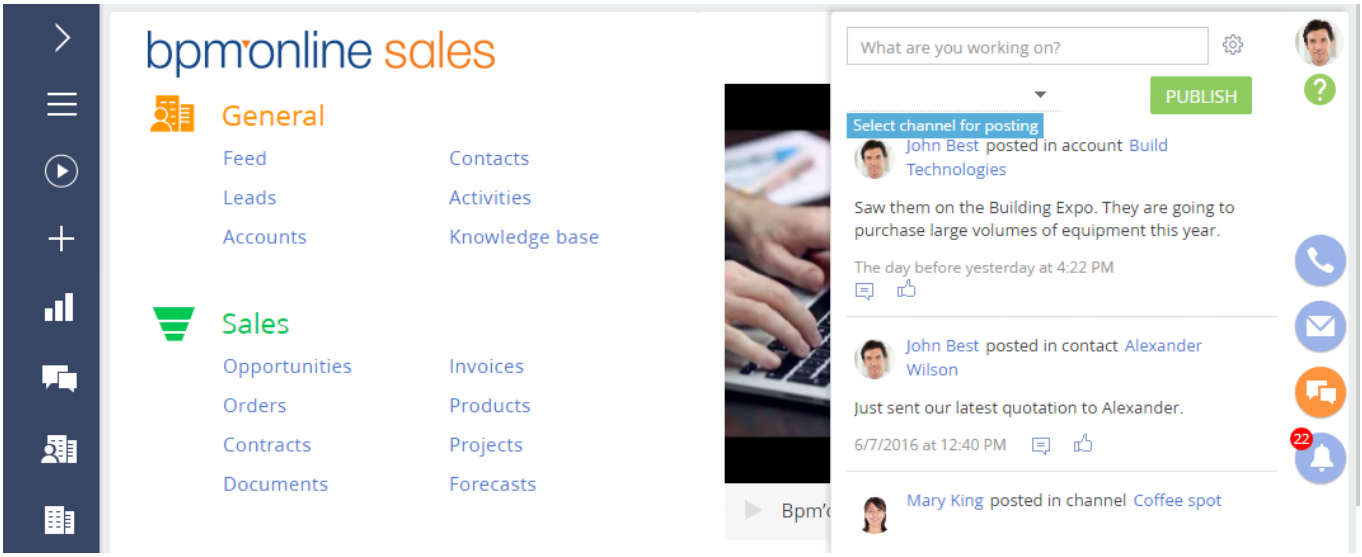
The communication panel is designed for user interaction with clients and colleagues without interrupting execution of the current task. Using the communication panel, you can make calls, process unread mails and post in the enterprise social network.

The communication panel contains the following tabs (Fig. 1):

- Calls (1) — enables you to accept and make calls directly in the application. Read more about the possibilities of telephony in the "Managing calls in bpm'online" article.

- Email (2) — designed to work with email. Features of configuration and integration with email services are described in the "Working with email" article.
- Feed (3) — displays messages of the [Feed] section and is used to view messages in subscribed channels and to add new posts and comments. More information can be found in the "[Feed] section" article.
- The notification center displays notifications about various events in the system. It is described in detail in the "Notification center" article.

Fig. 1 Communication panel



The communication panel is configured in the *CommunicationPanel* scheme of the *UIv2* package. The "Calls" tab is configured in the inherited *CommunicationPanel* scheme of the *CTIBase* package . The communication panel tab buttons (1...4) are located in the *communicationPanel* container, and the tabs in the *rightPanel* container (Fig. 1).

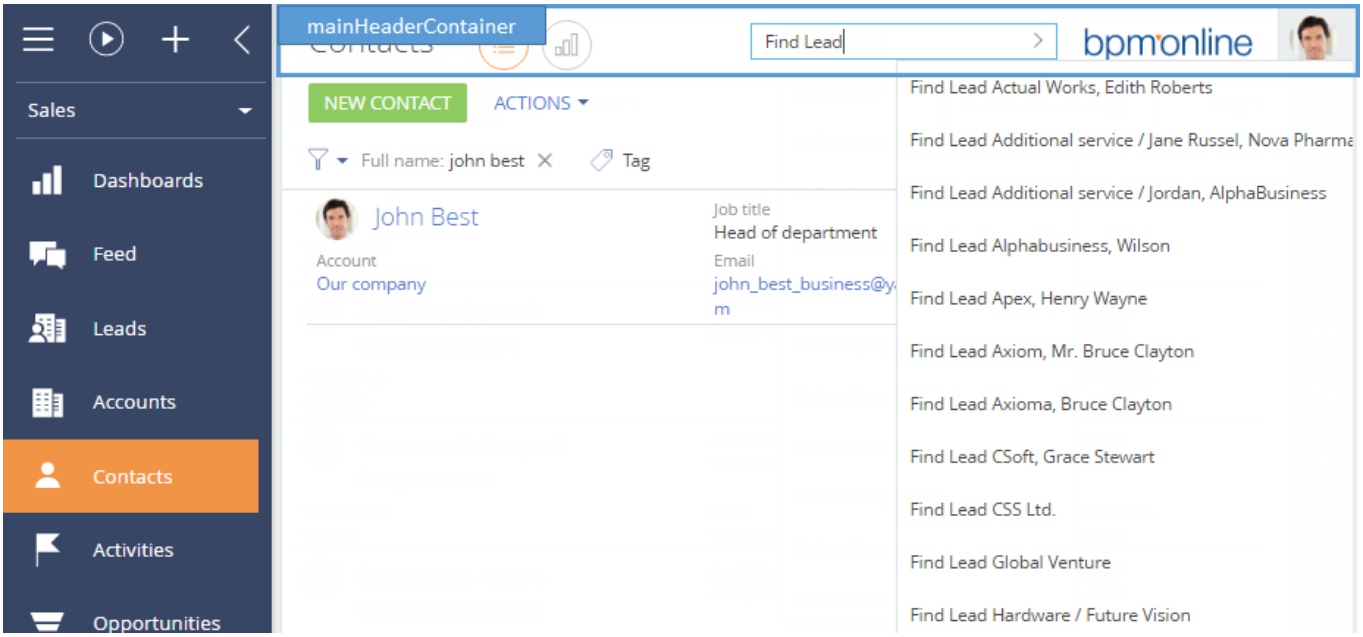
Command line

Overview

The command line enables quick access to the most frequently performed operations.

To run a command, type it in the command line and click "Execute command" (Fig. 1) or press [Enter] on the keyboard. If you enter an incomplete command, the system will offer a list of possible commands in the drop-down list.

Fig. 1 Command line



The features of the command line are:

- Navigation — "Go to...” a section
- Search for records —"Search ..." for contacts, accounts or records
- Creating records — "Create..." a record
- Start business process — "Start process ..."
- Create custom commands with the "Create custom command".

Read more about the possibilities of the command line in the "Command line" article.

The command line input field is located in the *mainHeaderContainer* conatiner (Fig. 1).

To track commands and their execution in the system, use the *CommandLineService* service. To store commands in the system, a database table is used. The structure of database table is described by the *Command* object schema. The command parameters are described by using the *CommandParams* object schema. To display the list of available commands for autocompletion and other functionalities of the command line, the *CommandLineModule* module is used.

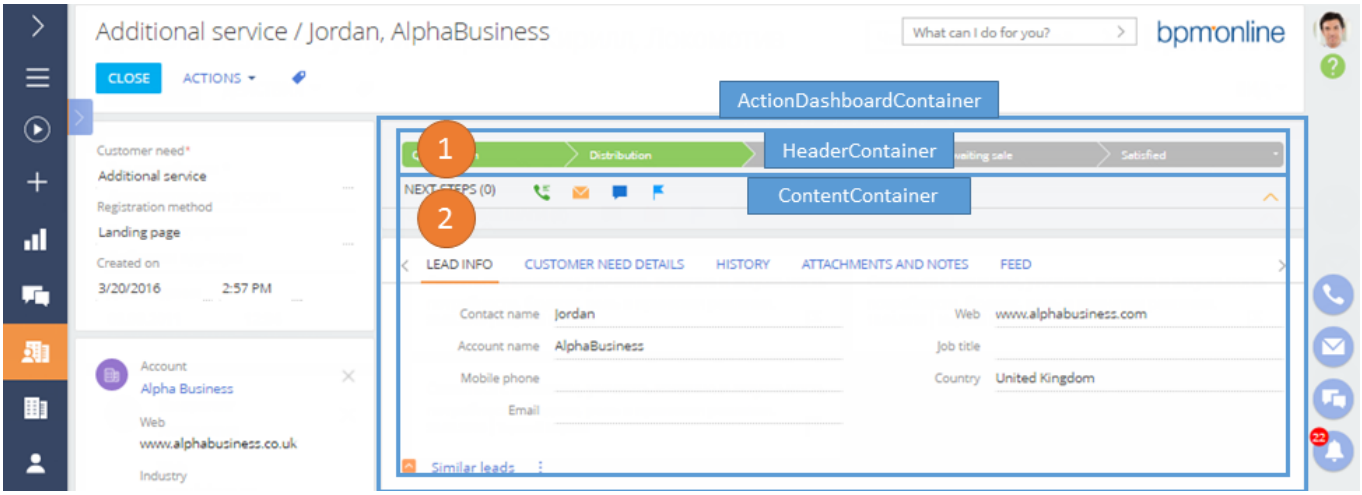
Action dashboard

Overview

The action dashboard is designed to display information about the current state of a record and consists of two parts (Fig. 1):

- The *Workflow bar* (1) — shows the business process stage status.
- The *Action panel* (2) — enables you to move on to the activity, work with email or feed, without leaving the section. The action dashboard displays business process activities that are connected to the section object by the corresponding field. The action panel can also display an auto-generated page, pre-configured page, question or object page.

Fig. 1 Action dashboard in the [Leads] section.



The action dashboard is located in the *ActionDashboardContainer* container of the section record edit page. The workflow bar is located in the attached *HeaderContainer* container, and the action panel — in the *ContentContainer*.

The arrangement of the elements of the action dashboard is configured by the *BaseActionsDashboard* view model schema and the *SectionActionsDashboard* inherited schema of the *ActionsDashboard* package.

How to start the development

Development of custom solutions in bpm’online is a complex and laborious process. Keep the following sequence of action to avoid difficulties.

1. Set up the development process.

Organization of development process depends on the volume and complexity of planned custom modifications of bpm’online. To add small and simple modifications to the bpm’online functions you do not need to set up specific processes. Implement these modifications in the application used for development and transfer them to the working version of bpm’online after preliminary testing.

To add complex and extensive custom functionality you need to set up three working environments (development, pre-production and production environments). Developed functionality can be transferred between these working environments only if it fits specific criteria. For more information on development of complex functionality, see "**Development process organization**" article.

The sequence of development and transfer of the developed solution to the working application depends on the organization of the development process. For more information about development sequence, see the “**Recommended development sequence**” article.

NOTE

To develop complex project solutions, use the recommendations provided in the "Terrasoft Project Life Cycle” documentation.

2. Select and configure development environment

To develop simple functionality that requires small modifications, you can use free trial version of bpm’online deployed on cloud. For more information on bpm'online deployment on cloud, please see the "**Deploying the bpm'online cloud application**" article.

To use specific development tools (for example, Visual Studio), you will need to deploy application on-site. Please refer to “**How to deploy bpm'online on-site**” for any details.

To add complex custom functionality that requires work of a group of developers, you will need to use specifically configured development environment. For more information refer to the “**Organizing a development environment**” article.

ATTENTION

The development in the bpm’online production environment is forbidden. In most cases the development is connected with errors and their tracking, debugging and compiling the application, etc. Usually this has a negative impact on bpm’online performance or can make the work of other users difficult or impossible.

3. Configure SVN storage (optional)

Version control system (SVN) is an optional component for development. If an active development of the application is expected, the version control system will facilitate the management of the development process.

More information about the version control system can b e found in the “**Create repository in SVN server**" and "**Working with SVN in the file system**" articles.

4. Create a custom package for developing new functionality

The bpm'online functionality is implemented in configuration elements – schemas. A set of schemas that implement some functionality is combined in a package. More information about purpose and structure of bpm’online packages can be found in the “**Package structure and contents**”, “**Package dependencies. Basic application**

packages” and “**Package [Custom]**” articles.

Create a new custom package to develop new functionality. Please refer to the “**Creating and installing a package for development**” article for any details.

ATTENTION

To use the version control system, the package must be connected to the SVN storage at the time of creation. Working with packages in SVN described in the "**Creating and installing a package for development**", "**Committing a package to repository**", "**Installing packages from repository**" and "**Updating package from repository**” articles.

For more information about creating a package in the development in file system mode, refer to the “**Creating a package in the file system development mode**” article.

5. Create schemas that implement the functionality

To implement the functionality, it is required to create various types of schemas in user packages. Creating of schemas is described in the “**Creating a custom client module schema**”,”**Creating the entity schema**” and “**Creating the [Source code] schema**” articles.

Templates of development of new functionality are described in the “**Bpm’online development cases**” article.

6. Transfer modifications to test and production environments

After the development is completed, the modifications must be transferred to the pre-production environment for the testing. If the testing was successful, transfer the modifications to the production environment. For more information refer to the “**Transferring changes between the working environments**” article.

See also

- **Development process organization**
- **Organizing a development environment**
- **Recommended development sequence**
- **Development rules**
- **How to deploy bpm'online on-site**
- **Deploying the bpm'online cloud application**
- **Create repository in SVN server**
- **Working with packages**
- **Transferring changes between the working environments**
- **Creating a custom client module schema**
- **Creating the entity schema**
- **Creating the [Source code] schema**

Development process organization

Overview

When adding a new complex custom functionality to bpm'online, be sure to follow the proper development process. We recommend deploying three separate environments: development, testing and operational.

NOTES

For more information on organization of the project development process, please refer to the "Project Life Cycle Methodology" documentation.

The Development Environment is a separate application (or a number of applications) where a new functionality is developed. These applications must be deployed on local computers (on-site), which gives the ability to export schemas to the file system and create a new program code using different IDE. SVN version control is also highly recommended. Use a separate application and database for developing new functions. For more information about the development Environment, please refer to the "**Organizing a development environment**" article.

The Pre-Production environment can be a separate application where the new functions are installed and tested. Usually, the testing is done by a system analyst from the development team or the customer who ordered the development of the new functionality. If needed, the application can be deployed in the cloud or on-site.

The production environment is a separate bpm'online application, in which all current user business processes are executed. If needed, the application can be deployed in the cloud or on-site mode, on customer servers.

For more information about deployment options, please refer to the "**How to deploy bpm'online on-site**" and "**Deploying the bpm'online cloud application**".

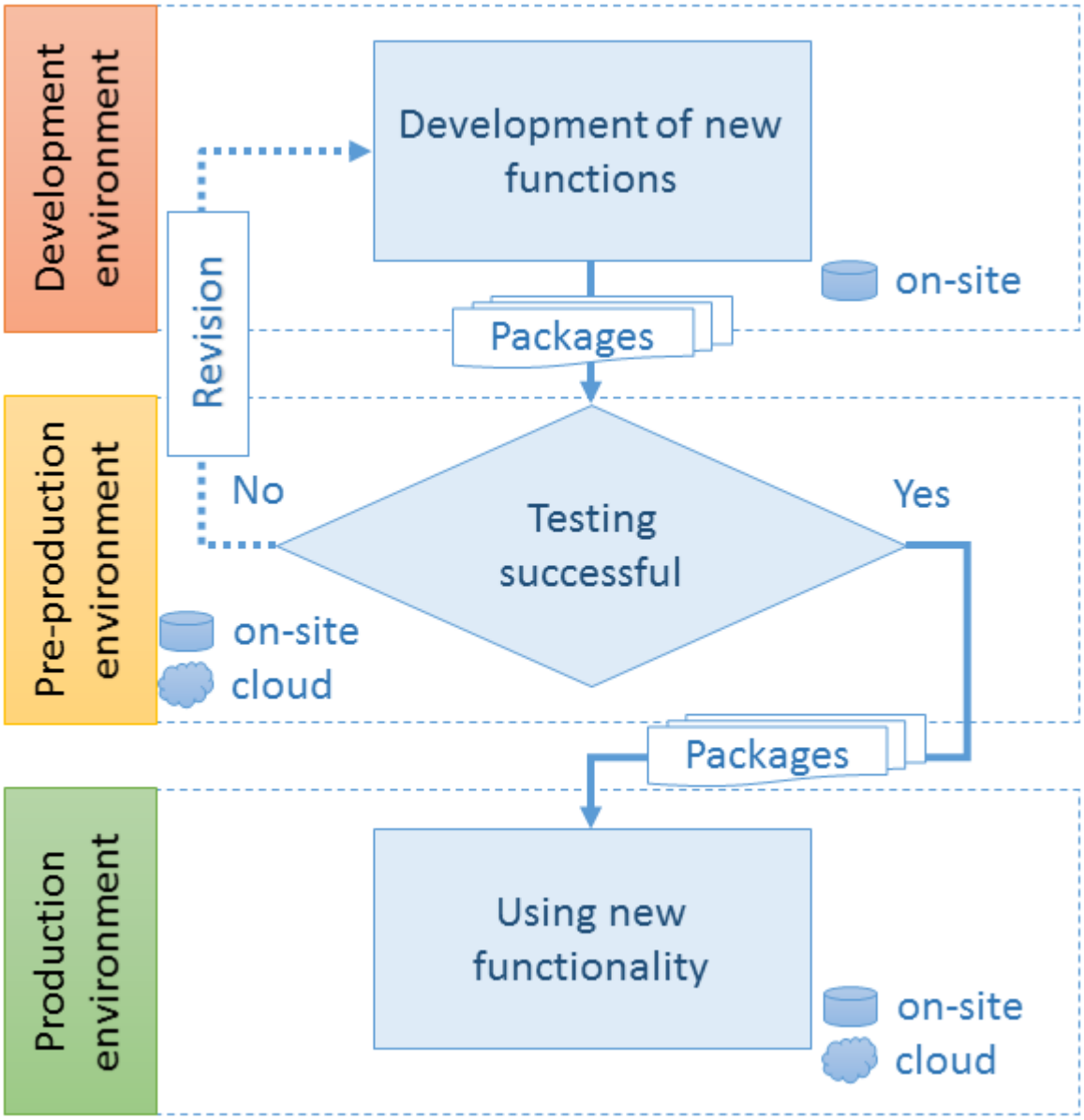
ATTENTION

The production database must never be used for development or pre-production testing. Development activities cannot be performed in the production environment.

The general development process is shown in Fig. 1.

1. All development activities are performed in the development environment.
2. After development is complete, the developers prepare packages with the new functions and install them in the pre-production environment.
3. The new functions are then tested in a pre-production application.
4. Any errors found during testing are corrected in the development environment (stage 1). When the testing is complete and all errors are corrected, the packages with the new functionality are installed in the production environment.

Fig. 1. General workflow of the development process



It is recommended to use zip-archives to transfer packages between environments. Zip-archives can be created **in the [Configuration] section** or **by WorkspaceConsole utility**. For more information about transferring changes between applications, please refer to the "**Transferring changes between the working environments**" article.

Organizing a development environment

Overview

The development environment is a separate bpm'online application (or a number of applications) used exclusively to develop new functions. Pre-production and production environments are used for testing the implementation of developed functionalities. For more information on pre-production and production environments, see "**Development process organization**" article.

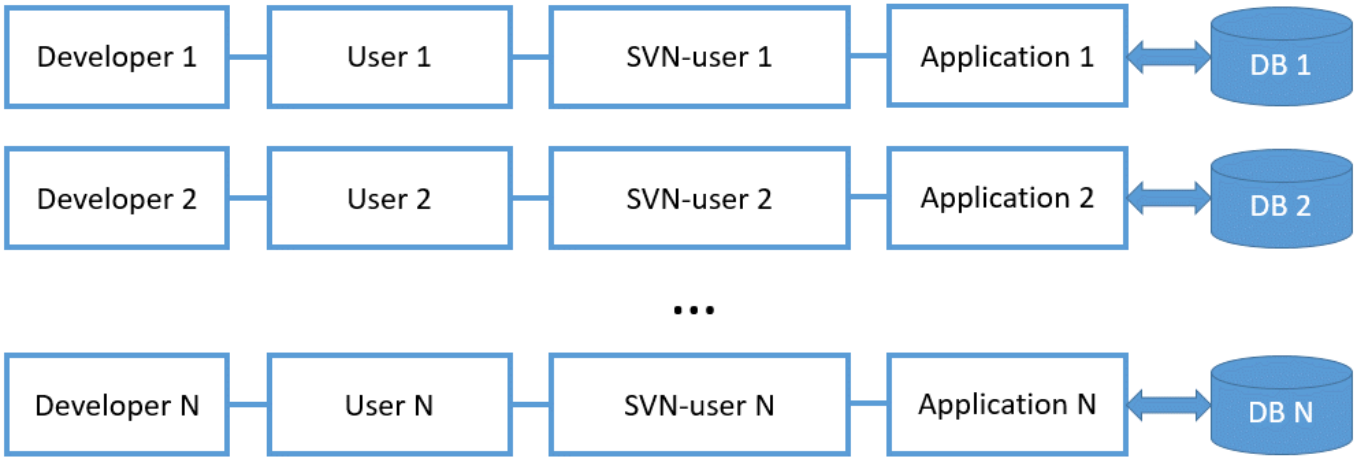
> ⚠ ATTENTION
>
> The production environment database must never be used for development. Development-related activities in the production environment are strictly forbidden.

The applications can be deployed either locally (*on-site*), or on bpm'online servers (*cloud*), with or without the use of SVN repository. You can also use **file system development mode**. For more information on bpm'online deployment options, please see "**How to deploy bpm'online on-site**" and "**Deploying the bpm'online cloud application**" articles. For more information on working with the version control repository, please see the "**Create repository in SVN server**" and "**Working with SVN in the file system**" articles.

Development in on-site application

Having separate development environment applications for each developer requires on-site deployment (Fig. 2). Because this option is aimed at maximum development productivity, the use of SVN, as well as development in the file system are required. For more information on development in the file system, please see the "**Development in the file system**" section.

Fig. 2. Organization of development environment in several configurations of one application



Advantages:

1. Fast and convenient development process.
2. Independent development environments. Since development is made in a separate application, other applications cannot be affected.
3. Using the version control system for saving and transferring changes.
4. Possibility of using IDEs and setting up continuous integration processes.

Disadvantages:

1. Cloud deployment option is unavailable.

Recommendations:

1. Development in separate applications is recommended for supporting active development or making changes to base functionality.
2. Recommended for both small and large developer teams.

Recommended development sequence

Introduction

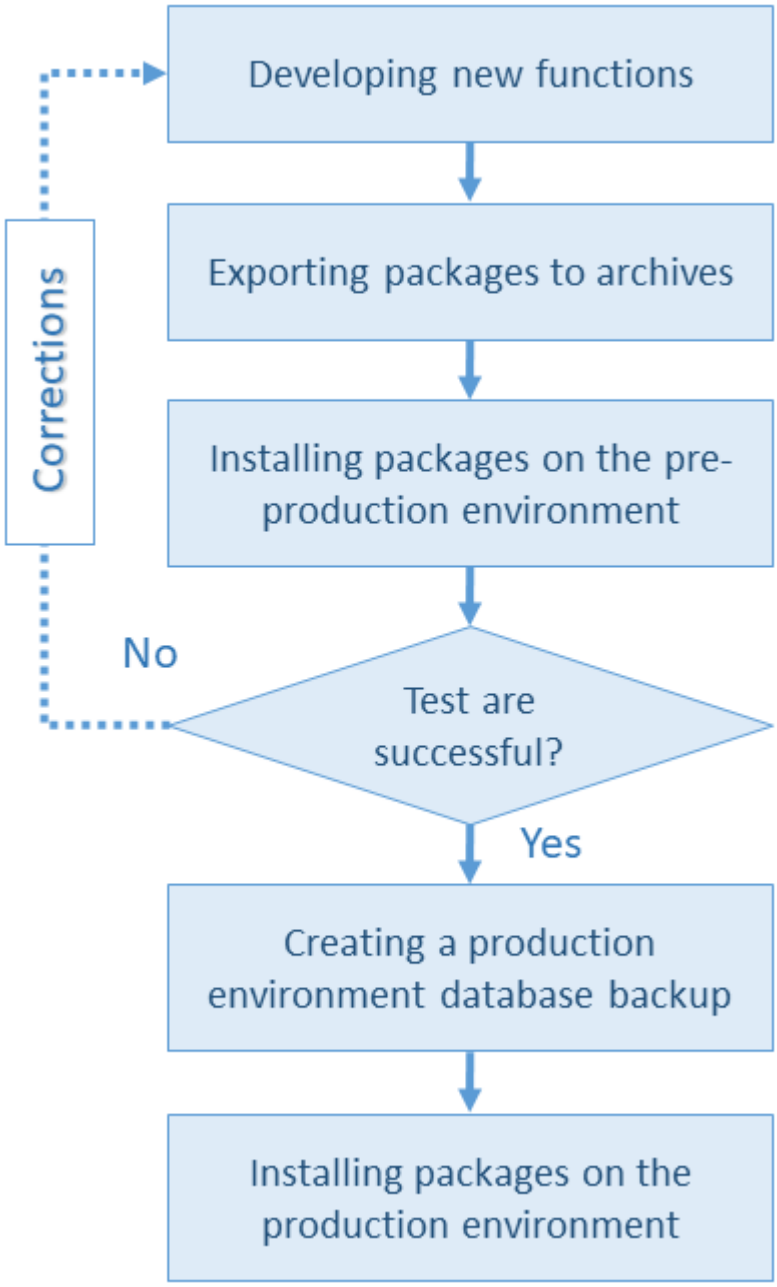
Development of complex functionality requires proper **organization of the development processes**. There are three general options for development environment deployment:

- All instances are deployed on-site.
- Development environment is deployed on-site, pre-production and production environments are deployed on the cloud.
- All instances are deployed on the cloud.

Development sequence

Recommended algorithm of the development process is given on Fig. 1.

Fig. 1. General sequence of development



1. Development of new functions

It is recommended to develop **in a separate application** with a separate database for each developer. Use **subversion control system** (Subversion, Git, etc.) to transfer changes between different development environments.

ATTENTION

Using SVN is not recommended for transferring changes to the production environment, as this method does not assume creating database backups. Transferring changes with SVN can only be used in the development environment.

2. Exporting packages to archives

Two options for uploading packages into archives:

- From the [Configuration] section (see the "**Exporting packages from the application interface**"

article).
- Via the WorkspaceConsole utility (see the “**Transferring changes using WorkspaceConsole**” article).

3. Installing the packages on the pre-production environment

There are two options for installing packages to application:

1. From the custom application interface (see the “**Installing marketplace applications from a zip archive**” article). You can use this option when placing a pre-production environment in the cloud.
2. Via the WorkspaceConsole utility (see the “**Transferring changes using WorkspaceConsole**” article). You can use this to set up the continuous integration processes when placing pre-production environment on-site.

> ATTENTION
>
> To migrate changes to an application deployed in the cloud, it is recommended that you use the options of the bpm'online user interface. Using WorkspaceConsole is not possible because the user does not have direct access to the cloud application database.

In case errors are found during the testing stage, the new functions are revised and the errors are corrected in the development environment. After all errors have been fixed, repeat steps 1—3.

4. Creating production database backup

Back the production database up before installing the packages with the new functions. This is a required step, since there is always a chance that the new functions developed by third-party developers may disrupt the operation of the application.

> ATTENTION
>
> Contact bpm’online support to create the backup of the database deployed in cloud. When deploying an on-site application, the database backup is created by the client on its own.

5. Installing the packages on the production environment

Options for uploading packages to the production environment are common to the options for pre-production environment (Step. 3).

Development rules

Introduction

During the creation of new functionality, bpm’online developers and partners have compiled a set of rules and recommendations. Development can be carried out by several employees simultaneously in personal **development environments**. Any employee with the appropriate skills can act as a developer.

Minimal required developer skills

Over 6 months of C#, JavaScript, and T-SQL (PL-SQL) programming experience.

Recommended developer skills

Over a year of C#, JavaScript and T-SQL (PL-SQL) programming experience. Expertise in WCF and OData

technologies, as well as Sencha Ext.JS framework and RequireJS library.

Development rules and recommendations

Using a development environment

New functionality must only be developed in the **development environment**. It is forbidden to develop new functionality in a **pre-production or production environment**.

Developing in a configuration

The development should be carried out only in the development database in the default workspace (the [Default] configuration, sequence number 0). Developing in custom configurations is not recommended, even in case of minor changes that will not be delivered to other users.

Developing in a custom package

The development of new custom bpm’online functionality must be carried out in a **separate custom package**. Do not use the **[Custom] package**. All the necessary data (for example, lookup contents content), SQL-scripts and dependencies must be attached to the package.

Using the SVN

If the development is carried out by several developers, you must use the **revision control system** (SVN). When the development is carried by a single developer, it is recommended to use SVN.

Identification of the solution provider

To prevent errors associated with same package element names and their properties created by different vendors, use the following system settings:

- [Publisher] (*Maintainer*) - contains the package vendor name. The default value is set to “Customer”.
- [Object name prefix] (*SchemaNamePrefix*) contains a prefix installed in the custom schema names and names of custom columns in the objects that are inheritors to the system objects. The default value is set to “Usr”.

Using the extending and replacing modules and schemas

If you want to create a extending **view model schema** (for example, schema of the section record edit page), you need to add only the differences from the parent schema. Most often, those are the new attributes, methods, events, and the *diff* array of modifications. You must only add only new view models that are not in the parent schema to the *diff* array of modifications.

If you need to create a replacement **module**, you need to copy the module’s source code, which is replaced, and to add a new functionality. Bpm’online modules cannot be expanded.

> ⚠ NOTES
>
> The [Configuration] section contains the same [Add] — [Replacing Client Module] command for creating extending and replacing schemas. That is why extending schemas are also called "replacing".

Using localizable strings

It is forbidden to use string literals in the schema source code. All string values that are displayed in the user interface must be presented as localizable strings. This is important for localization of solutions.

Data backup

Before moving changes to the production environment, it is imperative to create a backup copy of the database. The database must be backed up before installing updates and solution from third-party developers.

How to deploy bpm'online on-site

Introduction

The on-site (on-premises) deployment implies hosting the system on the servers or personal computers of customers.

To deploy the bpm’online application on-site, the server-side and the client-side must meet certain technical requirements. These requirements are described in the “Server-side system requirements” and “Client-side system requirements” articles. Complying with certain technical requirements will ensure high system performance.

The guide, which covers all stages of bpm’online on-site setup and deployment, including setup instructions for bpm’online, additional Windows components, database deployment, modifying configuration files, setting up DB server connection parameters as well as website setup in IIS, is available in the "Deploying bpm’online application on-site" article in the User Guide.

Deploying the bpm'online cloud application

Introduction

The standard procedure for deploying the bpm”online cloud application is as follows:

1. Use the free trial registration page at bpmonline.com to create your trial bpm'online site. During the trial, you can familiarize yourself with the main features of the application. After the trial period is complete, the demo version can be transferred to the primary bpmonline site.
2. Contact a bpm'online sales manager to deploy a new application on the cloud or transfer an existing application to the bpm'online cloud service. Bpm'online staff will perform the transfer.

When creating bpm'online cloud applications, certain limitations apply. Compliance with these requirements is critical for successful deployment.

Primary limitations

The use of SQL Agent is restricted

Tasks (Jobs) and other actions performed by SQL Agent cannot be created. The bpm'online task planner must be used instead.

The use of DB Mail is restricted

Email notifications must be sent via bpm'online platform features.

The use of Extended Stored Procedure is restricted

All logic must be implemented either through standard stored procedures on T-SQL, or through the use of the application server features.

The use of DBMS user names is restricted

Database users are not created within the DBMS on the bpm'online site. Domain users and domain authentication are used instead.

Modifications to Web.config are restricted

All required parameters must be stored as bpm'online system settings.

Binding to server and DBMS IP addresses is restricted

Server IP addresses may be changed. Therefore, any binding to any specific IP address will become invalid after such change. Always use domain names.

Installing additional software is restricted

No additional software can be installed on bpm'online servers.

Working with file system is restricted

Working with the application and DBMS server file system is restricted by OS access rights. Access to files is available through FTP and HTTP protocols.

Third-party applications cannot be run on server

Running third-party applications is restricted by OS access rights. All business logic must be implemented as part of the bpm'online application.

Database must be deployed on SQL Server 2016

To ensure compatibility with bpmonline site cloud infrastructure, the application database provided by customers must be created on SQL Server 2016.

The application must support both HTTP and HTTPS protocols

The use of logic that supports only one protocol is restricted. Instead, current application protocol must be defined.

The application must work with access rights of a regular user

The use of functions that require administrator access rights is restricted.

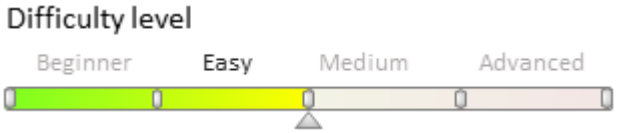
The application must work as a user without a profile

On the bpm'online site, the users are created without profiles or the ability to actually log in to OS.

Additional recommendations for partners

- Set the partner name as the "Maintainer" system setting.
- In the UsrPrefix system setting, specify a partner-specific prefix. For example, if the partner name is "FineSolution", the UsrPrefix could be "FS".
- The partner solution must not use replacing modules. Only schemas may be replaced.
- Server logic must be concentrated in C# classes and called where needed.
- The public API for server classes and client schemas must be covered by unit-tests.
- All required data, scripts, and libraries must be bound to packages.
- Development must be performed with the use of SVN and all packages must be committed to the repository.

Create repository in SVN server



Introduction

The purpose of version control system in bmp'online:

- Transfer of changes between workspaces.
- Storage of versions of configuration schemas.

Version control system is an optional component. However, if you intend to customize the application, the version control system is required.

bmp'online supports operation with Subversion control system (SVN) of version 1.7 and higher.

For more details on use of SVN see documentation.

Principles of operation with repositories of version control system

ATTENTION

The principles listed below are applicable when working with SVN repositories via the **bpmonline built-in development tools**. The principles are not applicable when the file system design mode is turned on (see "**Working with SVN in the file system**").

- You can add newly created packages to any repository in the list.
- You can commit an already installed package only to the repository that was specified when the package was created.
- You can install any number of packages from the list of available repositories in the configuration.

Register a repository and add it to the list of repositories in order to use it.

SVN setup

To set up integration with SVN:

1. Install SVN server

You can install SVN on the application server, DBMS server or on a separate dedicated server.

Use one of the publicly available SVN installers for Windows:

- VisualSVN
- CollabNet

You can download the last version of binary files of the SVN server for your operating system here.

SVN server can function independently or through Apache web-server (it is installed automatically be means of the VisualSVN and CollabNet utilities). In the first case, repositories are accessed through svn:// protocol. In the second case repositories are accessed through the http(s):// protocol.

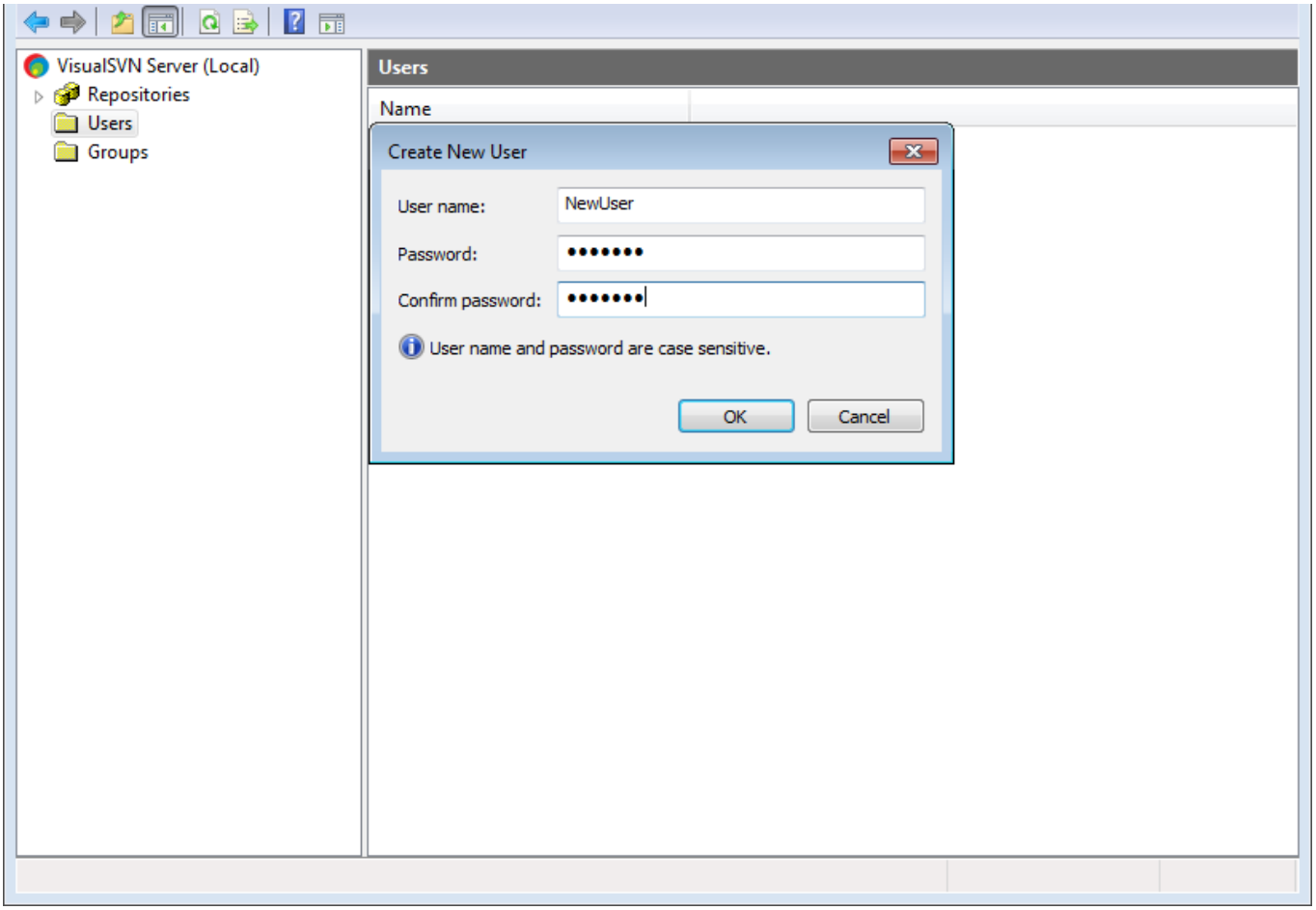
We recommend using the http(s):// protocol for integration with bpm’online.

2. Create a user on the SVN server

You can create an SVN server user via the standard tools that are supplied with the utility that was used for installation of the ASVN server, for example, VisualSVN (figure 1). Login and password are required for working

with the bpm'online repository.

Fig. 1. — Creation of a new user in SVN server (VisualSVN utility).



3. Create repository on the SVN server

The repository is created by standard tools of utility that were used for the SVN sever setup (i.e., VisualSVN and CollabNET).

> NOTE
>
> bpm'online supports simultaneous operation of several repositories that can be located on different SVN servers.

4. Install SVN client

You can additionally install an SVN client in the developer workplace, for example, TortoiseSVN.

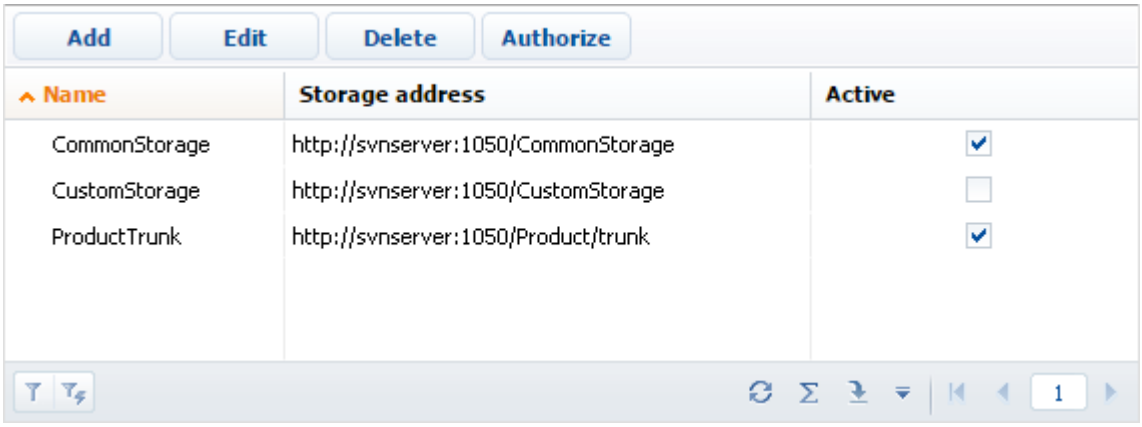
> NOTE
>
> We recommend using TortoiseSVN client version 1.8 and up.

The installation of an SVN client is optional since it does not affect bpm'online operation. Using an SVN client is convenient for viewing the local working copy, history, revert operations, review, etc.

List of repositories

To open the list of available repositories (figure 2), select the [Open repository list] action on the [Actions] tab of the [Configuration] section interface.

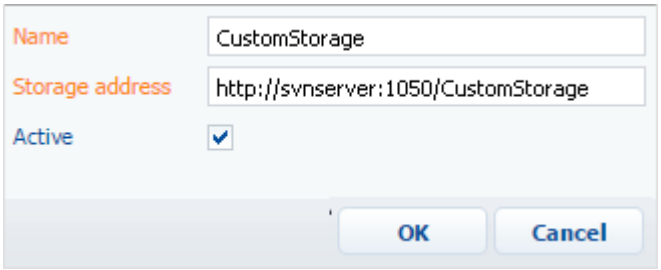
Fig. 2. - Window with repository list of version control system



Adding a new repository

In order to adda new repository, select [Add] on the list tool bar. As a result, a card for the new repository opens (Figure 3).

Fig. 3. — New repository card



[Name] — repository name.

[Storage address] — network address of existing SVN repository. Repository addressing is supported by both the HTTP protocol (standard network protocol) and SVN protocol (own network protocol of the Subversion system).

[Active] — checkbox that determines whether to use the repository in the system operation. Each new repository is marked as active by default.

> ⚠ NOTE
>
> You can work with active repositories only. Moreover, all repositories, from which the packages are updated, must be active. These include the repository from which the initial package is updated and the repositories from which all packages-dependencies of the initial package are updated.

After registration of a new repository it can be used for creating custom packages and installing created packages in the workspace.

Working with packages

Introduction

Any bpm’online product is a specific set of packages that are used to modify the configuration.

A bpm'online package is a collection of configuration elements that implements particular block of functionality.

Learn more about configuration elements in a package and their structure in the "**Package structure and**

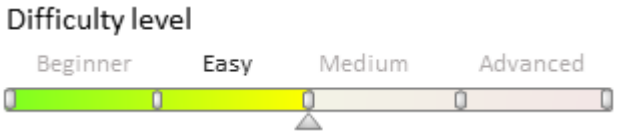
contents" article.

Package dependencies, package hierarchy and main system packages are described in the "**Package dependencies. Basic application packages**".

The "**Package [Custom]**" describes features of the package intended for custom application configuration with the help of system wizards.When a customer develops new functionality and, therefore, creates new packages they have to use a revision control system (SVN). Working with packages described in the:

- **Creating and installing a package for development**
- **Committing a package to repository**
- **Installing packages from repository**
- **Updating package from repository**
- **Installing marketplace applications from a zip archive**
- **Exporting packages from the application interface**
- **Creating a package in the file system development mode**
- **Binding data to packages**

Package structure and contents



General information about packages

A bpm'online package is a collection of configuration elements (schemas, data, scripts, additional libraries) that implement a particular block of functionality. In the file system, packages are directories with various subdirectories and files.

Any bpm’online product is a finite set of packages. To extend or change product functionality, you need to install the package in which all necessary changes are already implemented.

The bpm'online packages can be divided into two types:

- Pre-installed packages. Supplied with the system and are installed by default. These include packages with basic functionality (e.g., *Base*, *NUI*) and packages developed by third-party developers. These packages are installed from zip archives as marketplace appblication or by using the WorkspaceConsole utility.
- Custom packages are created by the users of the system. These packages can be attached to the SVN repository.

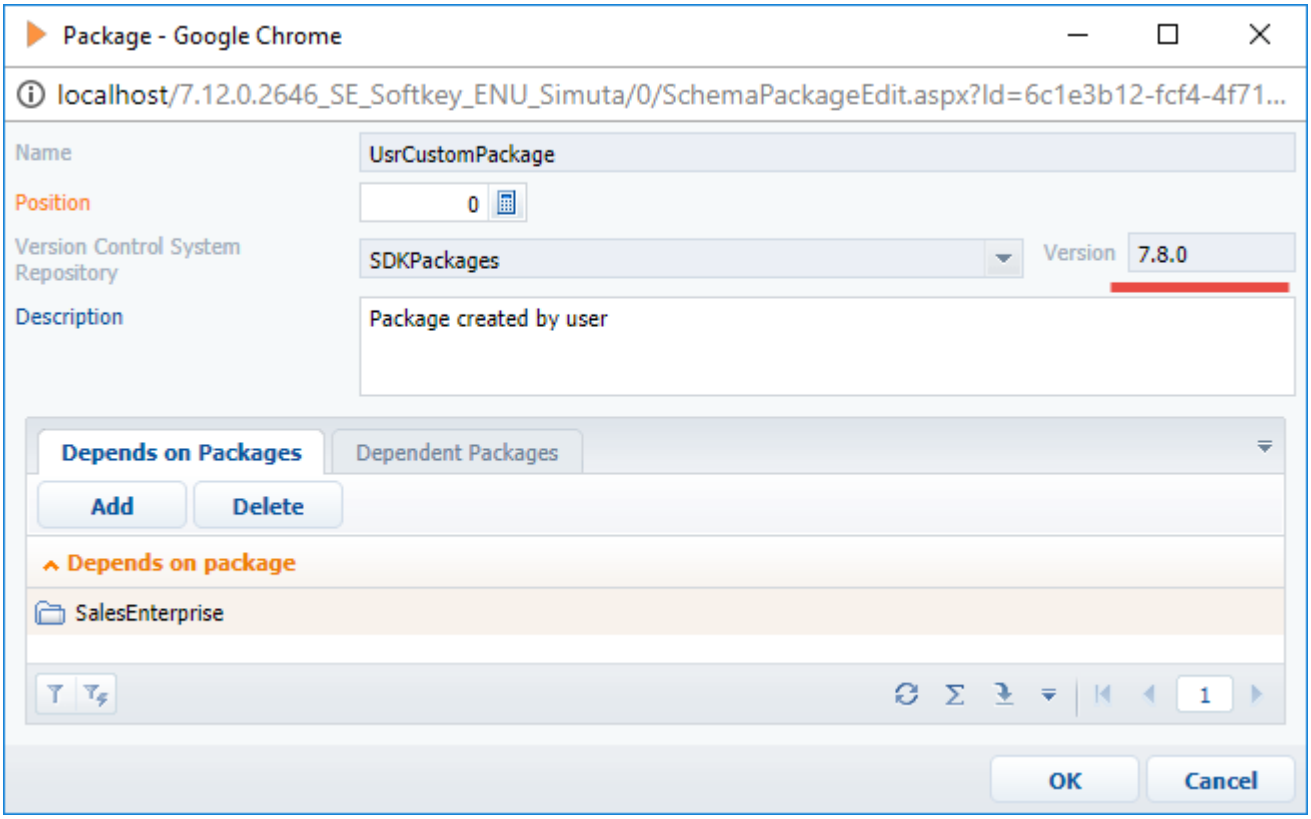
Configuration elements of the pre-installed packages cannot be modified. You can develop additional functionality or modify the existing functionality only via custom packages

Package version

One of the characteristics of a package is its version. The version is specified in the corresponding package mini-page field (Fig. 2). The package version can contain digits, Latin characters, "." and "_” symbols. The package version must begin with a numeric or alphabetic character.

> ⚠ ATTENTION
>
> The package versioning mechanism is deprecated and is not supported, starting with bpm'online 7.9. Therefore, all pre-installed packages have a version no higher than 7.8.

Fig. 2. Package version in the package mini-page



In addition, the package version is stored in its metadata of the *PackageVersion* object specified in the *descriptor.json* file. The *descriptor.json* file is created for each package version. Example of *descriptor.json*:

```
{
  "Descriptor": {
    "UId": "8bc92579-92ee-4ff2-8d44-1ca61542aa1b",
    "PackageVersion": "7.8.0",
    "Name": "UsrCustomPackage",
    "ModifiedOnUtc": "\/Date(1522412432000)\/",
    "Maintainer": "Customer",
    "Description": "Package created by user",
    "DependsOn": [
      {
        "UId": "e14dcfb1-e53c-4439-a876-af7f97083ed9",
        "PackageVersion": "7.8.0",
        "Name": "SalesEnterprise"
      }
    ]
  }
}
```

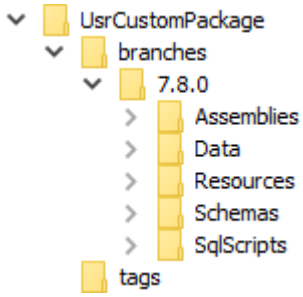
All elements of the package are of the same version as the package itself.

The application is updated by installing packages with the functionality of newer package versions.

Package structure

When you commit a package to the SVN, a folder with the package name is created, and the *branches* and *tags* directories are created inside it (Fig. 3).

Fig. 3. Package structure in the SVN



The *branches* directory contains all versions of this package. Each version is stored in a separate subfolder whose name matches the package version number in the system, for example, *7.8.0*.

> ATTENTION
>
> The structure that takes into account the package versions remained for compatibility with bpm'online versions below 7.9.

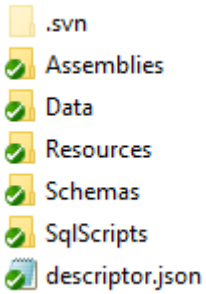
The *tags* directory stores tags. The tags in the version control system represent a "snapshot" of the project at a certain point in time, a static copy of the files required for saving some critical stage of development.

Working copies of the packages are stored locally in the file system. The path to the package repository is specified in the *ConnectionStrings.config* configuration file in the *connectionString* attribute of the *defPackagesWorkingCopyPath* element:

```
<add name="defPackagesWorkingCopyPath"
connectionString="%TEMP%\%APPLICATION%\%WORKSPACE%\TerrasoftPackages" />
```

The directory containing the package name is created in this path. Its inner structure is shown in Fig. 4.

Fig. 4. The package directory structure in the file system



The package schemas are contained in the *Schemas* directory. External assemblies attached to the package, data and SQL scripts are contained in the *Assemblies*, *Data* and *SqlScripts* directories. All package text resources, translated into different languages, are stored in a separate *Resources* directory.

> ATTENTION
>
> Starting with version 7.11.3 the *Files* catalog has been added to the package structure. The catalog contains the file content (see "**Using file content in packages**")

The *descriptor.json* file stores the package metadata in JSON format — ID, name, version, dependencies, etc.

Package dependencies. Basic application packages

Difficulty level

Beginner Easy Medium Advanced

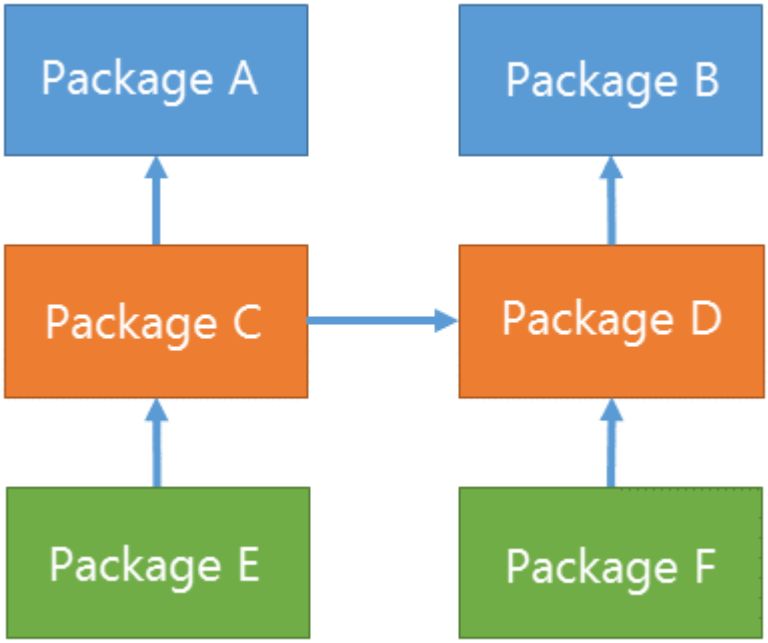
Introduction

Bpm’online application development follows the basic principles of software design, in particular the “don’t repeat yourself” (DRY) principle. In the bpm’online architecture, the concept of packages was built around this principle and is implemented using dependencies between packages. Each package contains certain application functionality, which should not be duplicated in other packages. If a package requires functions that are part of a different package, you will need to set up dependencies between the packages.

Dependencies and package hierarchy

Packages can have multiple dependencies. For example, package C (Fig. 1) depends on packages A and D. Thus, all the functionality of the packages A and D is available in the package C.

Fig. 1. Dependencies and package hierarchy

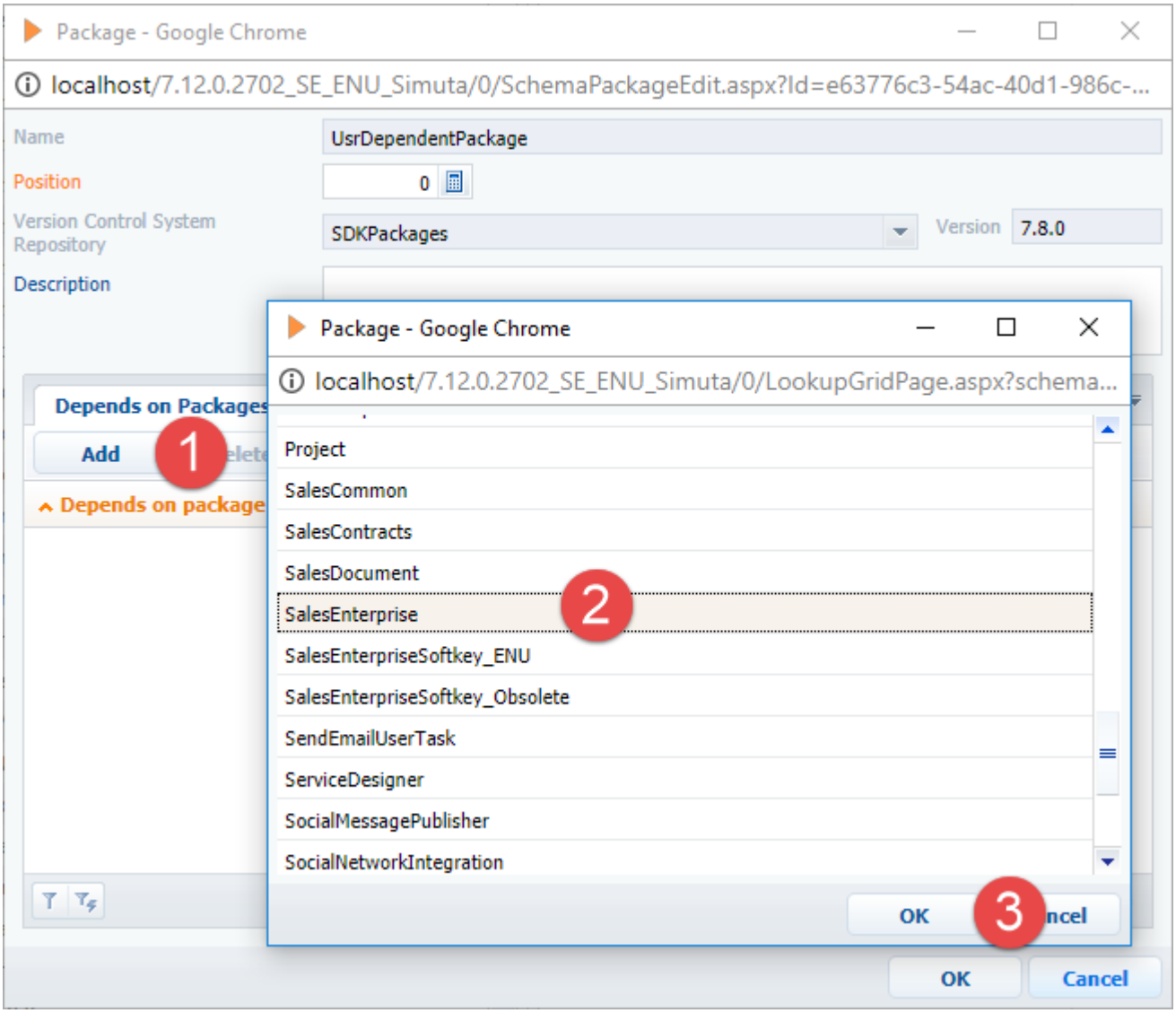


Package dependencies form a hierarchical chain. This means that if you add a package to the dependency of another package, the dependent package will contain all functionality of the added package as well as functionality of all packages that the added package depends on. The closest analogy of the package hierarchy is the inheritance hierarchy of classes in object-oriented programming. For example, package E (Fig. 1) contains not only package C functionality on which it depends, but also the functionality of packages A, B and D. In addition; package F contains the functionality of packages B and D.

How to add package dependencies

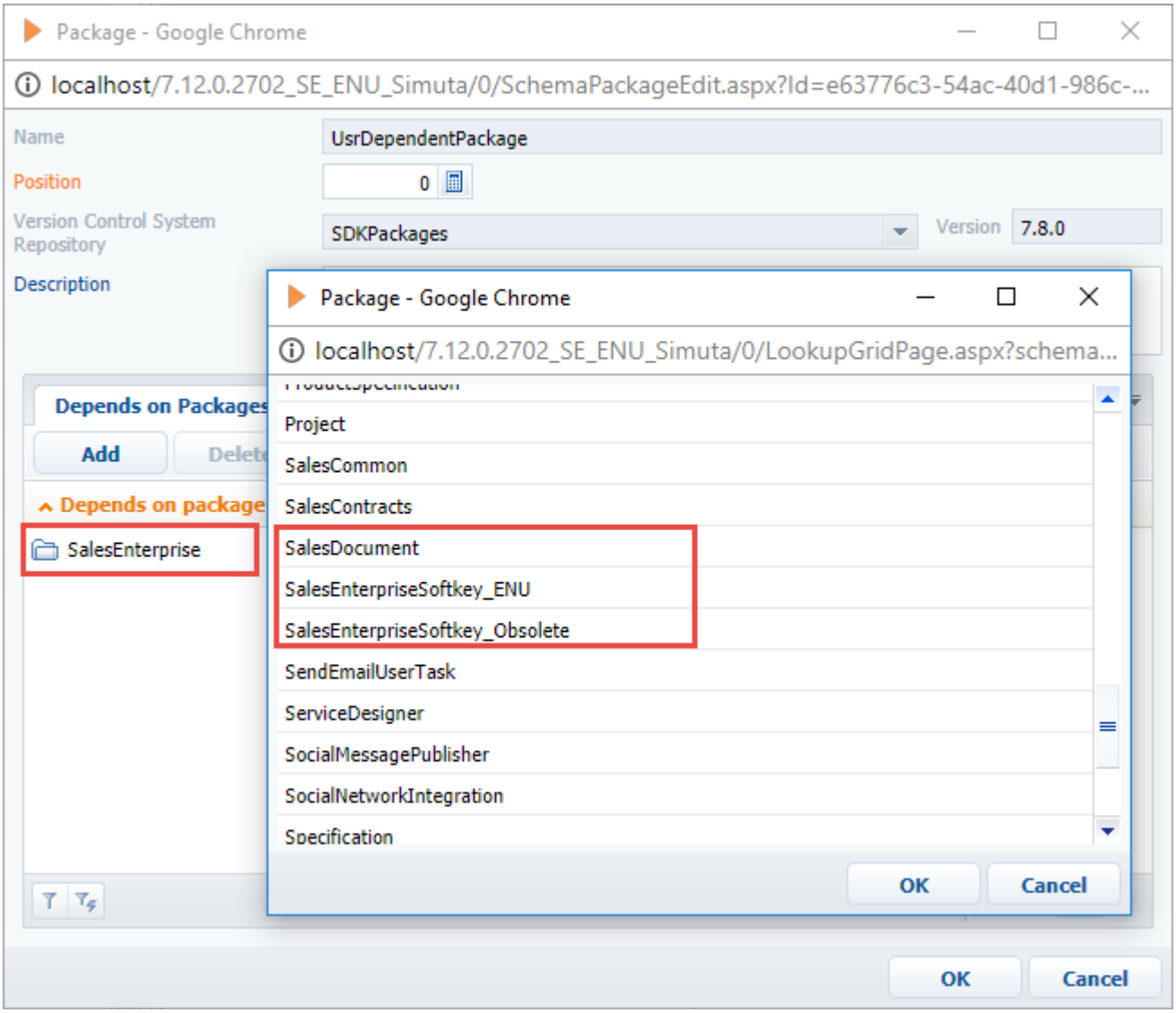
Dependencies can only be added to a custom package, and only after it has been created. To add dependencies, click the [Add] (1) button on the [Depends on packages] (Fig. 2) detail. In the opened dialog of the package lookup, select the required package (2) and click the [OK] button (3).

Fig. 2. Adding dependency to a custom package



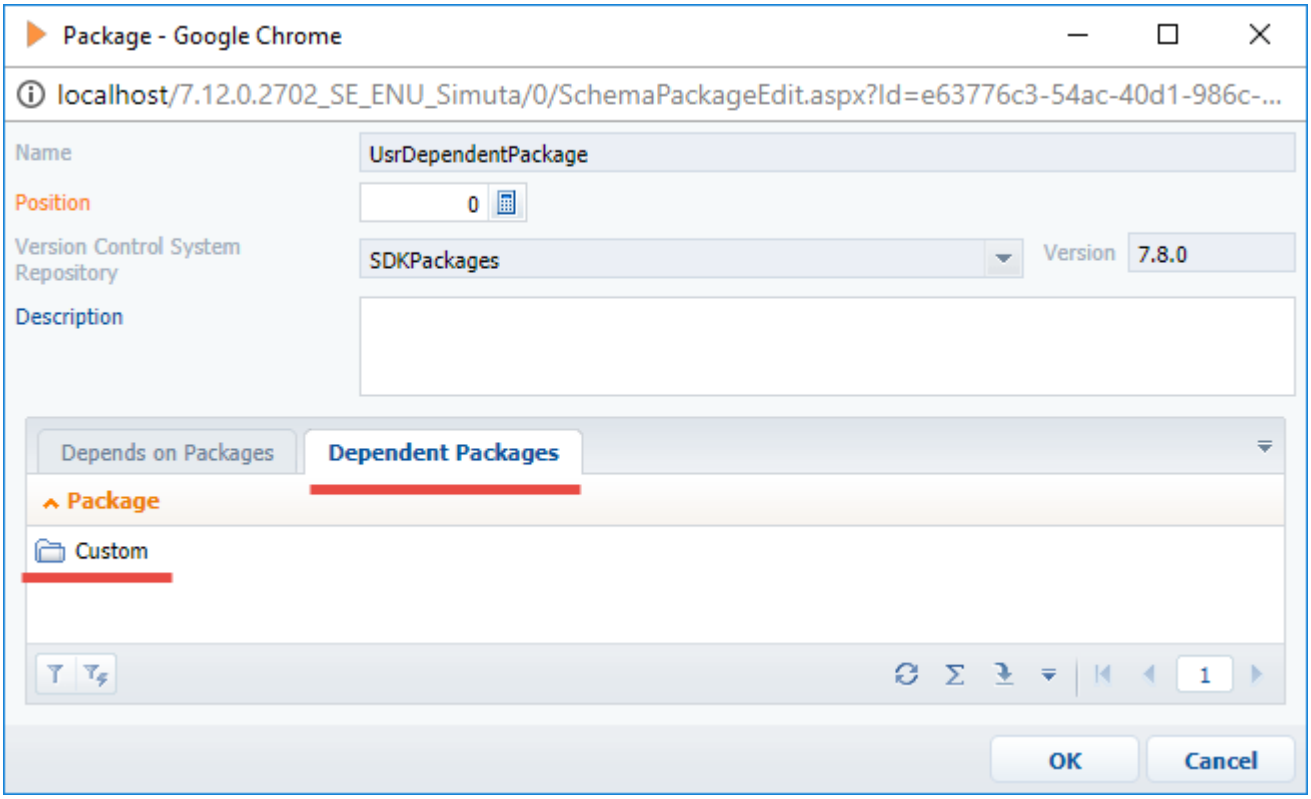
The selected package will be displayed in the list of dependencies of the current package. The packages that have been added to the dependencies are not displayed in the package lookup (Fig. 3).

Fig. 3. Added package dependency



After creating a new package, it will be automatically added to the dependencies of the pre-installed “Custom” package (fig. 4).

Fig. 4. "Dependent Packages" tab



The list of dependencies in the metadata

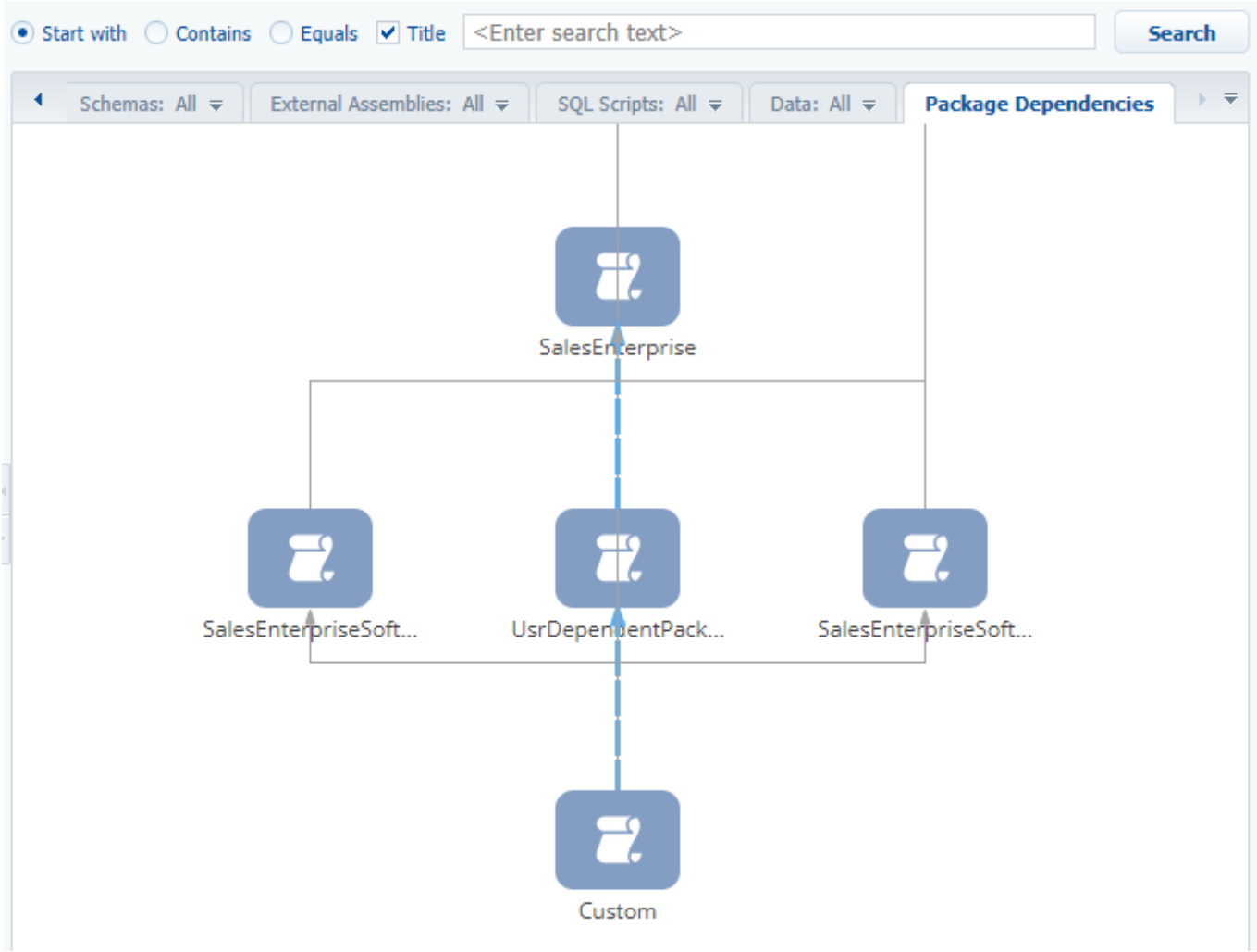
The list of package dependencies is stored in the package metadata in the *DependsOn* property of the object specified in the *descriptor.json* file. The *DependsOn* property is an array of objects that contain the package name, version and unique identifier by which the package can be identified in the application database. A *descriptor.json* file is created for each package version. Example of *descriptor.json*:

```
{
  "Descriptor": {
    "UId": "51b3ed42-678c-4da3-bd16-8596b95c0546",
    "PackageVersion": "7.8.0",
    "Name": "UsrDependentPackage",
    "ModifiedOnUtc": "\/Date(1522653150000)\/",
    "Maintainer": "Customer",
    "DependsOn": [
      {
        "UId": "e14dcfb1-e53c-4439-a876-af7f97083ed9",
        "PackageVersion": "7.8.0",
        "Name": "SalesEnterprise"
      }
    ]
  }
}
```

Application package hierarchy

Use the package dependency diagram to explore the hierarchy and application package dependencies. This chart is located on the [Package dependencies] tab of the [Configuration] section (Fig. 5).

Fig. 4. A fragment of package dependency hierarchy



If you click the node element of the package name diagram, the animated arrows will display package dependencies. For example, in the SalesEnterpise product, the [UsrDependentPackage] depends only on the [SalesEnterpise] package and all its dependencies (Fig. 5). The [Custom] package also depends on the [SalesEnterprise] package.

Primary packages

The application’s primary packages include the packages that are always available in all products. A brief list of such packages is shown in table 1.

Table 1. Basic application packages

Package name	Contents
Base	Base schemas of the primary objects, sections and object schemas, pages or processes connected to them.
Platform	Modules and pages of the section wizard, content designer, dashboard designer, etc.
Managers	Client modules of the schema managers.
NUI	Functionality connected to system user interface.
UIv2	Functionality connected to system user interface.
DesignerTools	Schemas of designers and their elements.
ProcessDesigner	Process designer schemas.

Package [Custom]

Introduction

There are two types of bpm’online packages:

- Pre-installed packages are supplied with the system and are installed by default.
- Custom packages are created by the system users. Packages can be bound to the SVN repository.

Configuration elements from the pre-installed packages are not available for editing. Any development can be done in the custom packages only.

The Section Wizard and Detail Wizard create various schemas that must be saved in a custom package. A clean application install does not include editable packages. The pre-installed packages cannot be modified.

By default, any custom changes are saved in a pre-installed package named “Custom”. This package enables adding schemas manually and using wizards.

Specifics of the “Custom” package

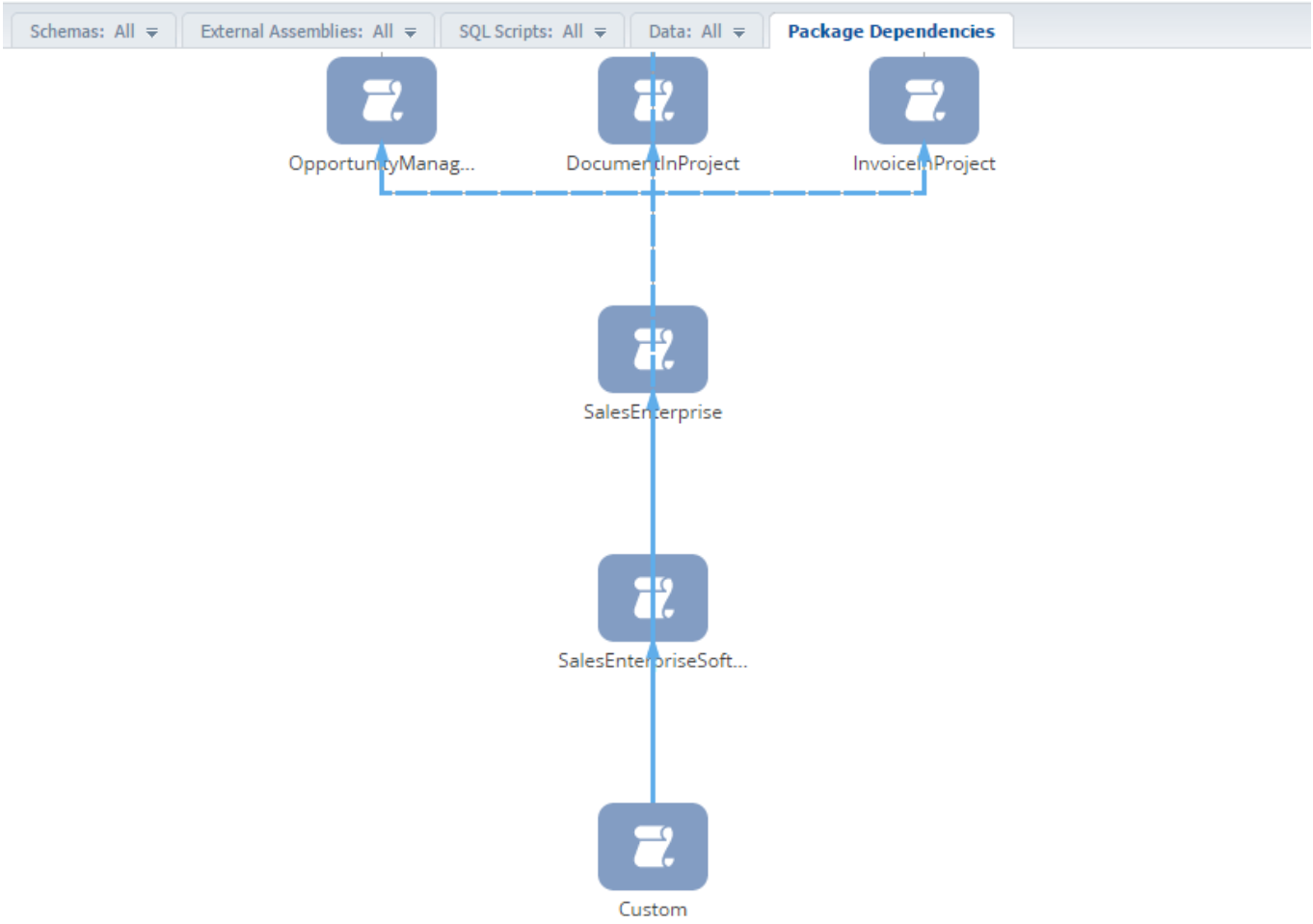
As a pre-installed package, “Custom” cannot be added to the SVN subversion control repository. The schemas can be transferred from the “Custom” package only by using the **export/import function**.

Unlike other pre-installed packages, the “Custom” package cannot be exported to the file system via the **WorkspaceConsole** utility.

The “Custom” package depends on all pre-installed packages. If a new custom package is created or installed, a dependency from this package is automatically added to the “Custom” package. The “Custom” package must always be the last in the package hierarchy (depend on all other packages). For more information on the package dependencies and hierarchy, please see the **Package dependencies. Basic application packages** article.

Custom packages cannot depend on the [Custom] package.

Fig. 1. The “Custom” package in a package hierarchy



NOTE

A custom package can technically be made last in the package hierarchy using the [Custom Package Id] (CustomPackageUId) system setting. You can add pre-installed packages (including the “Custom” package) to its dependencies only of the development is done without using SVN.

It is not recommended to replace the “Custom” package with other packages!

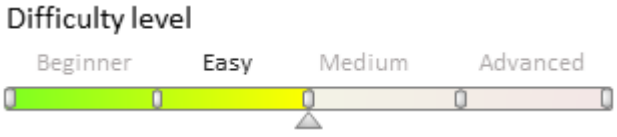
Recommendations

It is recommended to use the [Custom] package in the following cases:

- If the changes will not be transferred to another environment.
- If the changes are made using wizards or manually, and the amount of changes is not large.
- If there is no need to use SVN.

If the changes are significant, it is advisable to create a new custom package using the SVN. For more information on using custom packages please refer to the **Creating and installing a package for development** article.

Creating and installing a package for development



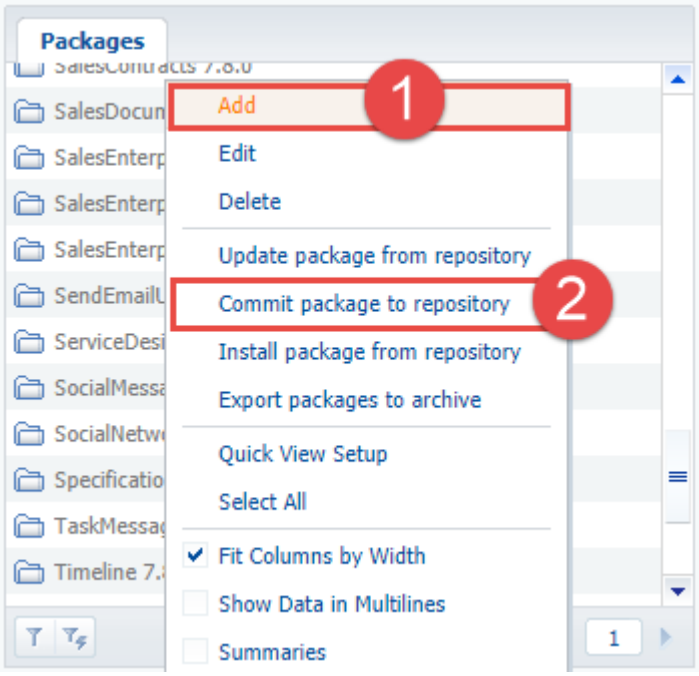
Introduction

A bpm'online package is a collection of configuration elements (schemas, data, scripts, additional libraries) that implements particular block of functionality. In the file system, packages are directories with various subdirectories and files. Basic information about the packages are described in the "**Package structure and contents**” and “**Package dependencies. Basic application packages**”

How to create a custom package

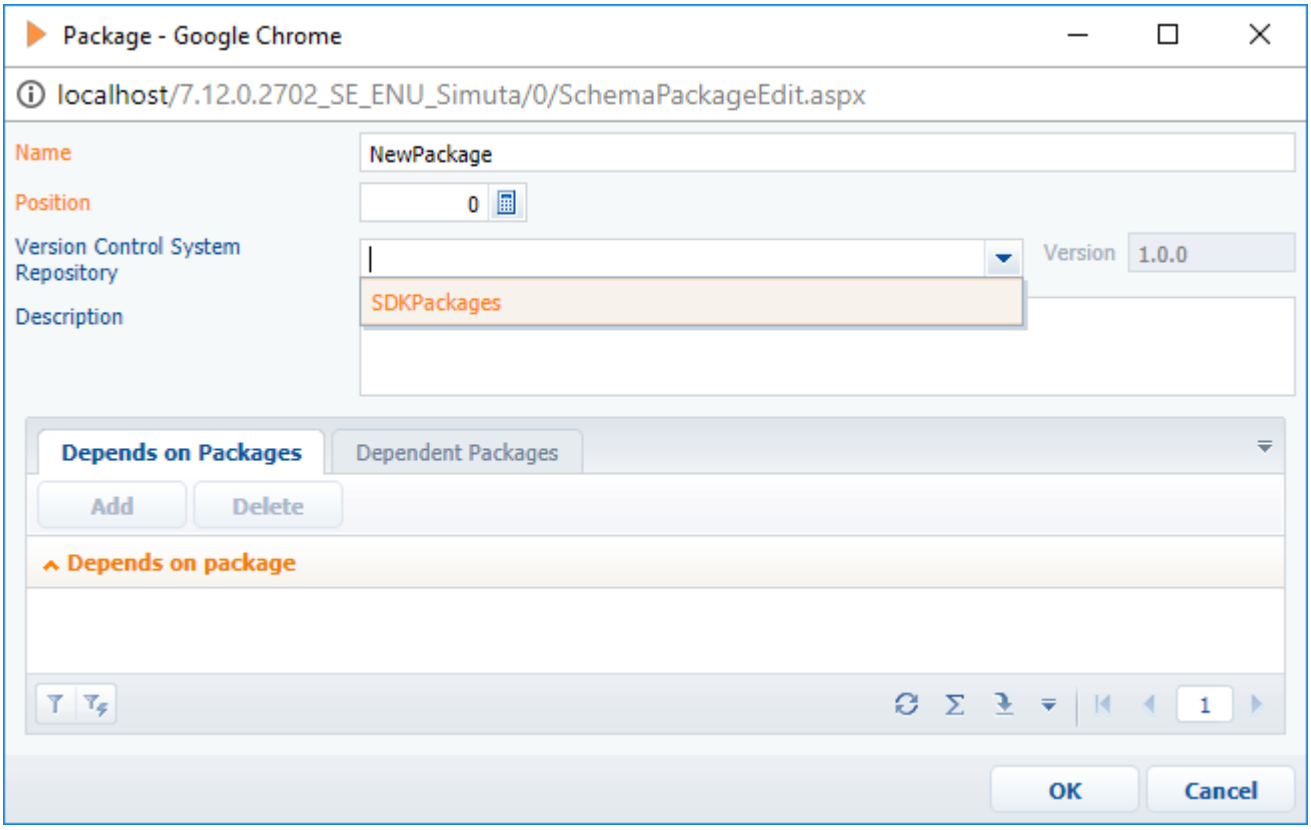
To create a new custom package, go the [Packages] tab menu of the [Configuration] section and select the [Add] action (Fig. 1. 1).

Fig. 1. How to add a new package



As a result, the package mini-page will open (Fig. 2).

Fig. 2. Package mini-page



Main fields of the package mini-page:

- [Name] - package name. Required field. Name cannot match the names of already existing packages.
- [Position] - package position in hierarchy Required field (see "**Configuration localizable resources**"). The default value is set to 0.
- [Revision control system storage] — the revision control system storage name to which package modifications will be committed (see "**The [Configuration] section**"). A list of available storages is generated from the list of storages in SVN. Storage located in the configuration storage list but not marked as active will not appear in the drop-down list of available storages. This is a required field.

ATTENTION

The [Revision control system storage] is populated, when you create a new package and becomes non-editable. If the revision control system is not used, this field is not displayed.

- [Version] - package version. Required field. The package version can contain digits, Latin characters, "." and "_" symbols. The added value text must begin with digits or letters. All elements of the package are of the same version as the package itself. The package version does not necessarily have to match the version of the application.
- [Description] - package description, for example, extended information about package functions. This is a non-required field.

NOTE

When you create a new package, you cannot specify its dependencies yet. Add dependencies, when you edit an already created package.

If a user is not logged in the selected package of the revision control system storage, they will be prompted to authorize before creating a package.

The contents of the key package mini-page fields pack will be saved in its metadata:

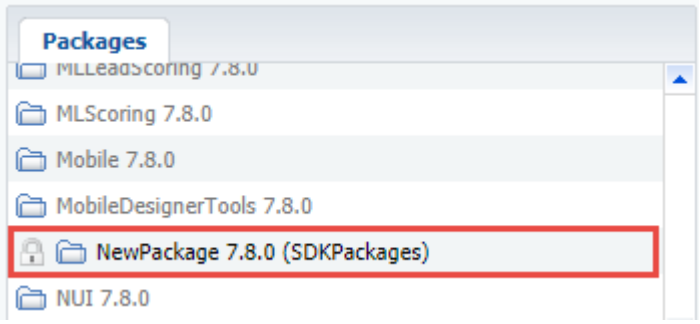
```
{
  "Descriptor": {
```

```
    "UId": "1c1443d7-87df-4b48-bfb8-cc647755c4c1",
    "PackageVersion": "7.8.0",
    "Name": "NewPackage",
    "ModifiedOnUtc": "\/Date(1522657977000)\/",
    "Maintainer": "Customer",
    "DependsOn": []
  }
}
```

In addition to these properties, the package metadata contains information about the **dependencies** (*DependsOn* property) and the developer (*Maintainer*). The *Maintainer* property value is set by using the [Publisher] system setting.

After filling in all the fields and clicking the [OK] button, the package will be created and will appear on the [Packages] tab (Fig. 3).

Fig. 3. New package on the [Packages] tab

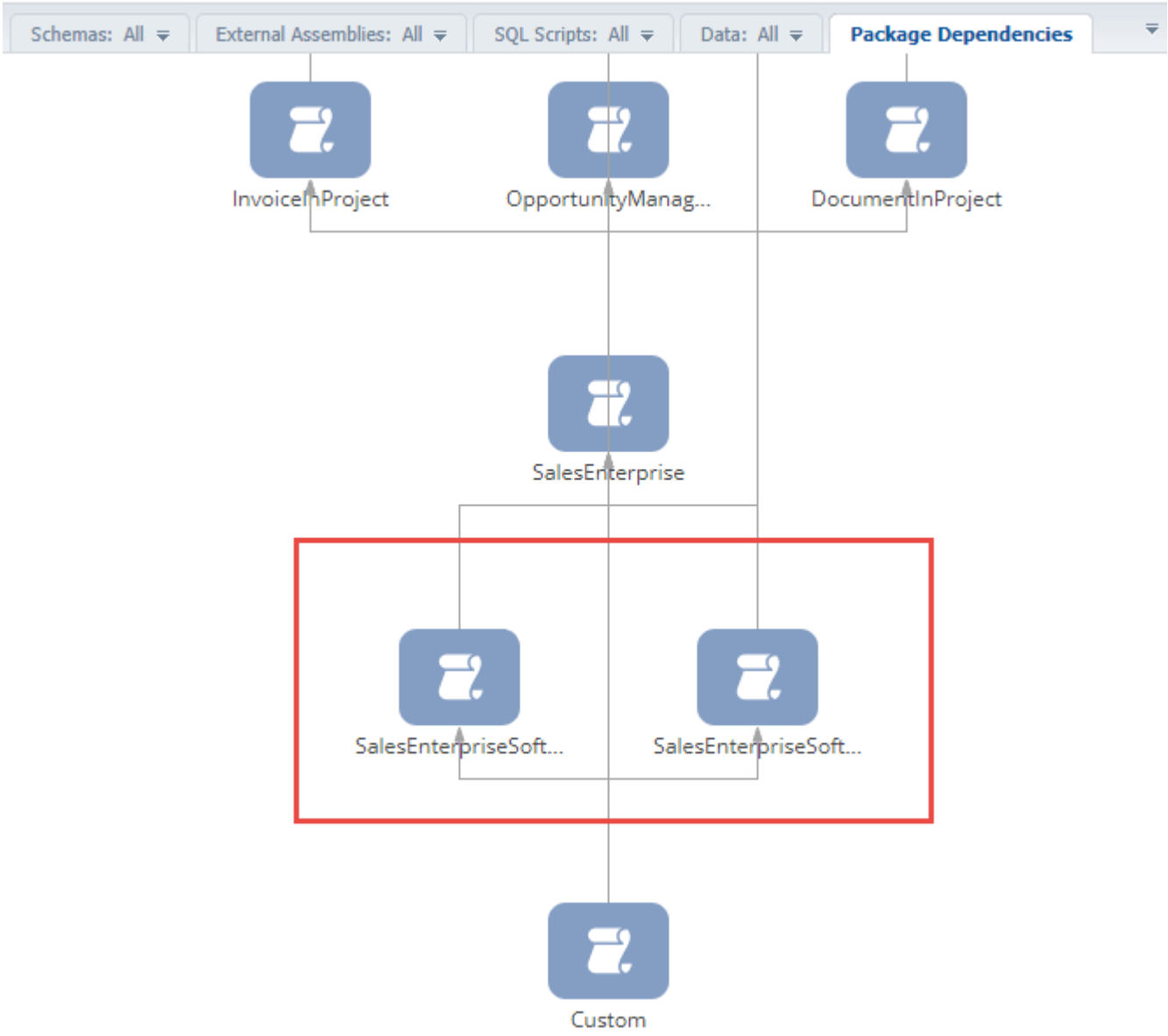


For the created package to have all the functionality that is inherent in the system, you need to specify the dependencies. To do this, indicate the last package in a hierarchy of pre-installed packages. To determine which of the packages in the hierarchy of packages is the latest, you need to go to the [Package dependencies] tab of the [Configuration] section. Next, you must find all packages located one level above the [Custom] package. For example, fig. 4 shows that the [SalesEnterpriseSoftkey_ENU] and [SalesEnterpriseSoftkey_Obsolete] packages are last in the hierarchy of packages. How to add the package to the dependencies is described in the "**Package dependencies. Basic application packages**”

NOTE

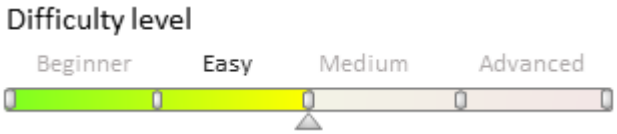
You cannot add the [Custom] package to the dependencies of a new package. The reasons for this are described in the "**Package [Custom]**” article.

Fig. 4. Defining the last package in the hierarchy of pre-installed configuration packages



The [Custom] package must contain all dependencies of all packages of the application. It is therefore necessary to ensure that the [Custom] package **contains the dependency** of the newly created package.

Committing a package to repository



Introduction

Committing package to storage is adding all package modifications to the SVN storage.

> ⚠ ATTENTION
>
> Packages are commited manually. Modifications of other configuration packages are not committed.

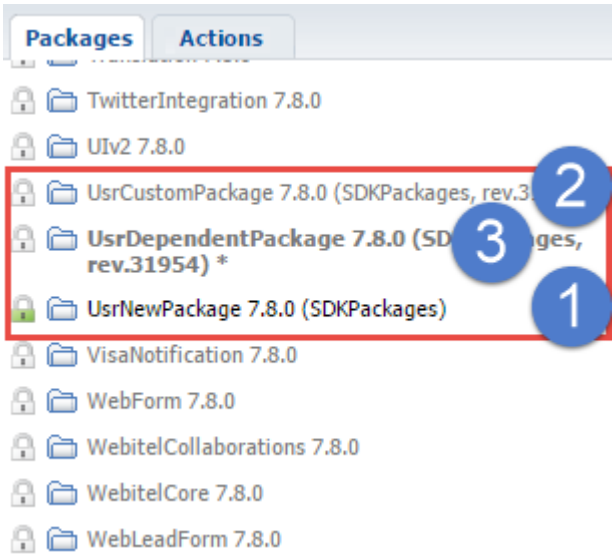
Package committing is required when:

- creating a new package
- adding new and modifying existing package components
- deleting package components
- modifying package properties

> ATTENTION
>
> The information below are applicable when working with SVN repositories via the **bpmonline built-in development tools**. The information are not applicable when the file system design mode is turned on (see "**Working with SVN in the file system**").

The system displays the names of custom packages that have not been committed yet and the name of the storage the packages will be committed to (Fig. 1.1). The SVN revision number is not displayed. and will be added after the committing. Such packages are locked by default.

Fig. 1. Package display



The system displays the names of custom packages that have already been committed, the name of the storage and the latest SVN revision number. The basic package is displayed in the same way as the custom package (Fig. 1.2). If a custom package has been modified, its name will be displayed in bold (Fig. 1.3).

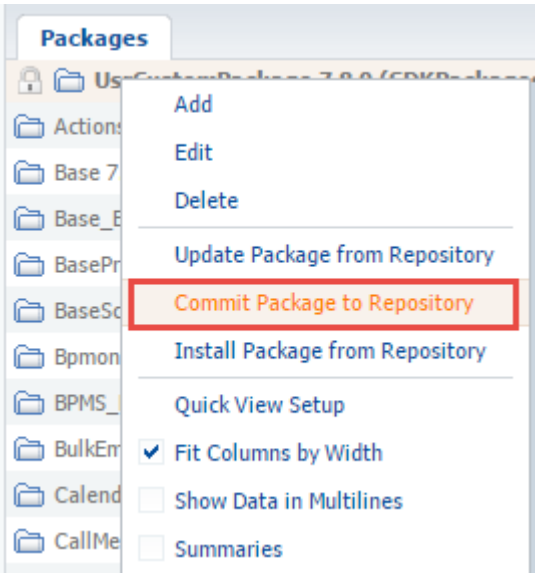
> ATTENTION
>
> If an element was deleted from a package (for example, schema or SQL script), then those modifications won't affect the package display.

Committing a package to storage

To commit a package to storage, first, select it on the [Packages] tab. In the context menu, select the [Commit package to storage] action.

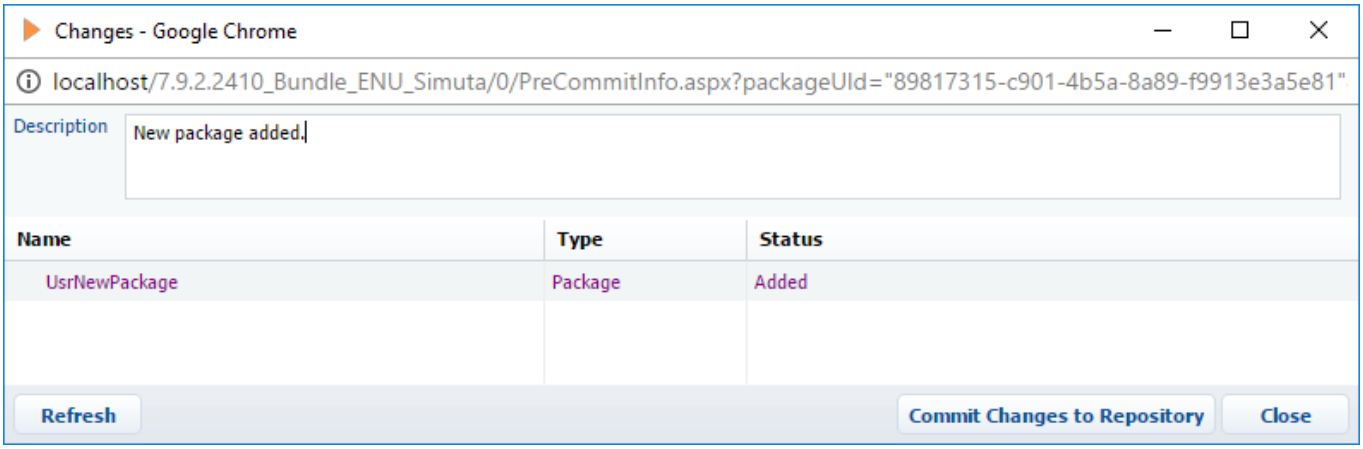
> ATTENTION
>
> When the **file system development mode is enabled**, the SVN integration mechanism is turned off. Therefore, the [Commit package to repository] action is unavailable.

Fig. 3. The [Commit package to storage] action



As a result, the [Changes] window will open (Fig. 4).

Fig. 4. The [Changes] window



You must add a comment in the [Description] window when committing a package. For example, describe the modifications made to the package. The committed files are displayed in the bottom of the window.

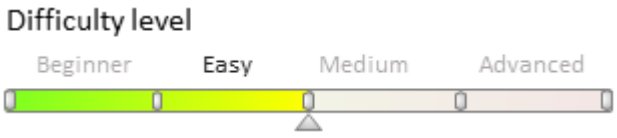
After pressing the [Commit Changes to Repository] button, the package will be committed and the modifications will be become available for other system users.

ATTENTION

The is committed to the storage specified in its properties. Packages can only be committed to an active storage.

When a package is committed, the lock is removed. The package and its components become available for other system users.

Installing packages from repository



Introduction

Installing a package from the repository is adding the package and all its dependencies from the version control system repository (SVN) to bpm’online.

Package installation is required when:

- Multiple developers work on the package functionality.
- Changes are **transferred** between applications.

ATTENTION

The information below are applicable when working with SVN repositories via the **bpmonline built-in development tools**. The information are not applicable when the file system design mode is turned on (see "**Working with SVN in the file system**").

Package installation sequence

Important!

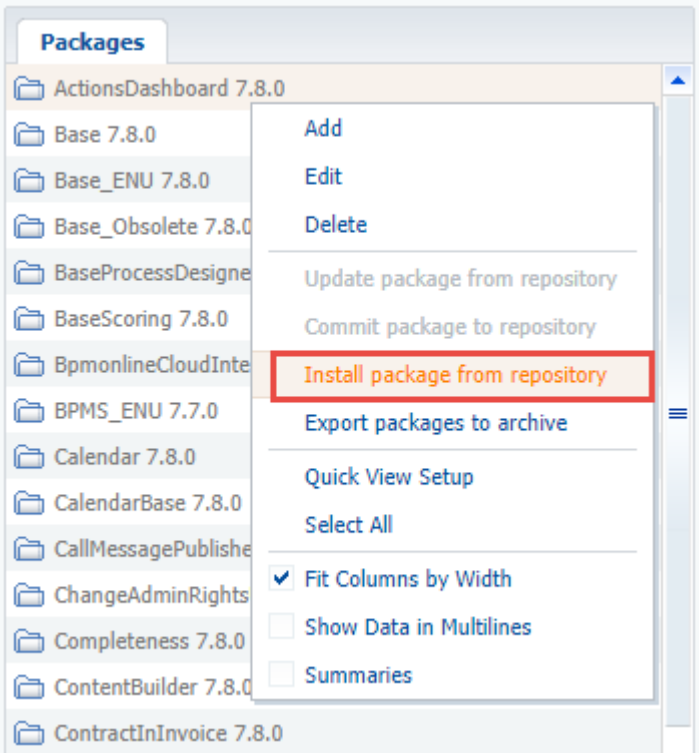
Before updating the application via SVN, you must back up the database. If the application is deployed in the cloud, you should contact support.

Please note that you cannot revert to the previous version of the application via SVN.

The package is installed from the repository using the actions in the [Configuration] section. More details about this section tools can be found in the "**The [Configuration] section**” article.

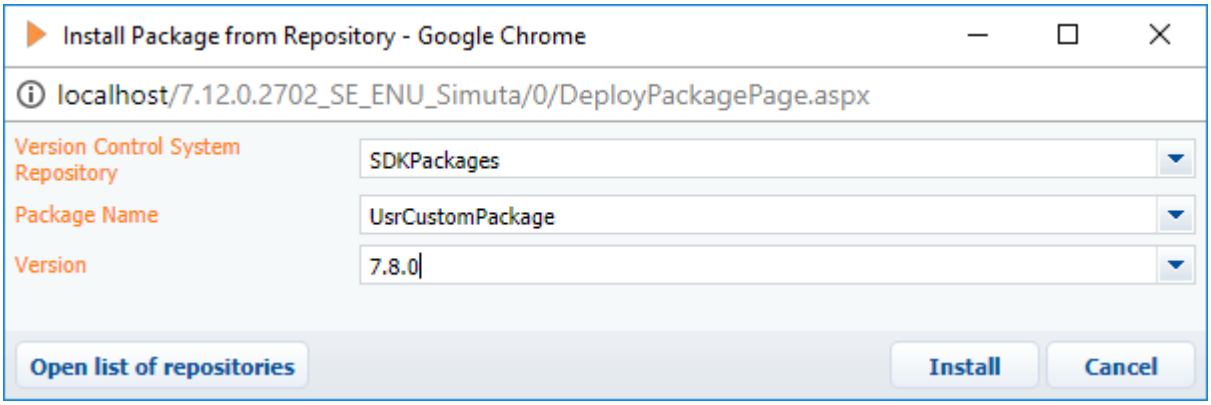
To install the package from the repository, go to the [Configuration] section, right-click the [Packages] tab and select the [Install Package from Repository] option (Fig. 1).

Fig. 1. The [Packages] tab context menu



Then, in the dialog box, select the repository, the name and version of the package to install, and then click the [Install] button (Fig. 2).

Fig. 2. The dialog box for the package installing from the repository



During the package installation, the bound data will be automatically applied, and dependencies will be installed.

If, for any reason, the automatic application of changes has not been enabled, then changes must be applied manually. To do this, perform the following actions for the installed package **in the [Configuration] section**:

1. Generate the source codes for items that require it.
2. Compile the modified items.
3. Update the database structure.
4. Install SQL scripts if necessary.
5. Install the connected data.

NOTE

Checkboxes in the [Database Update Required] and [Require database installation] columns on the [Schemas], [SQL scripts] and [Data] tabs of the [Configuration section] indicate that a schema, script or data needs to be installed in the database or requires modification of the database structure. In case of errors, the text of the last error can be seen in the [Last error message text] column.

Please note that not all of these columns are displayed on the [Schemas], [SQL scripts] and [Data] tabs of the [Configuration] section. If necessary, you can add them using the [Columns setup] context menu.

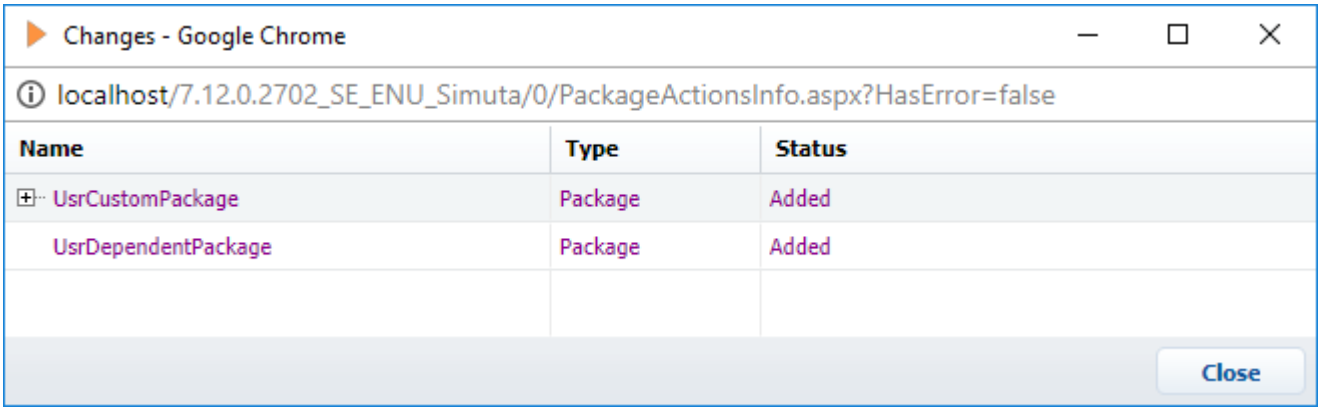
ATTENTION

Starting with version 7.11, after installing or updating a package from SVN, bpm'online application requires compilation (the [Compile all items] action in the [Configuration] section). In the process of compilation, the static content will be generated (see "**Client static content in the file system**" article).

Changes in package hierarchy

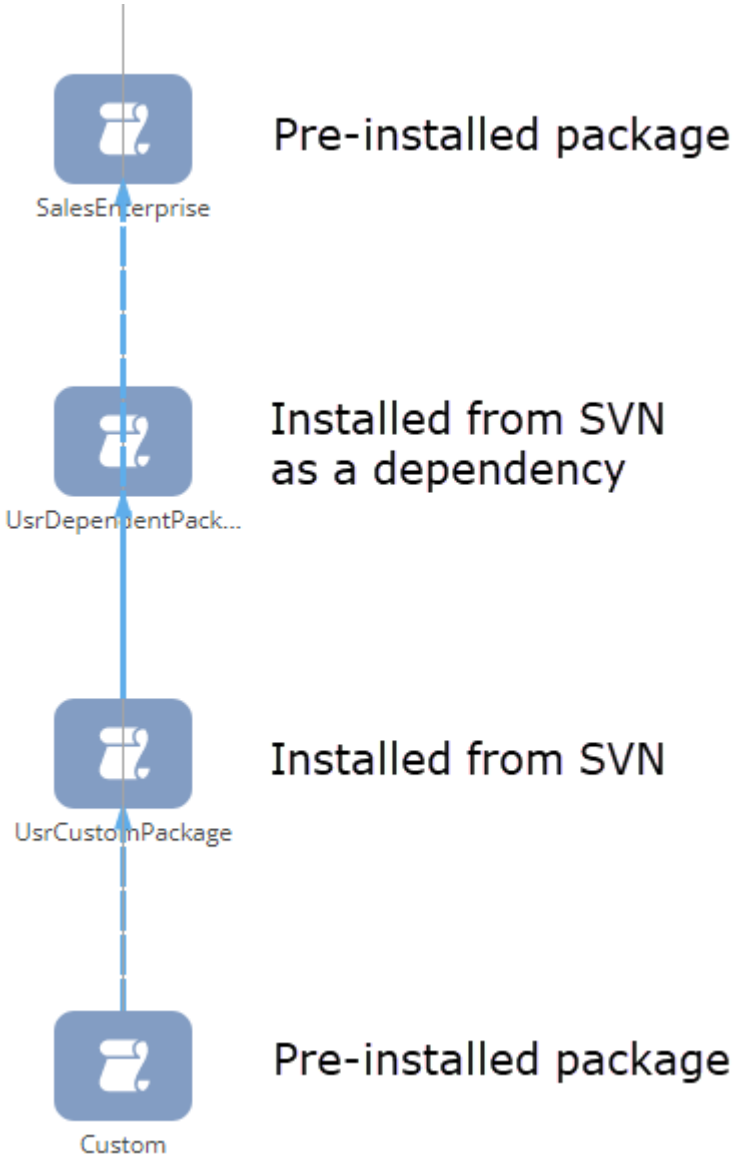
When you install a custom package, the system checks its dependencies and optionally installs or upgrades all the packages the current package depends on. For example, when you install the [UsrCustomPackage] package from the repository, the [UsrDependentPackage] package dependency will also be installed (Fig. 3).

Fig. 3. The [Changes] window after a package has been installed from SVN



This modifies the package hierarchy in the application (Fig. 4).

Fig. 4. New application package hierarchy



When a custom package is installed from SVN, the package hierarchy will be modified in the following way:

1) The application detects all dependencies of the installed package specified in its metadata in the *DependsOn* property.

```
{
  "Descriptor": {
    "UId": "8bc92579-92ee-4ff2-8d44-1ca61542aa1b",
    "PackageVersion": "7.8.0",
    "Name": "UsrCustomPackage",
    "ModifiedOnUtc": "\/Date(1522671879000)\/",
    "Maintainer": "Customer",
    "Description": "Package created by user",
    "DependsOn": [
      {
        "UId": "51b3ed42-678c-4da3-bd16-8596b95c0546",
        "PackageVersion": "7.8.0",
        "Name": "UsrDependentPackage"
      },
```

```
        {
          "UId": "e14dcfb1-e53c-4439-a876-af7f97083ed9",
          "PackageVersion": "7.8.0",
          "Name": "SalesEnterprise"
        }
      ]
    }
}
```

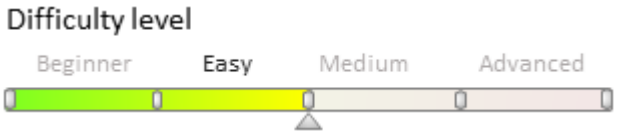
2) Then the system checks whether the package dependencies are installed in the configuration. If the dependencies are installed, they update, if not — the application installs them.

> ⚠ ATTENTION
>
> If the package dependencies are not found in the repository (e.g., repository is not registered or not active), you will see the package installation error message. When you install the package the whole hierarchy of its dependencies is updated, so all repositories that may contain the package dependencies should be included in the configuration and activated.

3) When a package is installed, only the dependencies installed from the version control system (SVN) are installed or updated. Packages installed from zip files and pre-installed packages are not updated.

> ⚠ ATTENTION
>
> If the workspace is missing any pre-installed dependency packages that were installed from zip files, the package installation will fail.
>
> You must first install the packages on which the installed custom package or its dependencies depend on.

Updating package from repository



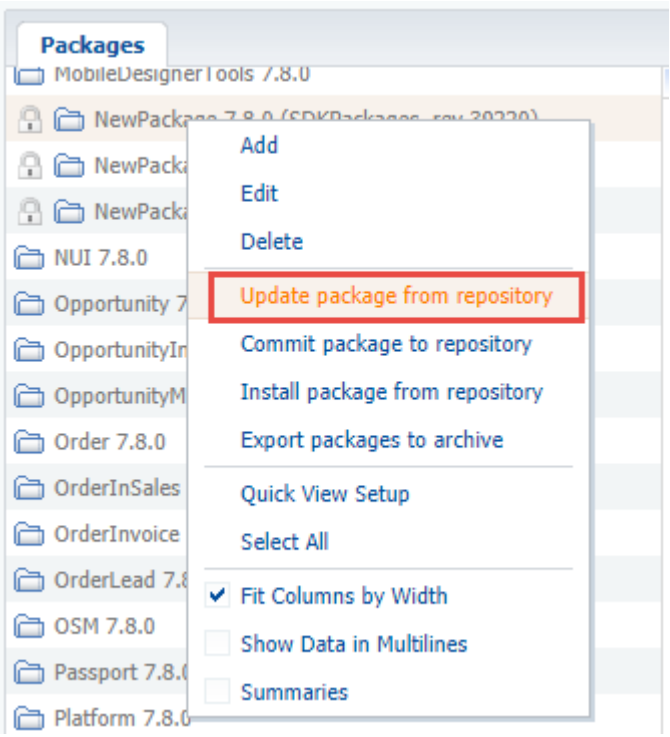
Introduction

The package upgrade process is downloading to the application all changes of the selected package and all its **dependencies** changes from the version control system storage (SVN)

The sequence of package update from SVN

Open the context menu, go to the [Packages] tab and select the [Update package from storage] action (Fig. 1).

Fig. 1. The [Update package from storage] action



This will start the update process of the selected package and all its dependencies from the active SVN storages.

The system will detect all dependencies of the installed package specified in its metadata in the *DependsOn* property. The metadata and the package properties are described in detail in the "**Package structure and contents**" article.

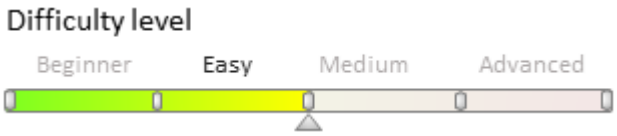
ATTENTION

If package dependencies are located in an inactive storage, the package update error message will pop up. When a package is updated, the whole hierarchy of its dependencies are updated as well, so all SVN storages, which can be contain package dependencies, must be activated.

ATTENTION

Starting with version 7.11, after installing or updating a package from SVN, bpm'online application requires compilation (the [Compile all items] action in the **[Configuration] section**). In the process of compilation, the static content will be generated (see "**Client static content in the file system**" article).

Exporting packages from the application interface



Introduction

To transfer custom packages between non-shared environments (e.g. development and test environments), you must first export these packages to the file system.

Since bpm'online version 7.10.1, packages can be exported directly from the application interface. This enables you to export packages without using the **Workspace Console** utility.

ATTENTION

The [Custom] package cannot be transferred between applications. Learn more about this package in the

"**Package [Custom]**" article.

Exporting packages

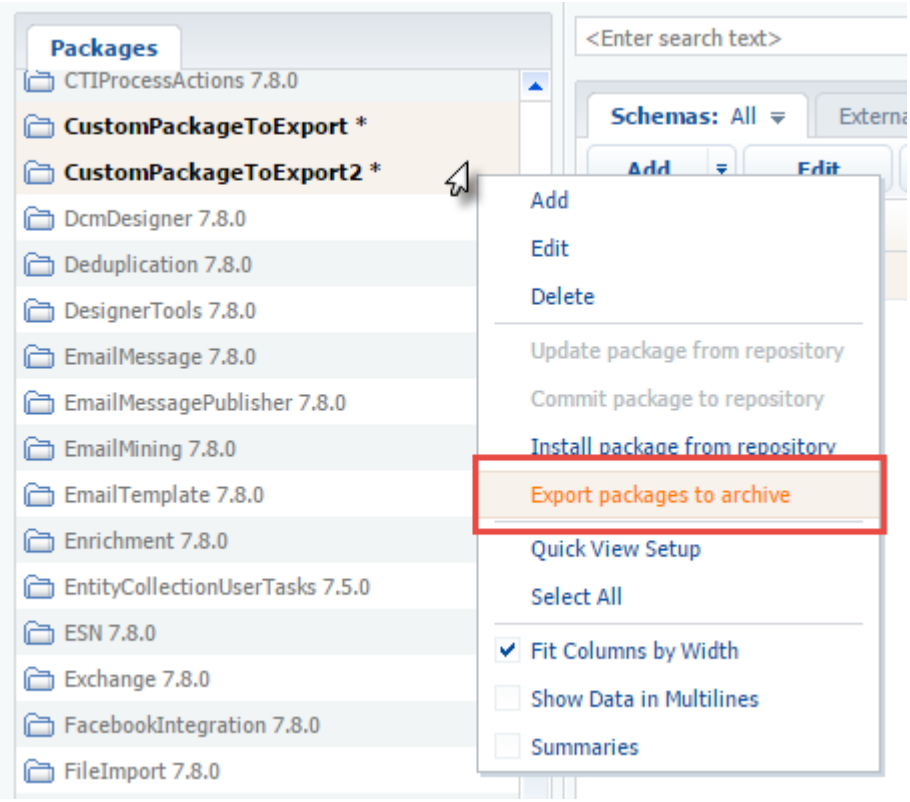
To install packages from the application interface:

1. Go to the **[Configuration] section**.

2. On the [Packages] tab, select one or multiple packages (hold Ctrl or Shift to select multiple packages).

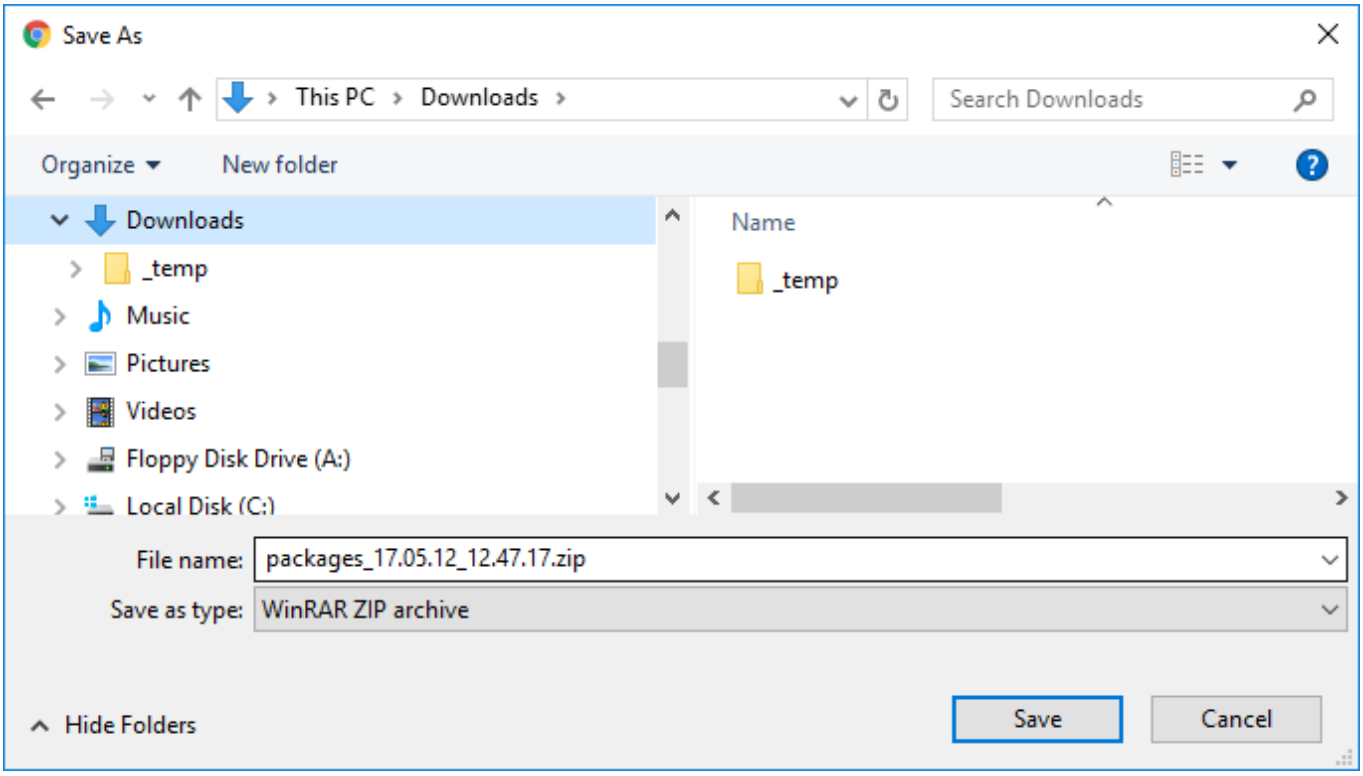
3. Trigger the [Export packages to archive] action (Fig. 1).

Fig. 1. The [Export packages to archive] action



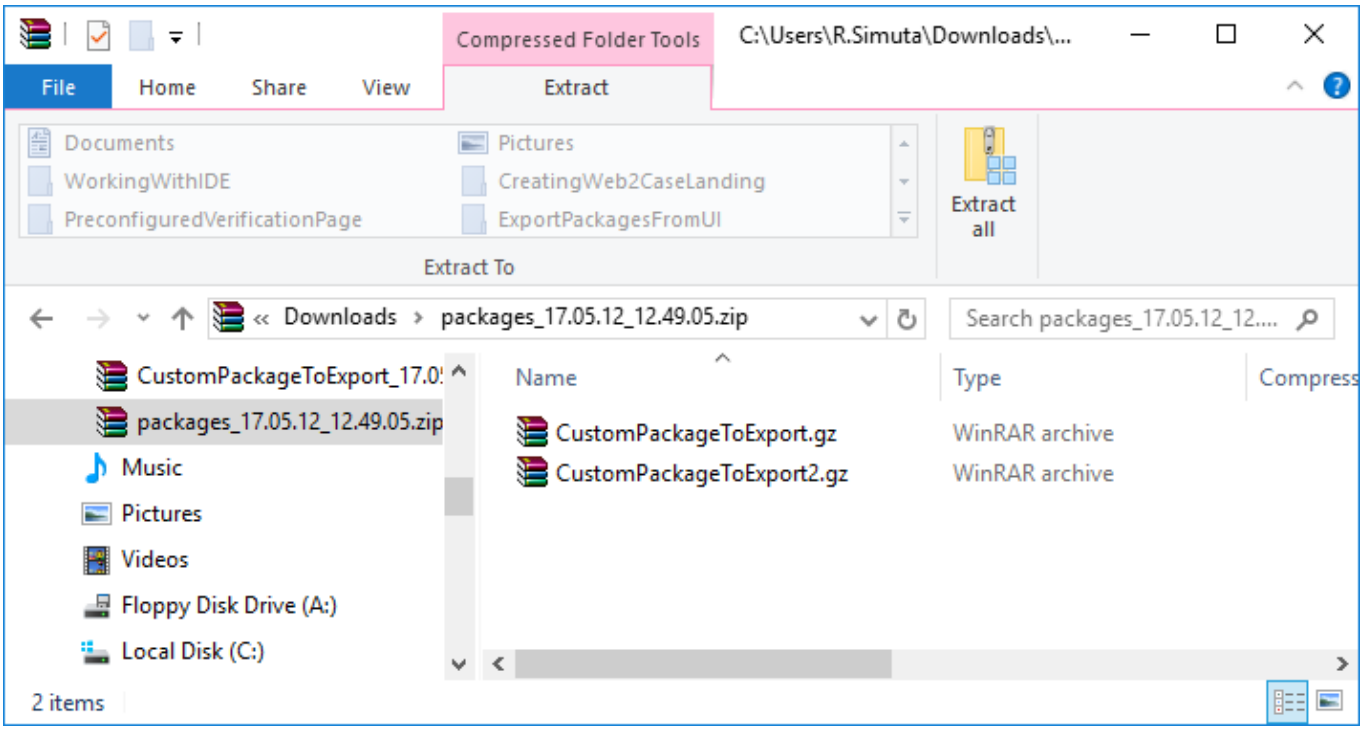
Depending on the browser settings, the zip-archive with packages will either be saved to the default downloads folder, or the browser will display a dialog box for selecting a folder for the archive (Fig. 2).

Fig. 2. A dialog box for selecting a folder for the archive



The zip-archive will contain one or multiple packages (Fig. 3) and can be imported into another bpm'online application (see "**Installing marketplace applications from a zip archive**").

Fig. 3. A zip-archive with packages



ATTENTION

You cannot create packages in a production environment, then create a development environment on the basis of the production environment, finalize the functionality of packages, and transfer them back to the production environment. The development sequence is described in more detail in the "**Recommended development sequence**" article.

Creating a package in the file system development mode

Difficulty level

Beginner Easy Medium Advanced

Introduction

If you do not intend to use SVN in the development process, then the process of creating a package is the file system development mode is the same as that in the normal mode. For more information on creating packages please refer to the “**Creating and installing a package for development**” article.

> Attention!
>
> The working with SVN mode is enabled in the bpm'online by default. If the [Version control system repository] field is empty when a package is created, then the package will not be bound to the repository. The versioned development of this package can be performed only after you manually bind it to the repository from the file system.

When the **file system development mode** is enabled, the SVN integration mechanism is turned off. The packages from the SVN repository can be only **installed** and **updated** with built-in tools. It is recommended to create a package with built-in tools and bind it to the repository with the external utilities like TortoiseSVN.

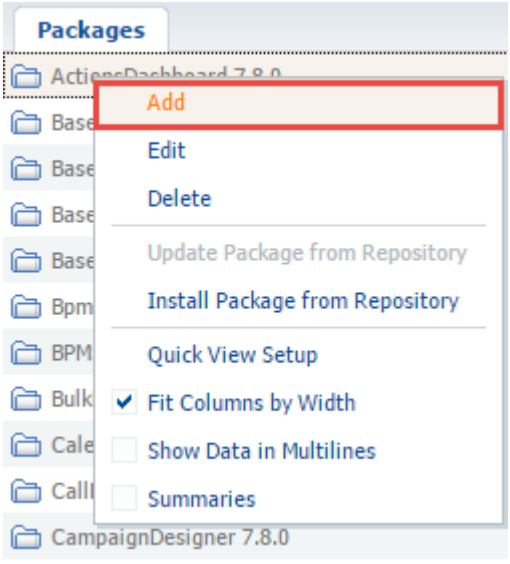
> Attention!
>
> Ensure that **application is configured to access the SVN repository** before binding the package to the SVN repository when development mode in the file system is enabled.

Package creation process

1. Create a package in the application

Select the [Add] action in the context menu of the [Packages] tab in the [Configuration] section (Fig. 1).

Fig. 1 Adding a package in the [Configuration] section

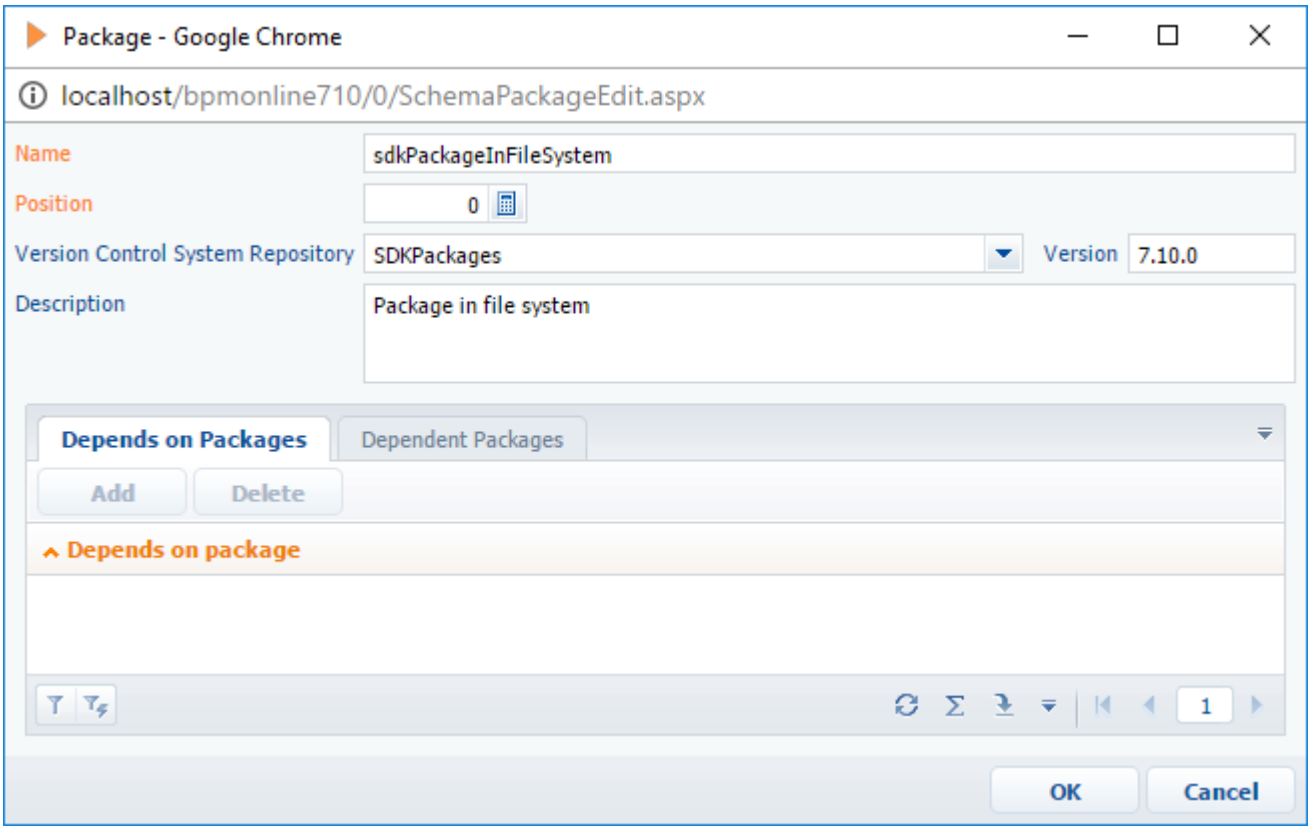


Fill out the main package properties fields in the package edit page (Fig. 2). Please see the "**Creating and installing a package for development**” article for details. Specify the repository name to which the package will be bound.

Attention!

The repository name in the package edit page indicates that the package will be created by third-party tools in this repository. This allows updating the package from the [Configuration] section in future.

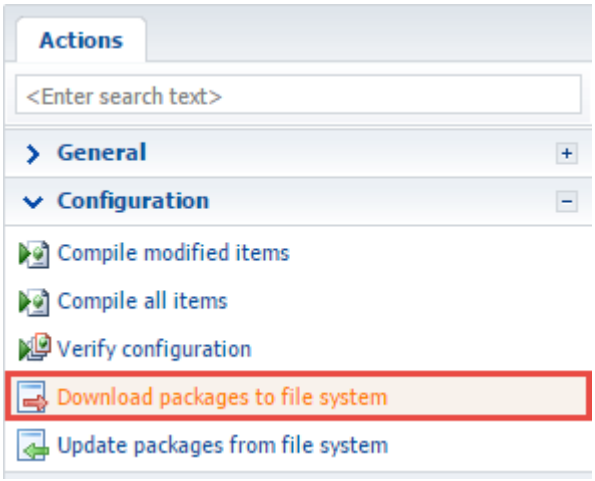
Fig. 2 Package summary



2. Download created package to file system

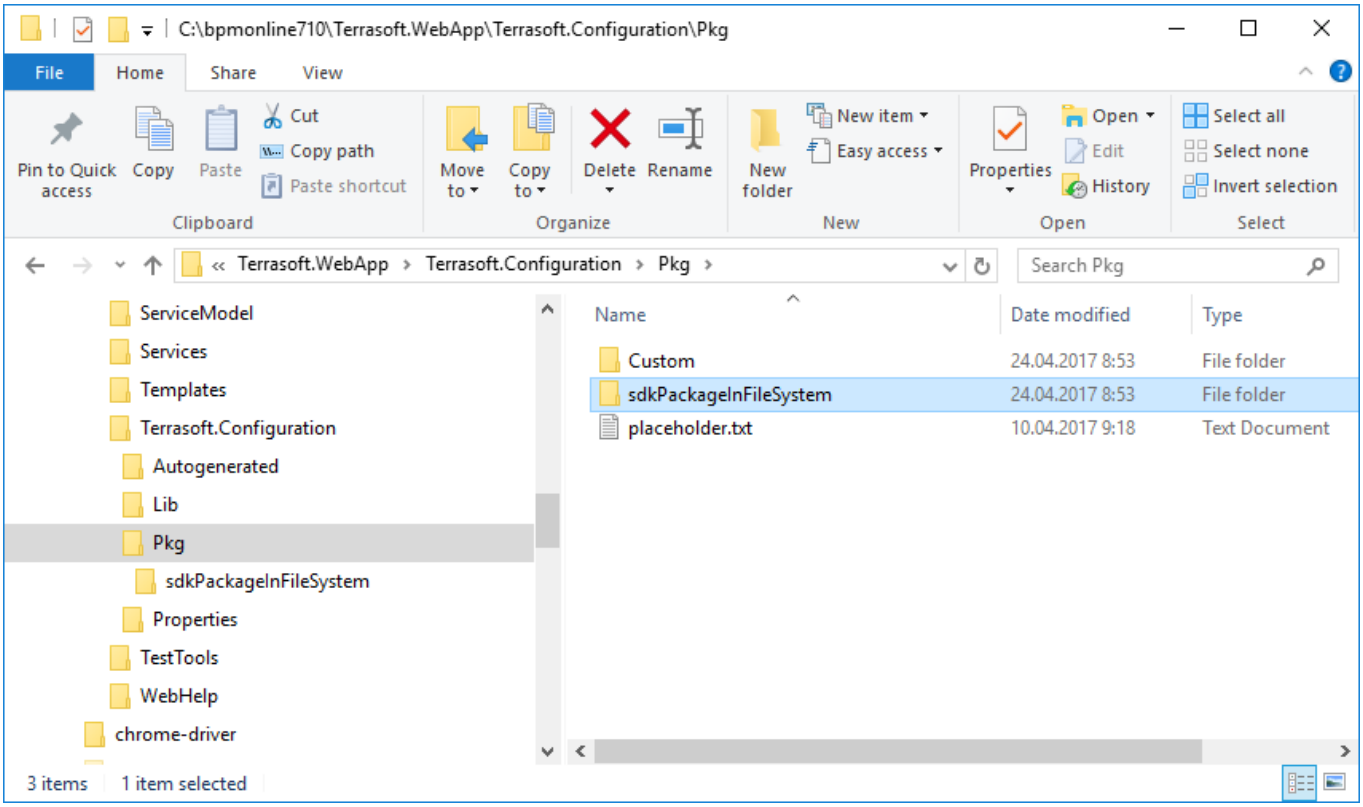
Click [Download packages to file system] (Fig. 3).

Fig. 3 [Download packages to file system] action



As a result, the empty package will be downloaded to the *[Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageInFileSystem* folder (Fig. 4).

Fig. 4 Package in the file system



NOTE

When the file system development mode is enabled, the package must be manually added to the repository.

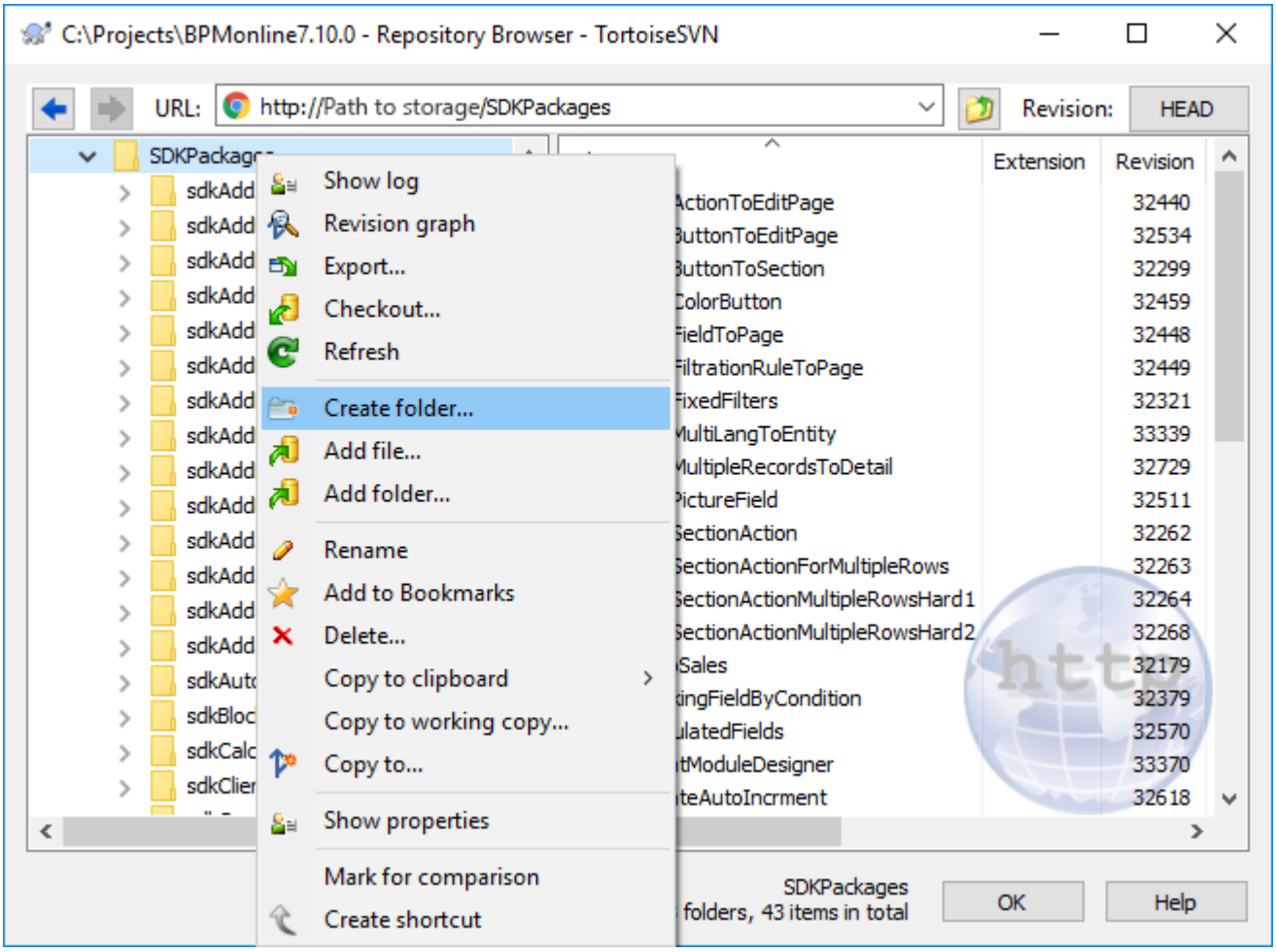
3. Create necessary folders for the packages in the SVN repository

Go to the repository specified in the package edit page to create folders for the package via SVN client (such as TortoiseSvn). Create a folder in the repository with the name that matches the name of the package created in the application.

Attention!

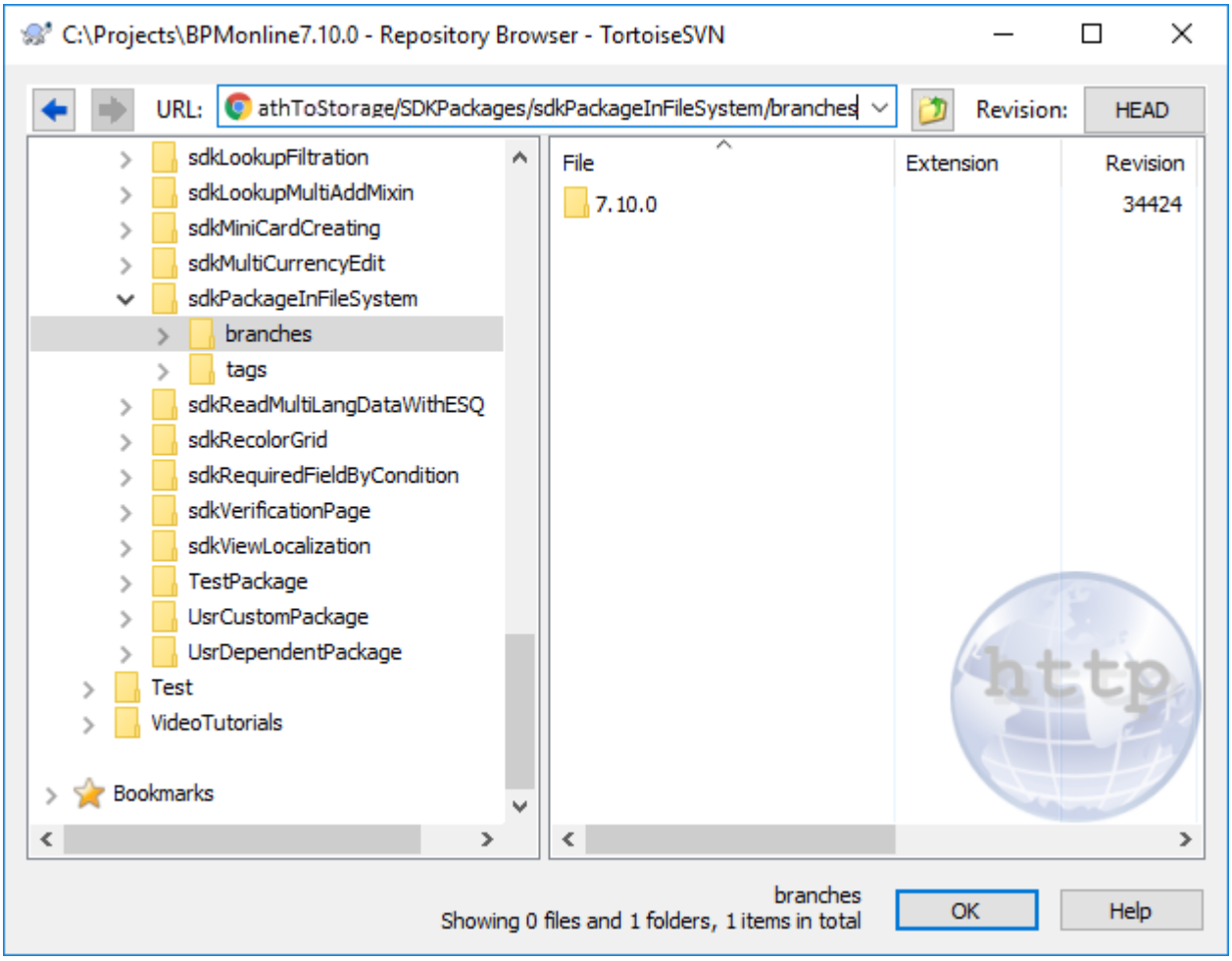
This is a brief example of working with SVN via TortoiseSvn. More information about working with SVN repository via TortoiseSvn can be found in the documentation.

Fig. 5 Creating a folder in the SVN repository



Create *branches* and *tags* sub-folders in the created folder to replicate the bpm’online **flat package structure**. Finally, create a folder with the name that matches the package version number (*7.10.0*) in the *branches* folder (Fig.6).

Fig. 6 Flat package structure in the repository



4. Create a working copy of the package version branch

To create a working copy of the package version branch, execute SVN checkout from the repository folder with the name that matches the package version number, to the package folder in the file system (Fig. 7) and confirm the download to the existing folder (Fig. 8).

Fig. 7 Obtaining the working copy of the package version branch from the repository

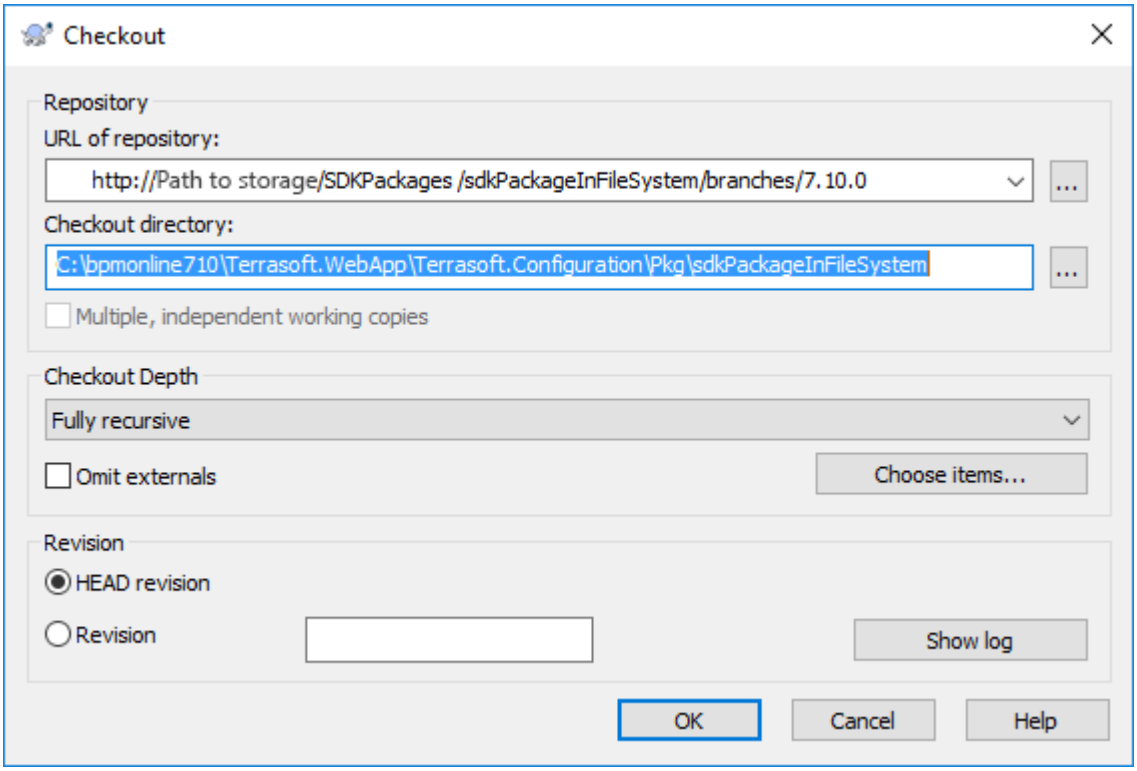
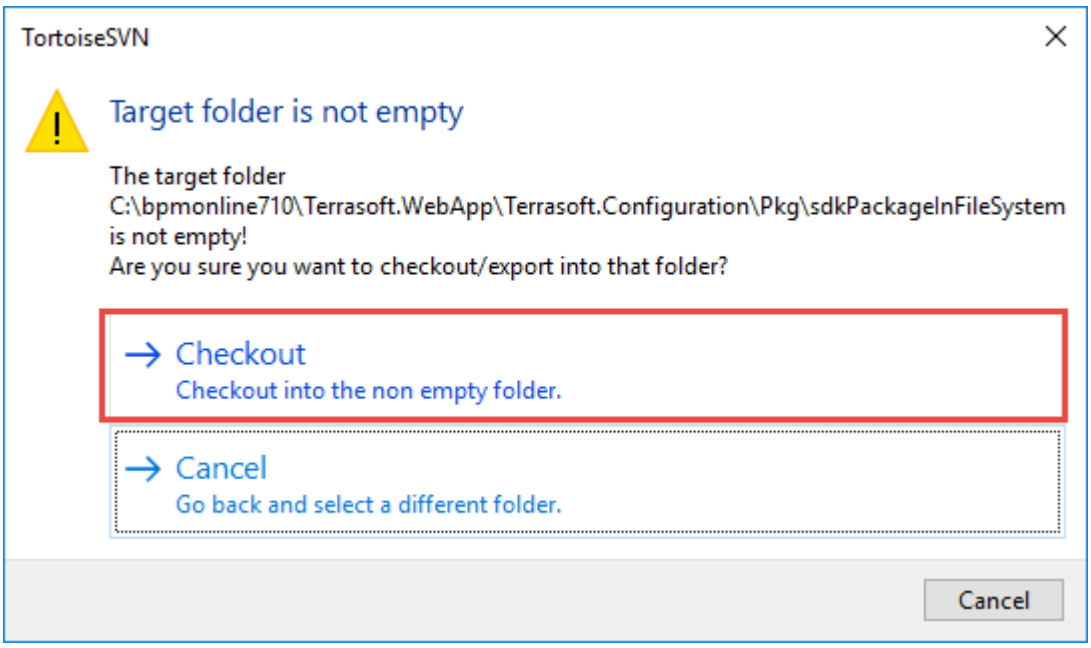
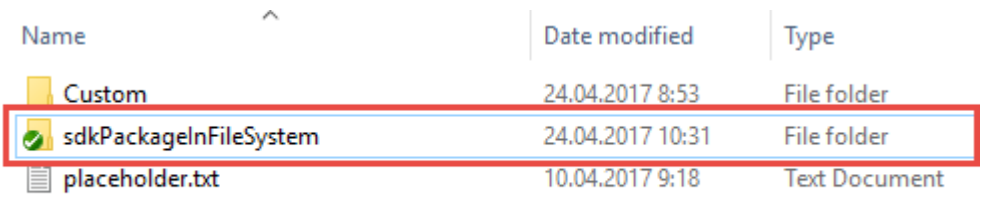


Fig. 8 Confirmation of the SVN checkout operation to the existing folder.



As a result the *[Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageInFileSystem* package folder will be bound to the branch of the 7.10.0 version of the package in the repository (Fig. 9).

Fig. 9 Visual mapping of the bound of the folder with the SVN repository



5. Commit the package folder in the repository

To commit the package folder, add all the contents of the following folder to the repository: *[Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageInFileSystem*. After adding the folder to the repository, execute the “Commit” command (Fig. 11).

Fig. 10 Adding a folder to the repository

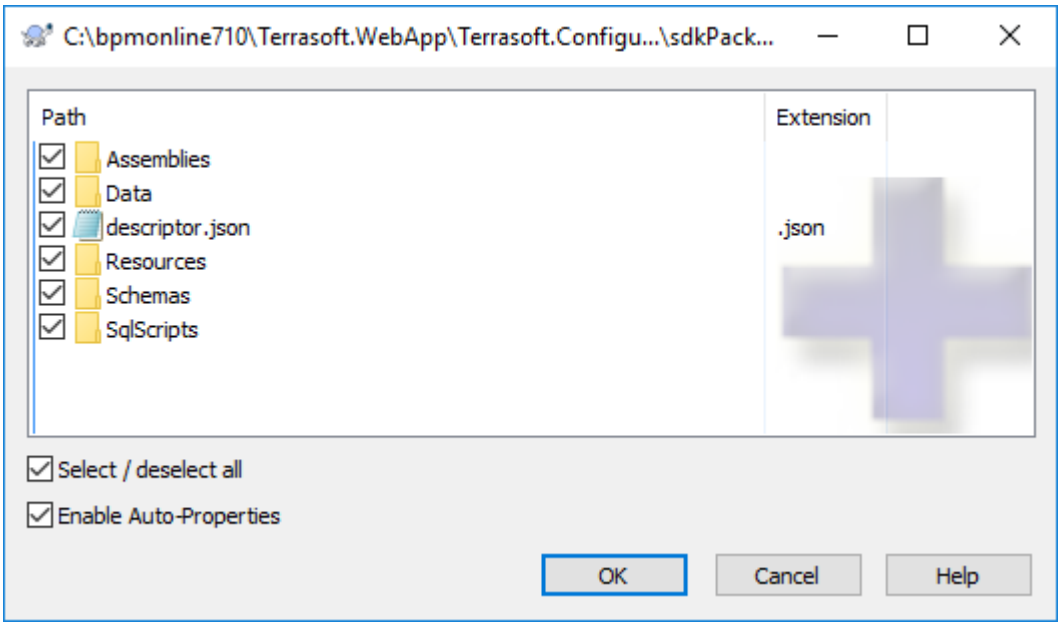
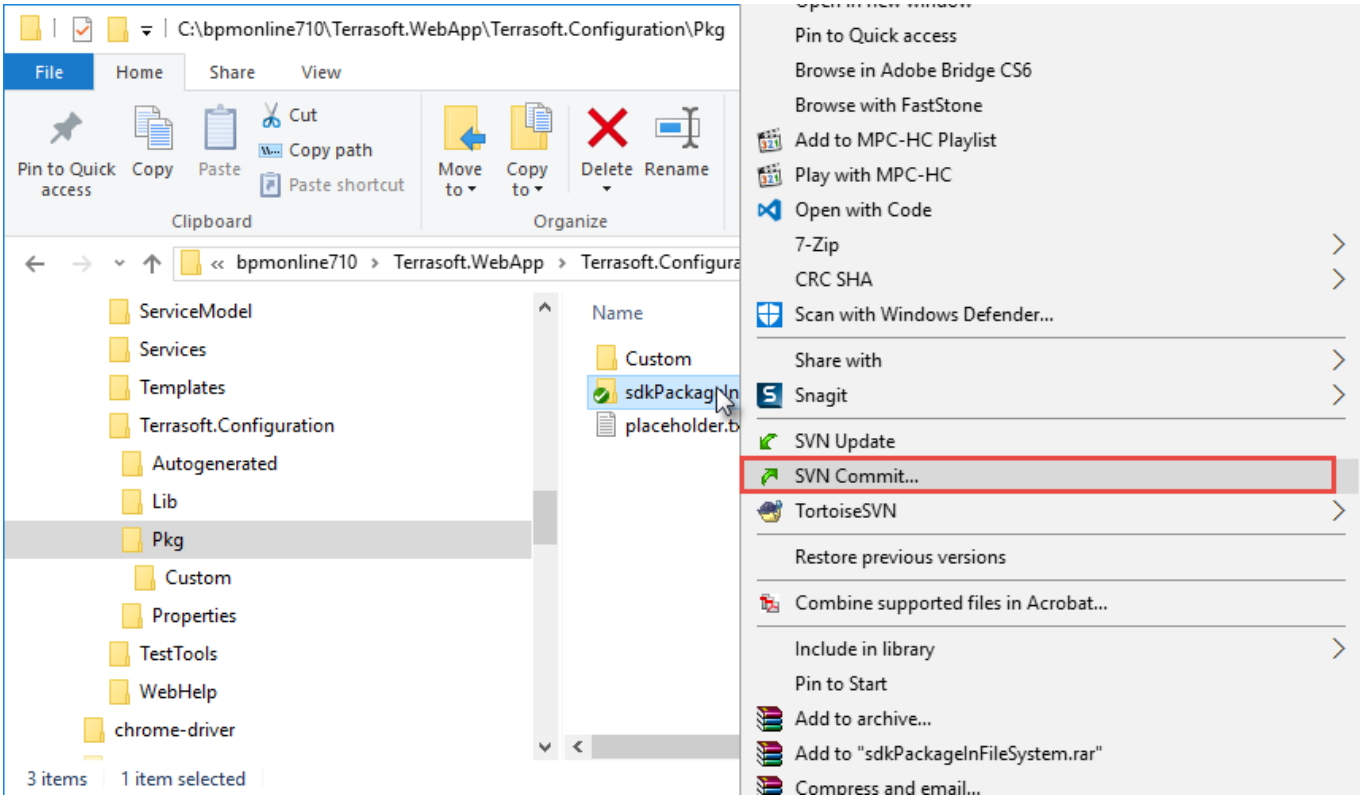


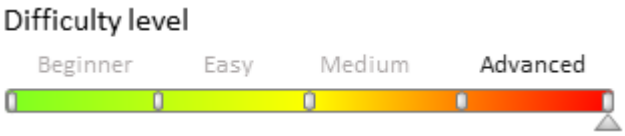
Fig. 11 Committing changes to the repository



See also

- **Working with SVN in the file system**
- **How to install an SVN package in the file system development mode**
- **How to bind existing package to SVN**
- **Updating and committing changes to the SVN from the file system**
- **Creation of the package and switching to the file system development mode**

Binding data to packages



Introduction

It is often necessary to provide certain data together with newly developed functions when delivering custom packages. The data might include lookup values, new system settings, demo section records, etc.

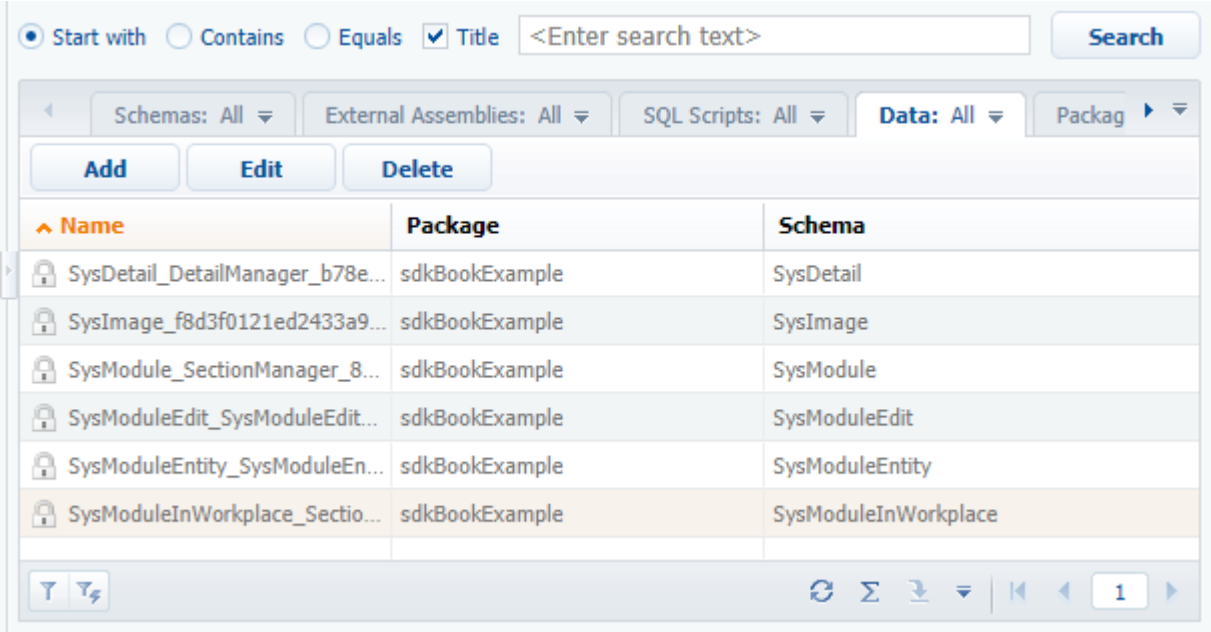
Use the [Data] tab of the [Configuration] section to bind needed data to a package containing the developed function. You can find general information about the tab including recommendations and common data binding mistakes in the "**The [Configuration] section. The [Data] tab**" article.

Case description

Binding two demo records and their linked records from other sections for the [Books] custom section.

> ⚠ ATTENTION
>
> When adding a section, the data necessary for registration and correct section operation are linked to the package via wizard (Fig.1).

Fig. 1. Data collection linked to the package via wizard



Source code

You can download the package with case implementation using the following link.

Case implementation algorithm

1. Adding a new [Books] section

> ⚠ ATTENTION
>
> Create a new section function in a **separate developer package**. Select the developer package in the [Default value] column of the [Current package] system setting to create schemas in the developer package via section wizard. After wizard operation is over, you can set up **Custom package** as your current package.

Use the section wizard to add a new [Books] section. Section properties and field location on record edit page are shown on fig. 2 and fig. 3.

Fig. 2. The [Books] section properties

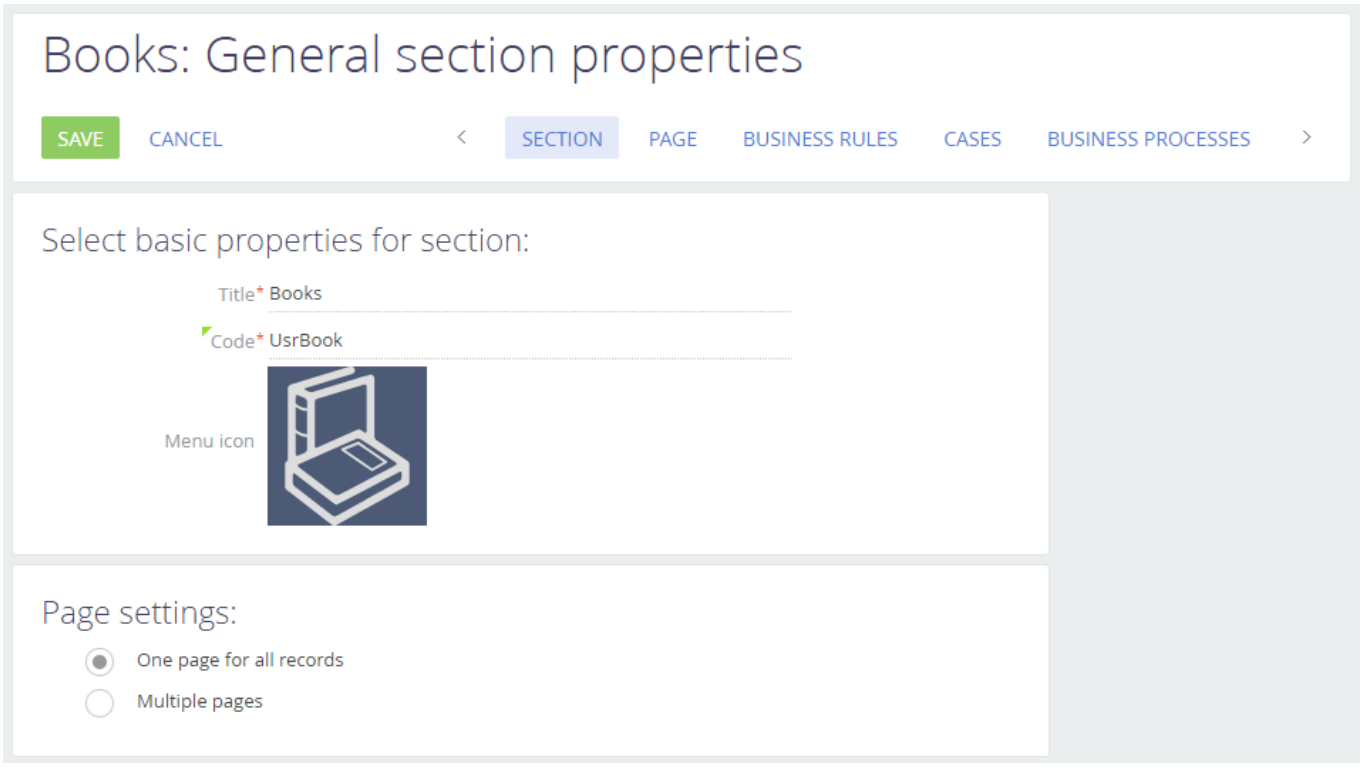
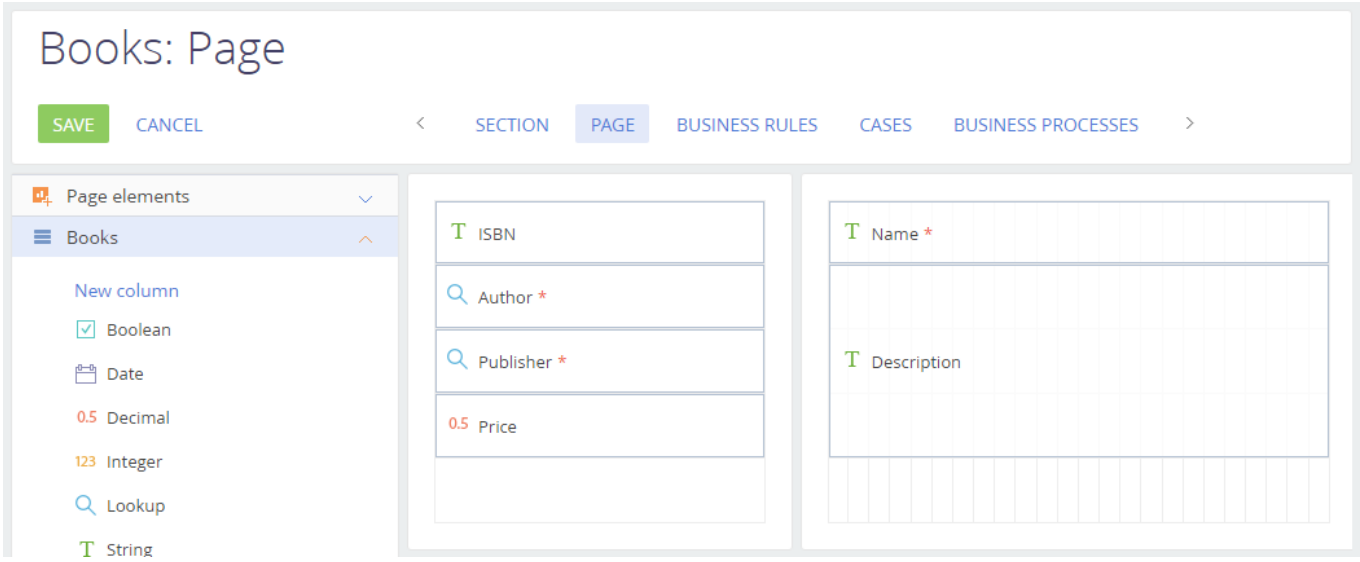


Fig. 3. Record edit page properties



The base properties of edit page columns are specified in table 1.

Table 1. Edit page column properties of section records

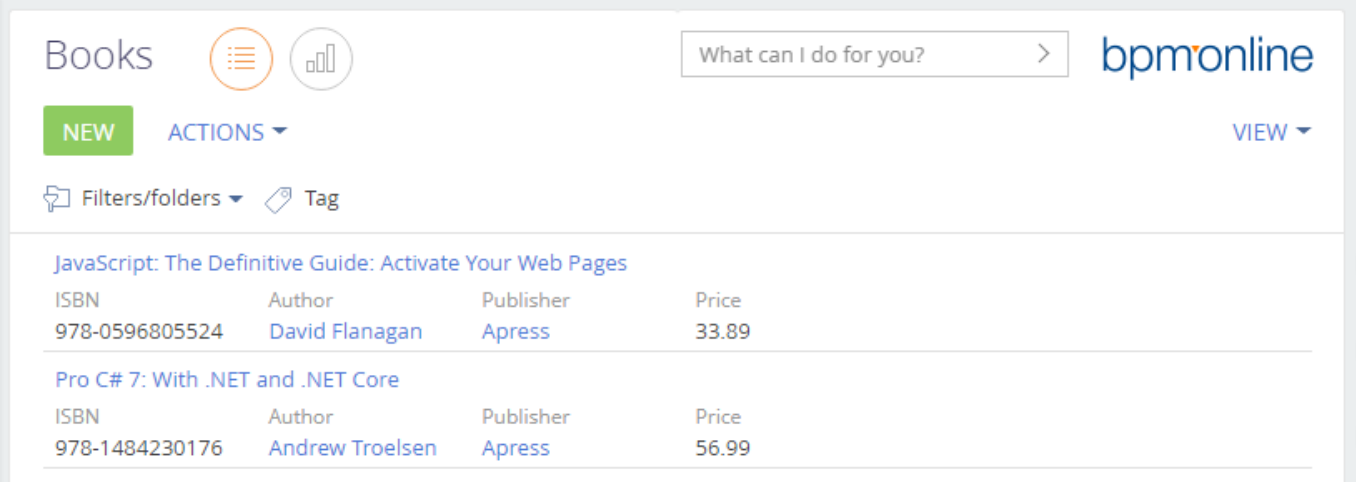
Title	Name (Code in DB)	Data type
Name	UsrName	String
Description	UsrDescription	String Multiline text
ISBN	UsrISBN	String
Author	UsrAuthor	The [Contact] lookup. The column value will be bound to one of the [Contacts] section records.
Publisher	UsrPublisher	The [Account] lookup. The column value will be bound to one of the [Accounts] section records.

Price	UsrPrice	Decimal

2. Adding needed records to the section

Add two demo records to the section (Fig.4). Add records to the bound [Contacts] and [Accounts] sections if needed.

Fig. 4. Section records



3. Binding contact data to the package

Since the [Books] section records are bound to the [Contacts] section records by the UsrAuthor column, bind the author data to the package first. Run the [Add] command on the [Data] tab of the [Configuration] section and set up the following properties of the **page for binding data to package** (Fig/5):

1. [Name] – “ContactsInBooks”

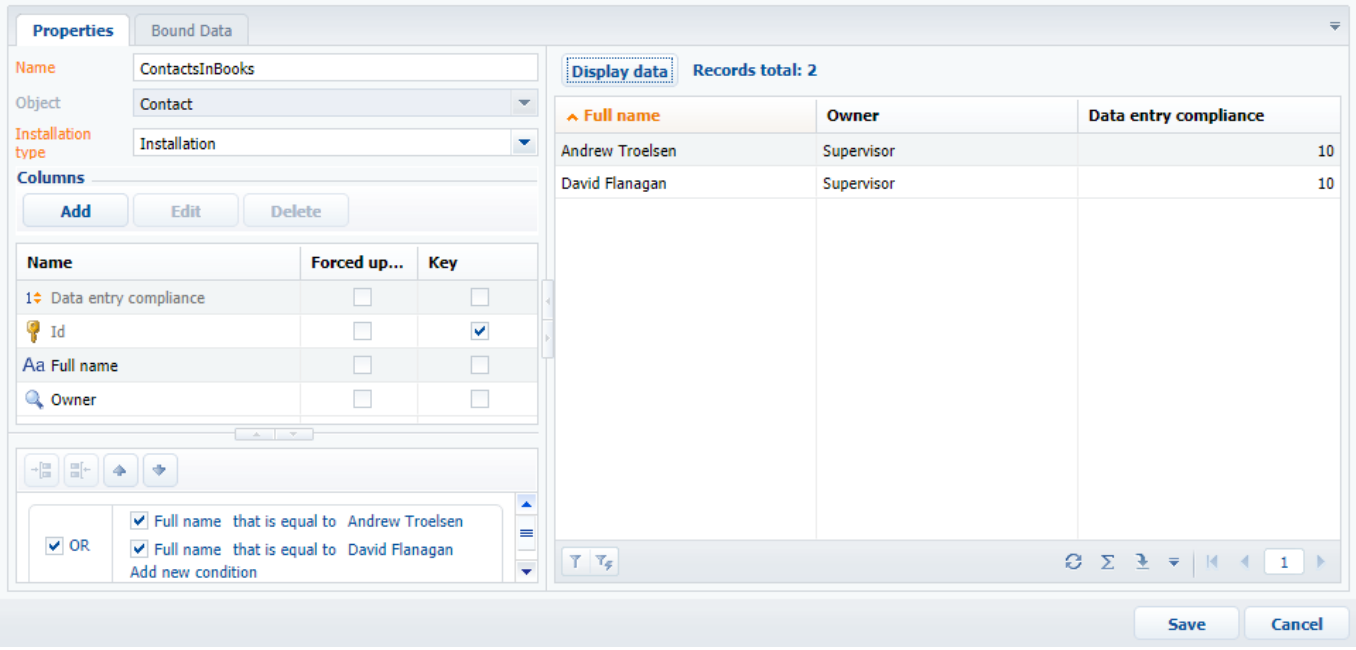
2. [Object] – “Contact”

3. [Installation type] – “Installation”. Possible installation types are described in the "**The [Configuration] section. The [Data] tab**" article.

4. [Columns] – select only the populated columns. It is required to select the [Id] column.

5. Data filtering – filter the needed data, for example, by contact name.

Fig. 5. Page for binding contact data to package



> **NOTE**
>
> Filtering by the *Id* column is recommended (see the following step), since the full contact name can be changed.

4. Binding account data to the package

Run the [Add] command on the [Data] tab of the [Configuration] section and set up the following properties of the **page for binding data to package** (Fig/6):

1. [Name] – "AccountsInBooks"

2. [Object] – "Account"

3. [Installation type] – "Installation"

4. [Columns] – select only the populated columns. It is required to select the [Id] column.

5. Data filtering – filter the needed data, for example, by account identifier. You can determine the identifier from the browser address bar by opening the needed record edit page (Fig. 7)

Fig. 6. Page for binding account data to package

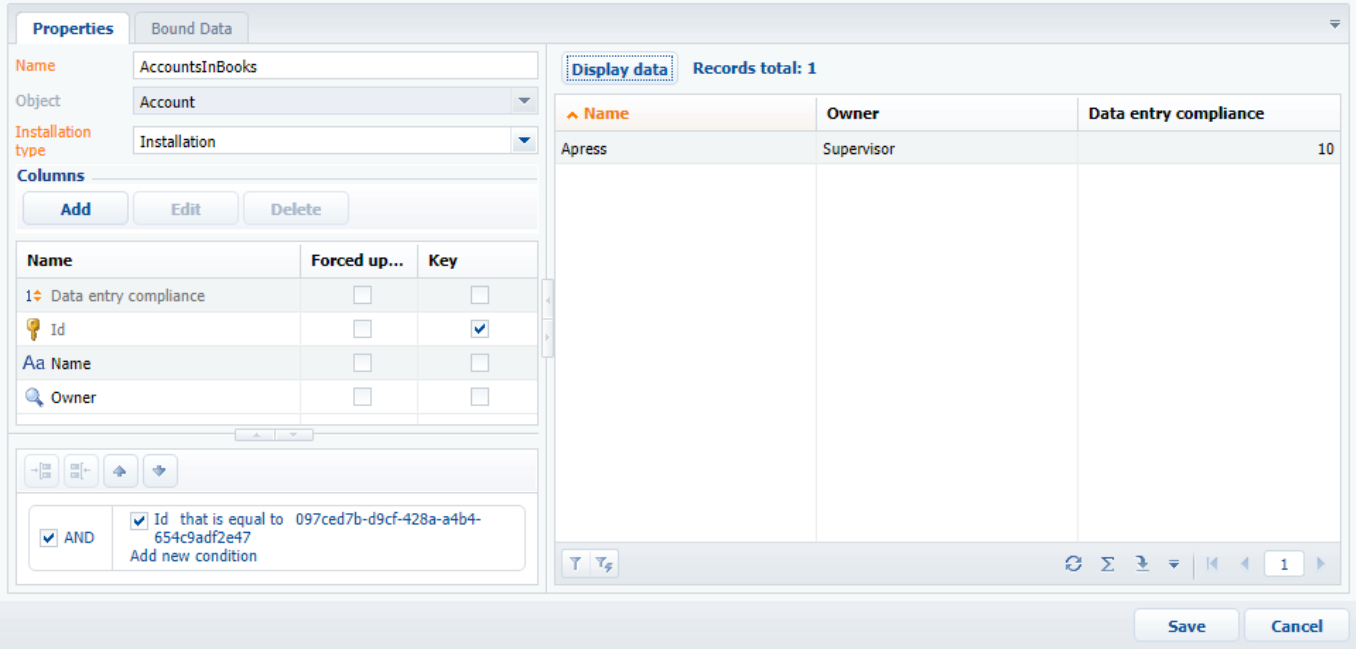
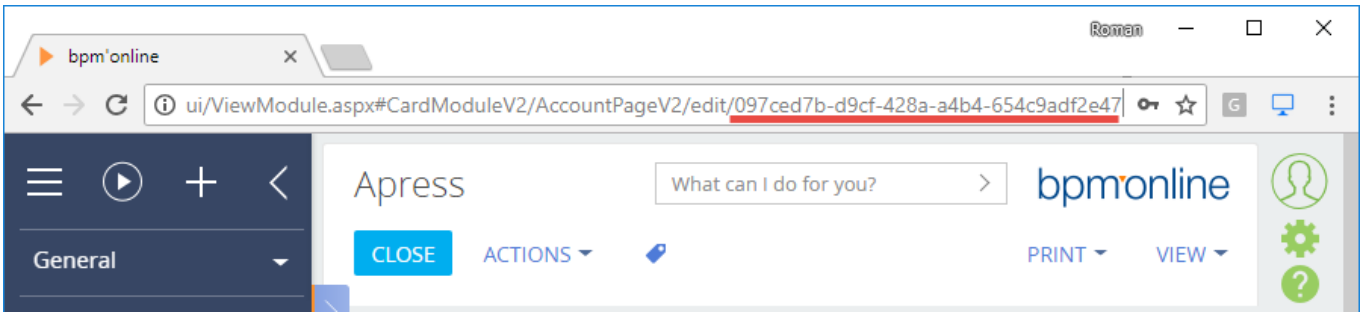


Fig. 7. Determining the account identifier



5. Binding custom package data to the package

Run the [Add] command on the [Data] tab of the [Configuration] section and set up the following properties of the **page for binding data to package** (Fig/8):

1. [Name] – "Books"

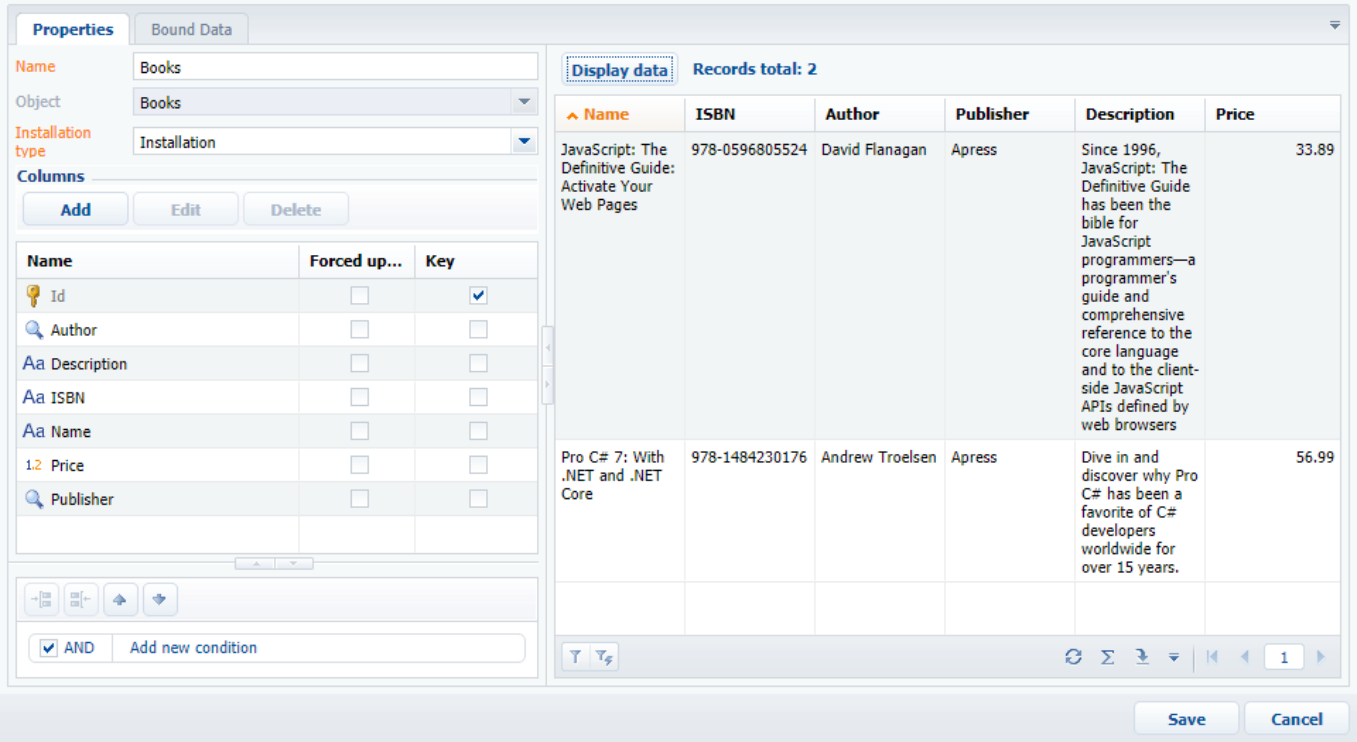
2. [Object] – "Books"

3. [Installation type] – "Installation"

4. [Columns] – select only the populated columns. It is required to select the [Id] column.

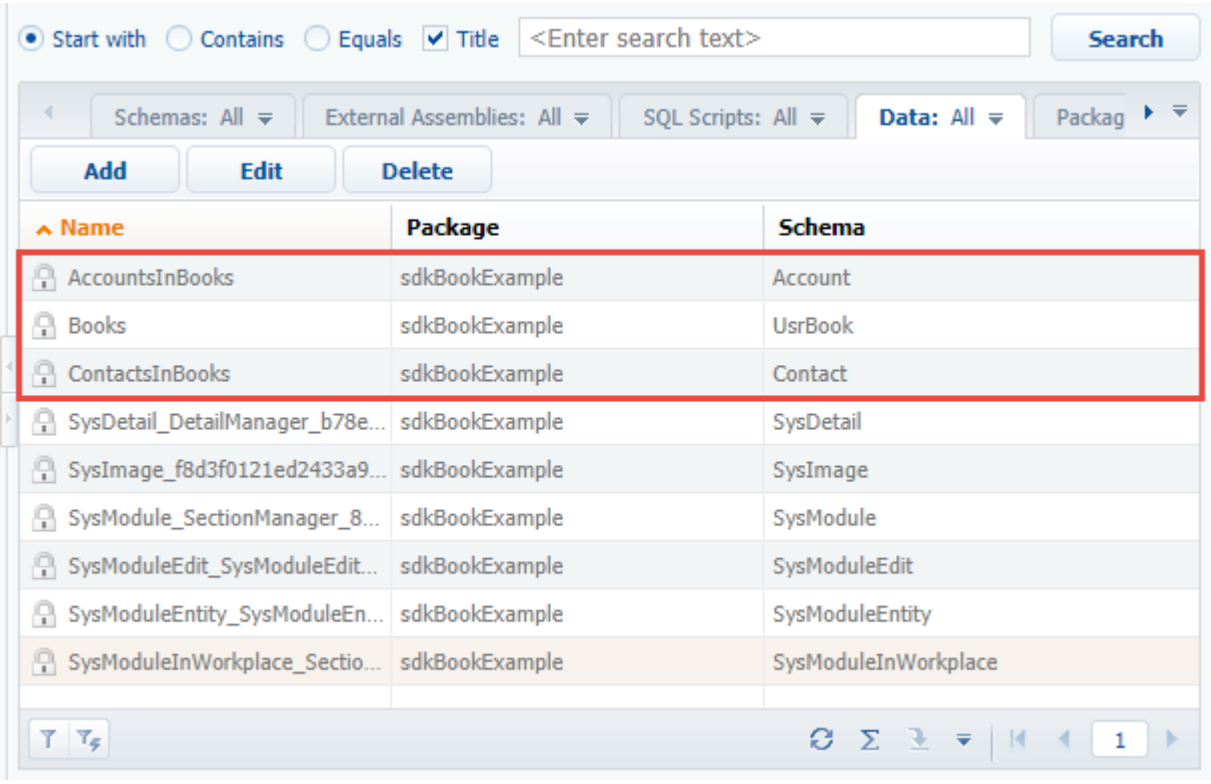
5. Data filtering – filter the needed data. If a section contains only two records you can avoid using the filter (Fig.8).

Fig. 8. The page of binding custom section data



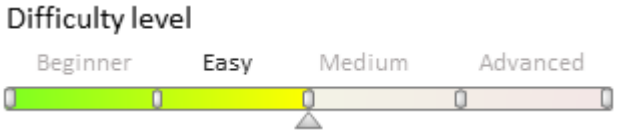
As a result of case implementation, three additional data collections for three sections will be bound to the package (Fig.9).

Fig. 9. The [Data] tab of the developer package



You can export packages to archive using the corresponding function (see “**Exporting packages from the application interface**”). After you install the package into another application, the bound records will be displayed in the corresponding sections.

Transferring changes between the working environments



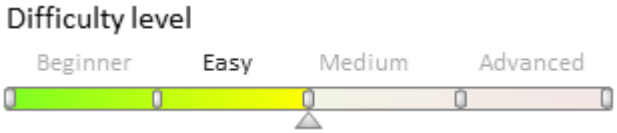
Overview

It often becomes necessary to transfer individual changes between the **working environments** (applications) in the development process, for example, when you install changes made in the development environment to a test environment.

There are several ways to transfer changes between configurations in the bpm'online:

- To transfer custom packages between non-shared environments (e.g. development and test environments), you must first export these packages to the file system. How to do this, see the "**Exporting packages from the application interface**" article.
- Since bpm’online version 7.11 marketplace application can be installed from .zip archive. The marketplace application is a set of bpm'online pacakges. Installation functions of the marketplace applications are available in the [Installed applications] section. More information about installation functions can be found in the "**Installing marketplace applications from a zip archive**" article.
- Schema import and export is used for transferring one or more schemas, for example, when exchanging incomplete features between developers. For more information about this method, see the "**Transferring changes using schema export and import**” article.
- Use version control system if you need to manage changes in different versions during development. This option is also recommended if several developers are working with the same schemas simultaneously. For more information about this feature, see the "**Transferring changes using SVN**” article.
- If you install service packs provided by third-party developers, use the WorkspaceConsole utility. A detailed procedure for uploading and downloading packages with the WorkspaceConsole utility is described in the “**Transferring changes using WorkspaceConsole**” article.

Exporting packages from the application interface



Introduction

To transfer custom packages between non-shared environments (e.g. development and test environments), you must first export these packages to the file system.

Since bpm’online version 7.10.1, packages can be exported directly from the application interface. This enables you to export packages without using the **Workspace Console** utility.

ATTENTION

The [Custom] package cannot be transferred between applications. Learn more about this package in the "**Package [Custom]**" article.

Exporting packages

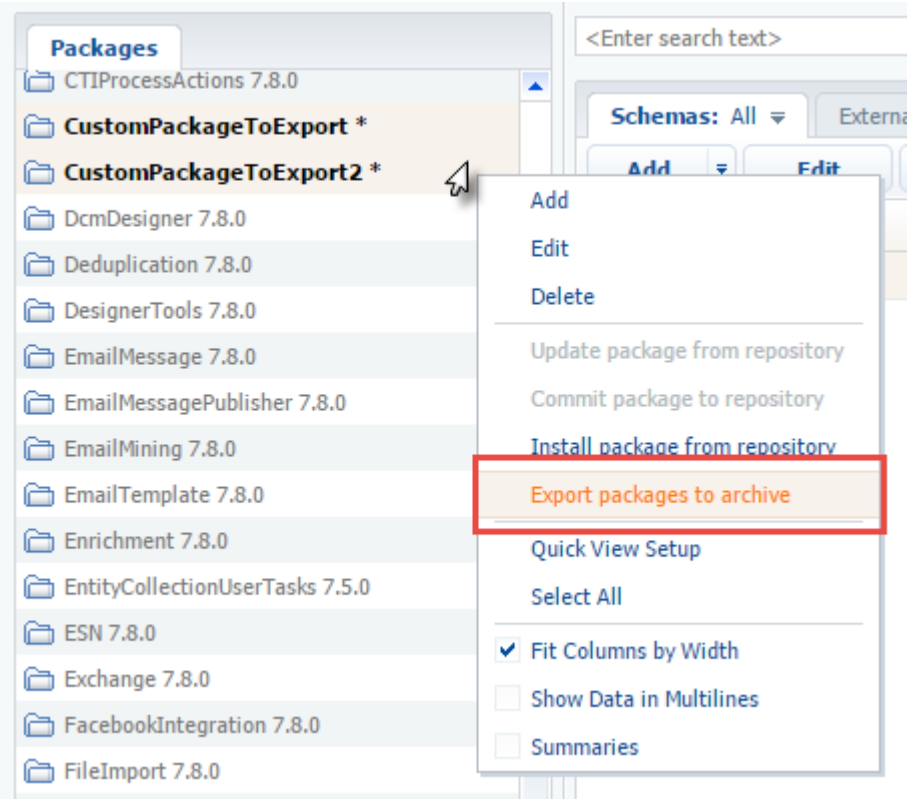
To install packages from the application interface:

1. Go to the **[Configuration] section**.

2. On the [Packages] tab, select one or multiple packages (hold Ctrl or Shift to select multiple packages).

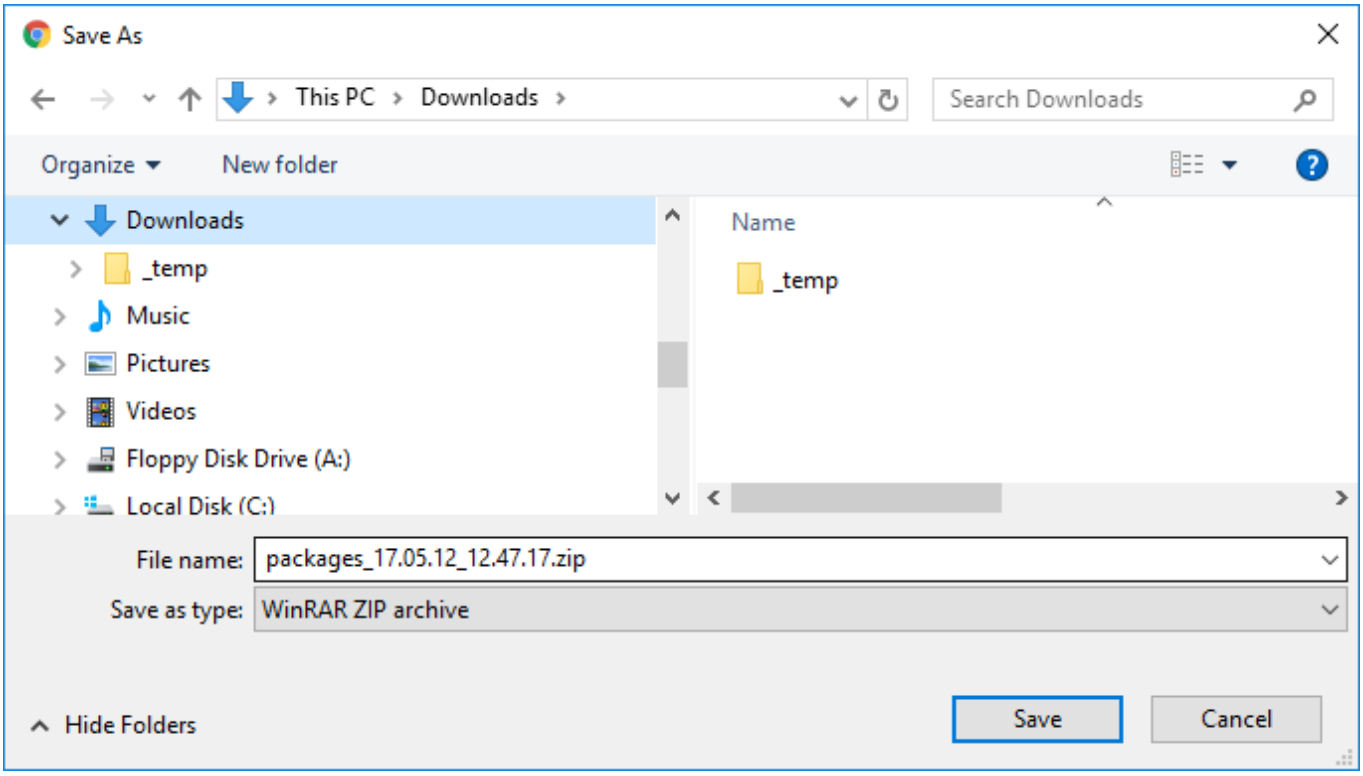
3. Trigger the [Export packages to archive] action (Fig. 1).

Fig. 1. The [Export packages to archive] action



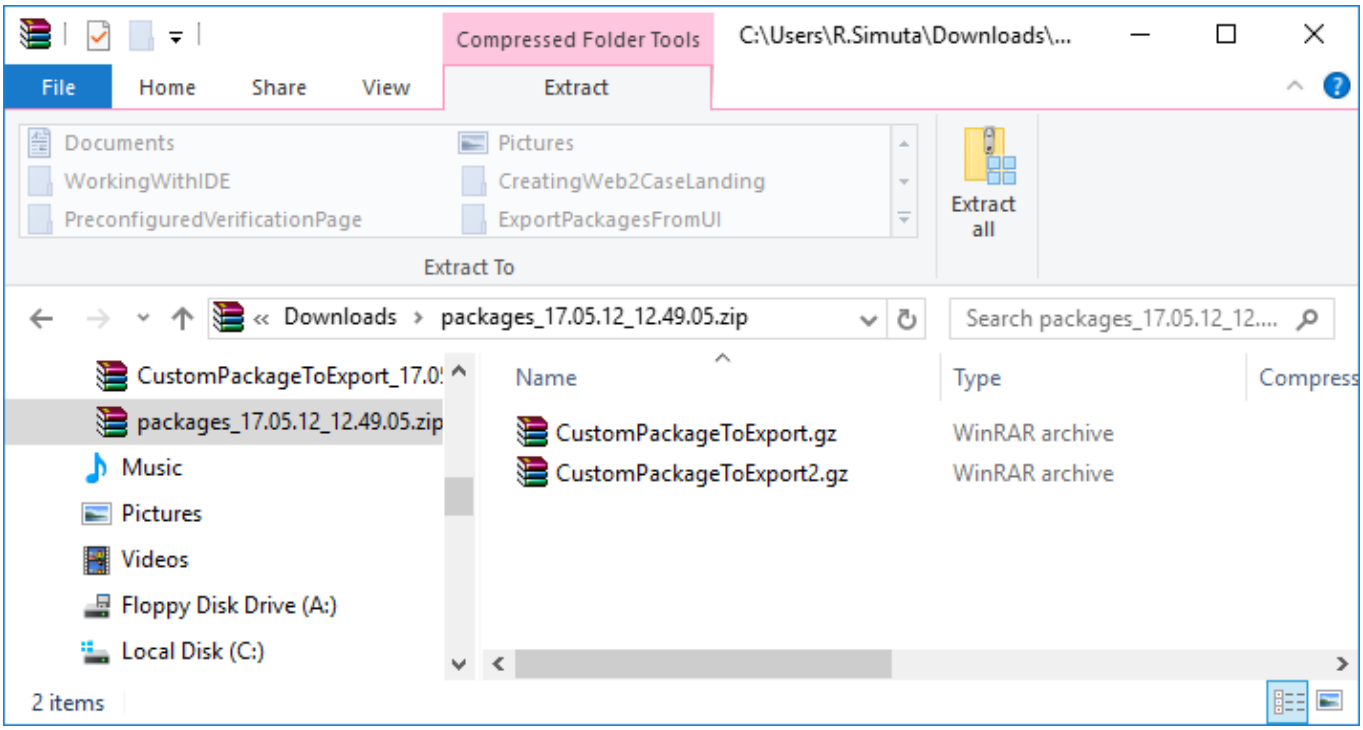
Depending on the browser settings, the zip-archive with packages will either be saved to the default downloads folder, or the browser will display a dialog box for selecting a folder for the archive (Fig. 2).

Fig. 2. A dialog box for selecting a folder for the archive



The zip-archive will contain one or multiple packages (Fig. 3) and can be imported into another bpm'online application (see "**Installing marketplace applications from a zip archive**").

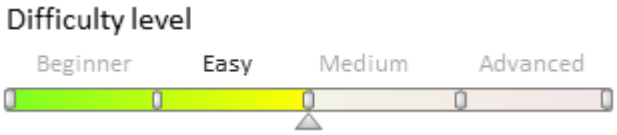
Fig. 3. A zip-archive with packages



ATTENTION

You cannot create packages in a production environment, then create a development environment on the basis of the production environment, finalize the functionality of packages, and transfer them back to the production environment. The development sequence is described in more detail in the "**Recommended development sequence**" article.

Installing marketplace applications from a zip archive



Introduction

Since bpm’online version 7.11, marketplace applications can be uploaded and installed directly in the application interface. This functionality is available in the [Installed applications] section. Installing the application directly from the marketplace is described in more detail in the “Installing applications from the marketplace” article.

To install the marketplace application from the bpm'online interface, a *.zip archive containing package archives (*.gz) is used. This archive can be **exported from the [Configuration] section**. Package archives (*.gz) can be downloaded **from the database** or from **the SVN repository** using the WorkspaceConsole utility (see: “**Working with WorkspaceConsole**”).

ATTENTION

When installing the marketplace application from *.zip-archive, the name of the application in bpm'online is formed based on the name of *.zip-archive. If you use a *.zip archive with the same set of packages but with a different name when you update this application, a new application record will be created in the [Installed applications] section.

ATTENTION

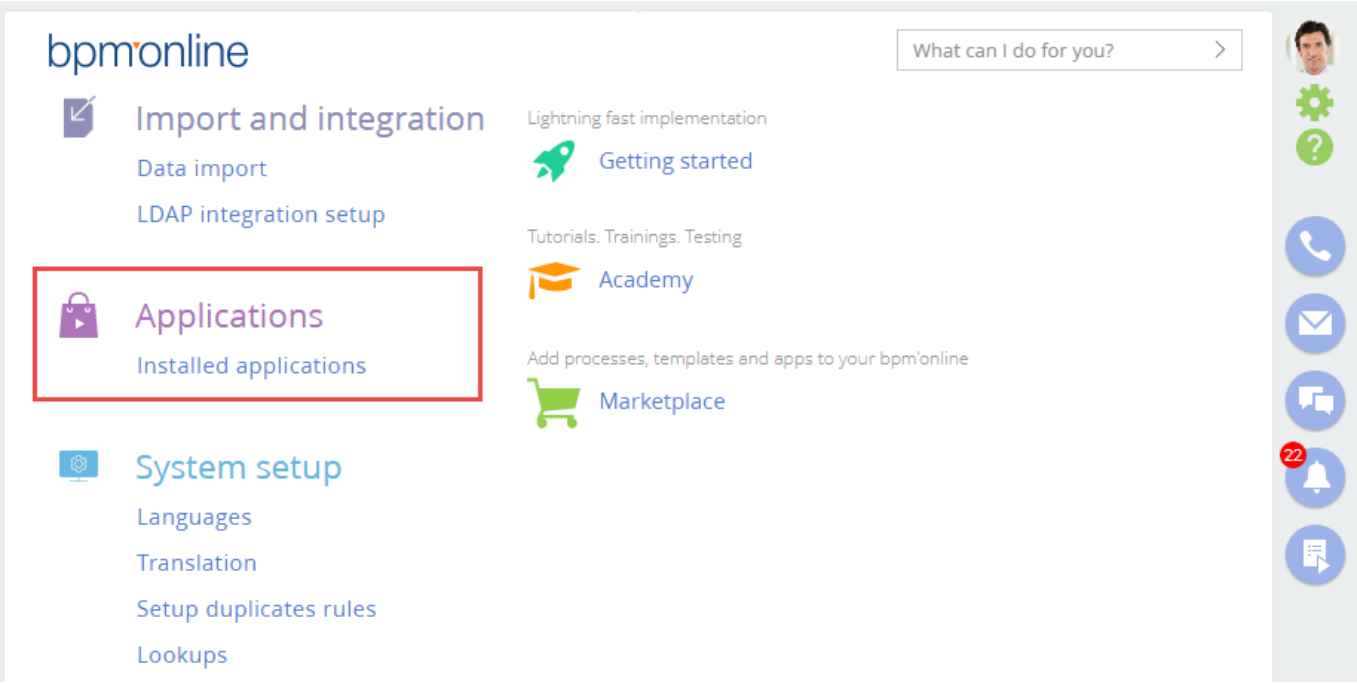
You cannot use the [Custom] package in the marketplace application. For more information about a [Custom] package, please see the “**Package [Custom]**” article.

Installing an applications from a zip archive

To install packages from the application interface:

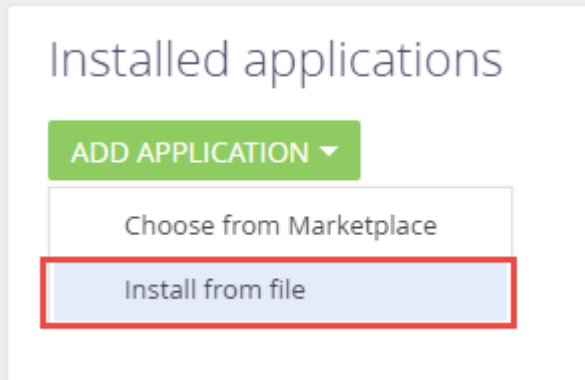
1. Go to the [System Designer] section and in the [Admin area] group, click the [Installed applications] link (Fig. 1). The [Installed applications] section will open in a separate window.

Fig. 1. System designer



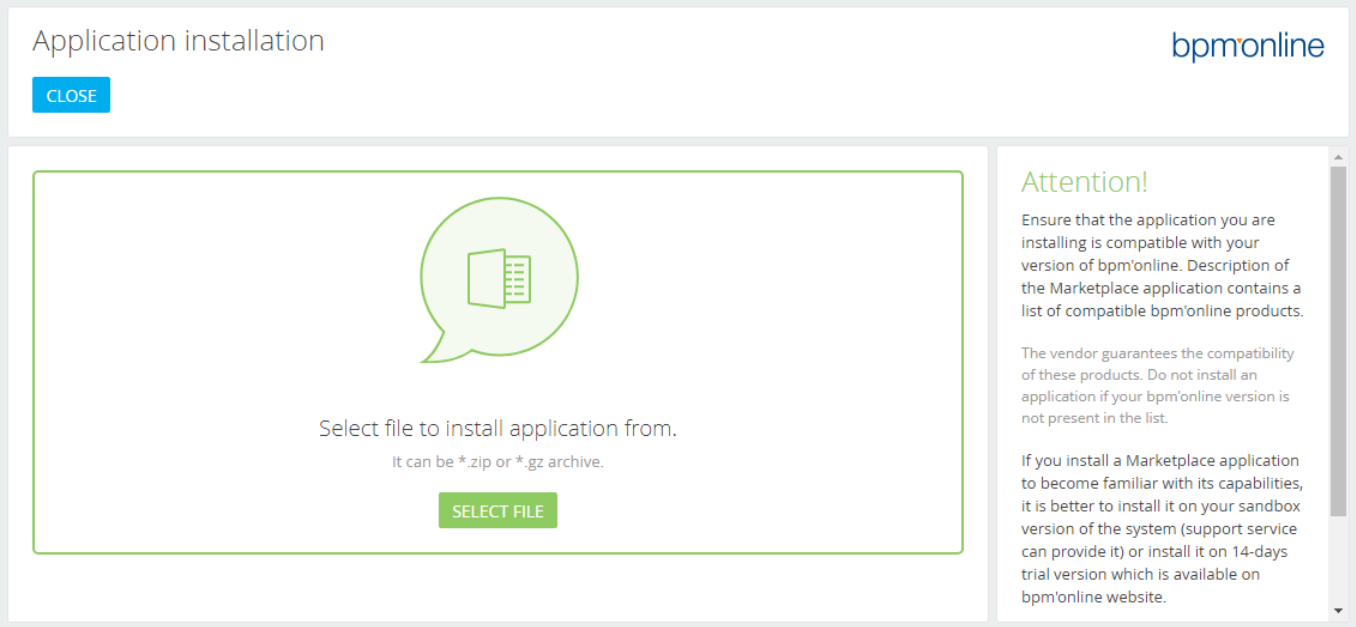
2. In the [Installed applications] section, select the [Install from file] command from the [Add application] drop-down menu (Fig. 2).

Fig. 2. The marketplace application installation menu



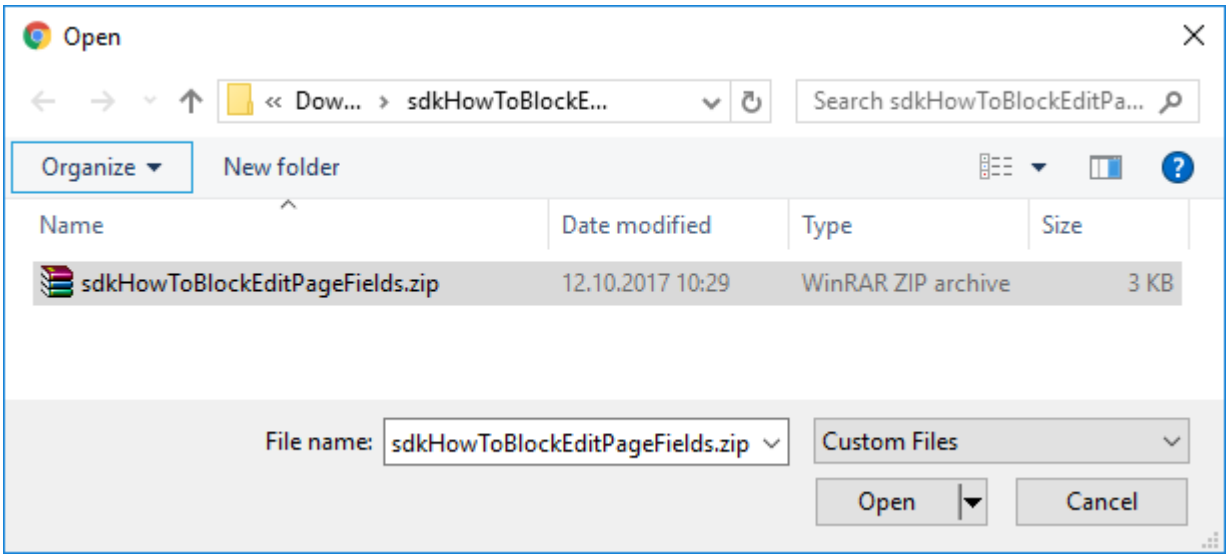
A page for installing an application will open in a separate window (Fig. 3).

Fig. 3. Application installation page



3. Click on the [Select file] button, and select the necessary * .zip archive (Fig. 4).

Fig. 4. Selecting a package for import



The system will be backed up (Fig. 5) and the application will be installed (Fig. 6).

Fig. 5. System backup

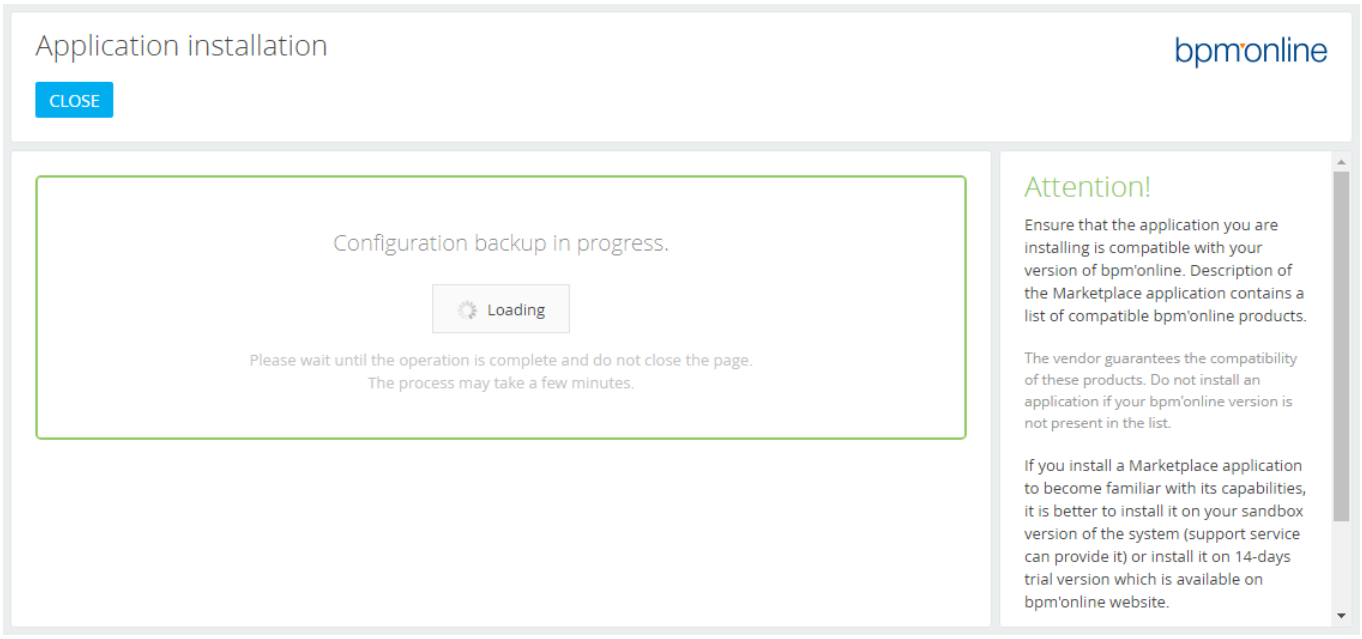
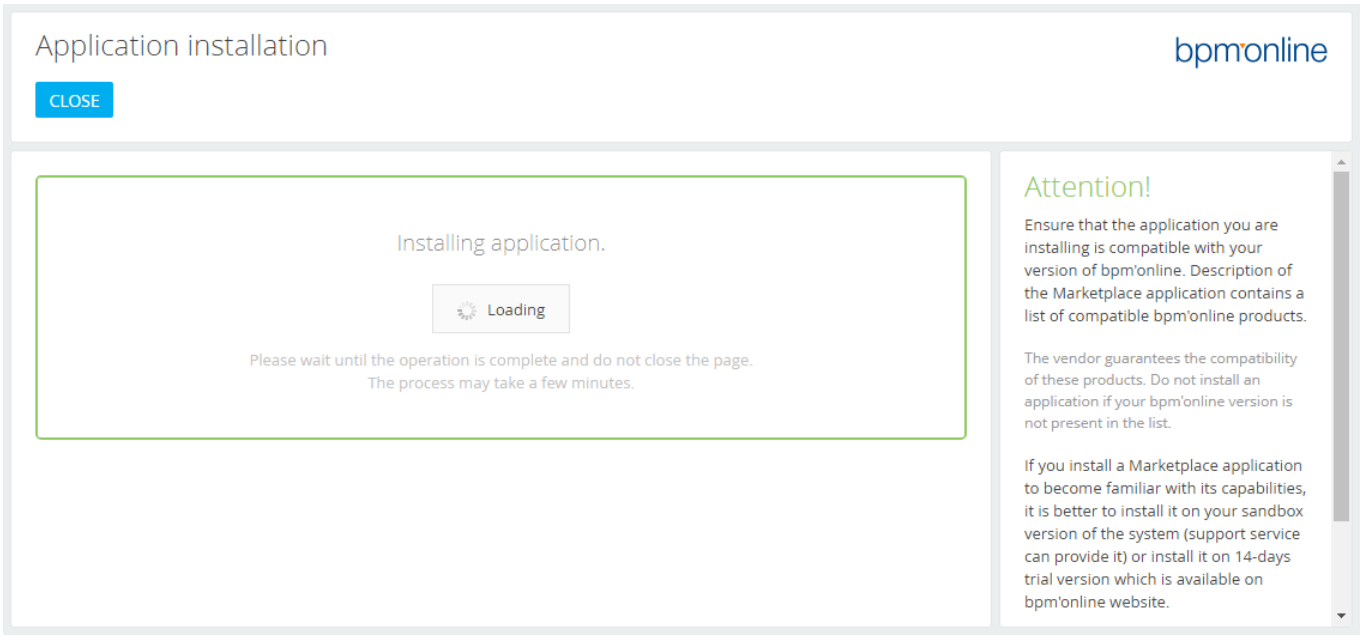


Fig. 6. Application setup



A corresponding message will be displayed (Fig. 8).

Fig. 7. Successful installation message

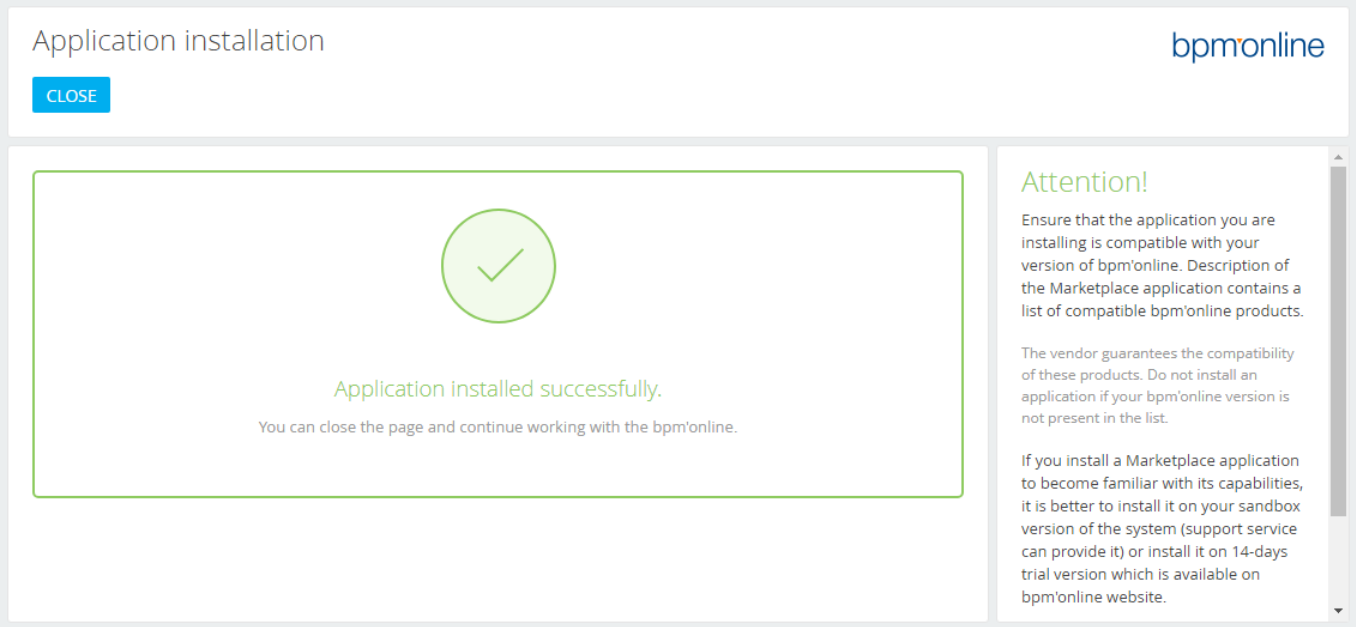
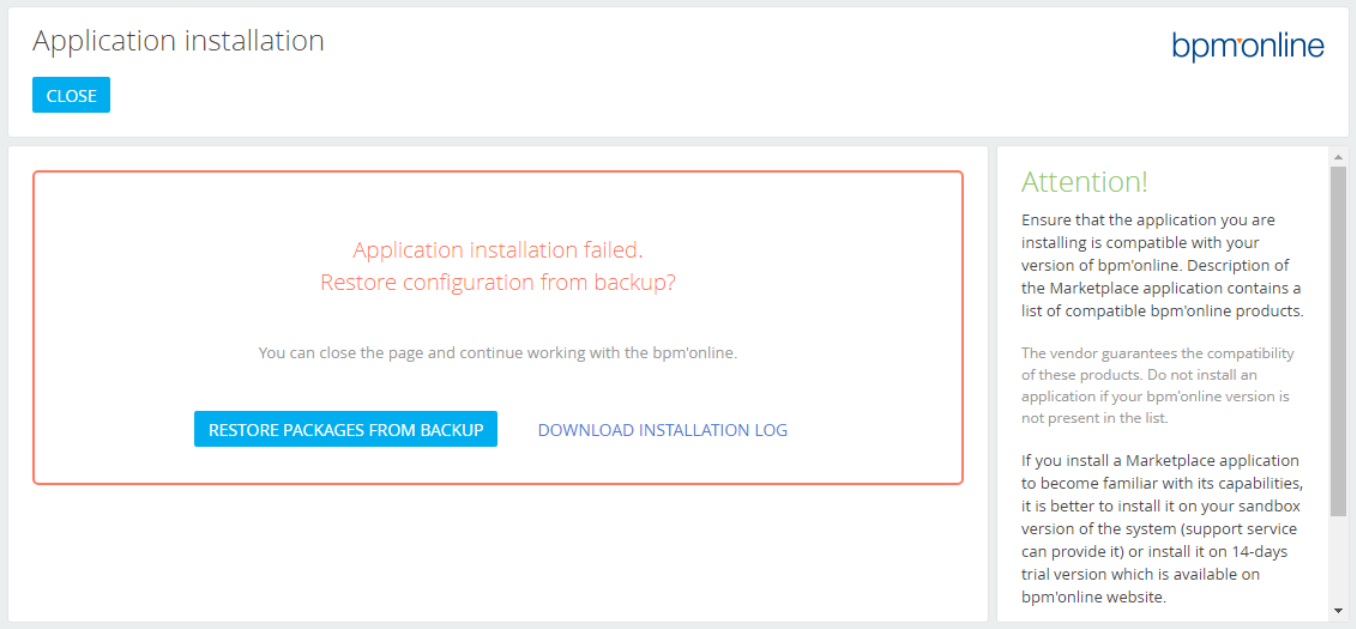


Fig. 8. Unsuccessful package installation message



After the installation is complete, you can download the log file by clicking the [Download Installation Log] button.

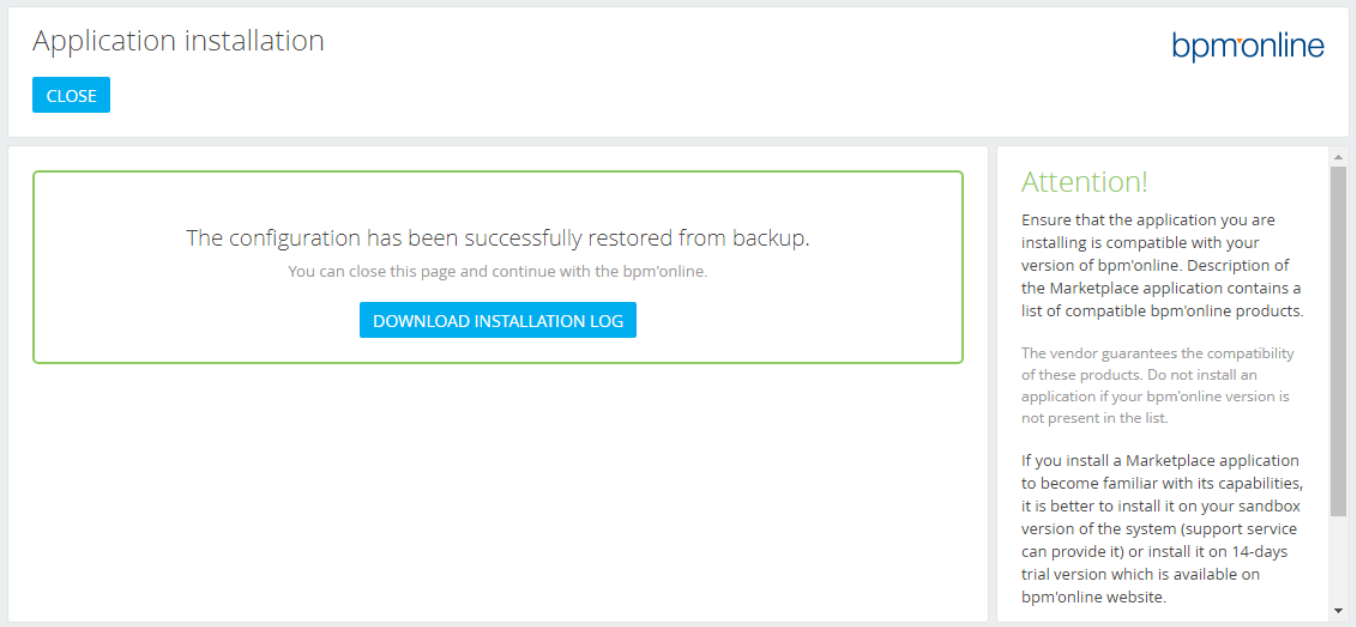
ATTENTION

You cannot create application packages in a production environment, then create a development environment on the basis of a production environment, refine the functionality of packages, and transfer them back to the production environment. For more information about development sequence, see the “**Recommended development sequence**” article.

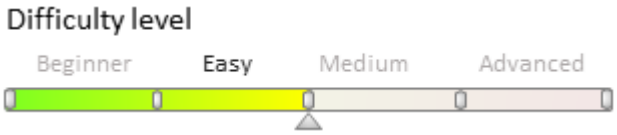
Restoring from backup

If you encounter an error during the installation process, you can restore the previous configuration by clicking the [Restore packages from backup] button (Fig. 8). After the backup is restored, a corresponding message will be displayed (Fig. 9). You can select another package file for import.

Fig. 9. Restoring a backup copy message



Transferring changes using schema export and import



Introduction

One way to transfer changes between work environments or between configurations of the same work environment (usually a development environment) is by exporting and importing schemas.

The schema export and import is used in the following cases:

1. Transferring “work-in-progress” schemas from one developer to another, as committing incomplete features to the version control system (SVN) is not a good practice.
2. Saving development results (if the SVN version control system cannot be used for this purpose).
3. Quick schema transfer between the environments.

Advantages:

Ability to replace contents of a schema quickly.

Disadvantages and limitations:

- The mechanism enables you to export and import only schemas. You cannot export or import packages. In addition, it is not possible to transfer data connected to the packages.
- Administrator access rights in the application are required.
- It is not possible to load several schemas at the same time.

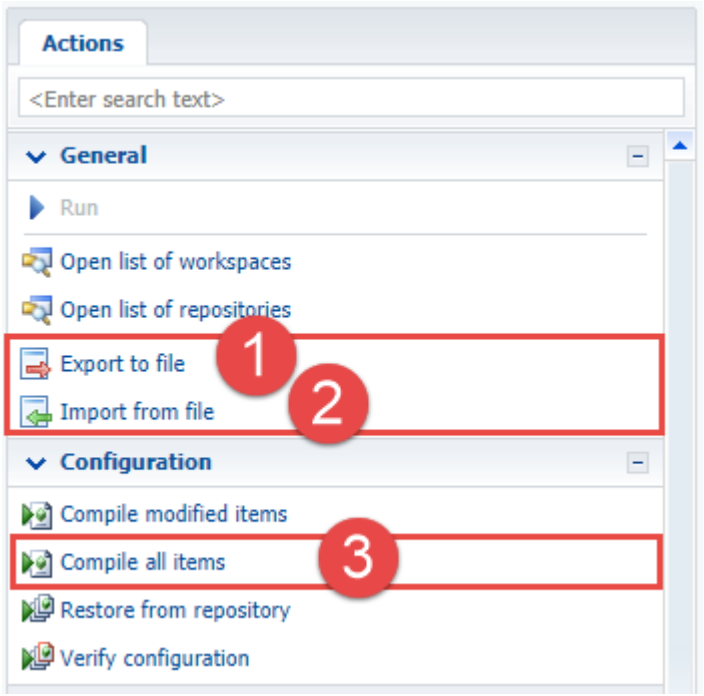
Exporting schemas

To export a schema:

1. Go to the **[Configuration] section** of the system designer.
2. Select the package in which the schema is located.
3. Select a schema to export.

4. Click the [Export to file] on the [Actions] tab (Fig. 1,1).

Fig. 1. Configuration action for client schema export and import



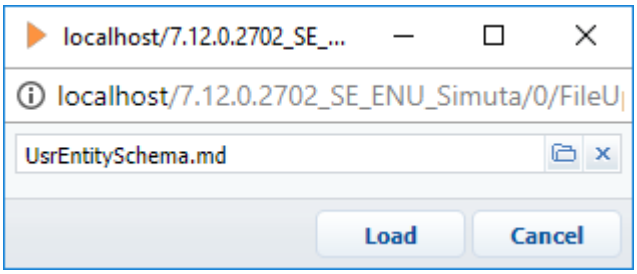
A file with the schema name and *.md extension saved on your hard drive as a result.

Importing schemas

To import a schema:

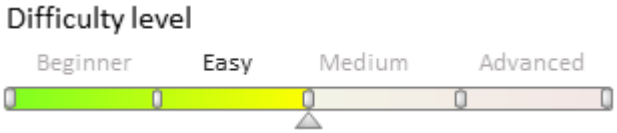
1. Go to the **[Configuration]** section of the system designer.
2. Select the package in which you want to import a schema.
3. Click the [Import from file] on the [Actions] tab (Fig. 1, 2).
4. Select the file of the previously exported schema in the dialog box (Fig. 2).
5. Compile the configuration by selecting [Compile all items] on the [Actions] tab (Figure 1, 3).

Fig. 2. Import file selection window



> **Attention!**
>
> When importing multiple schemas, you need to consider their mutual dependencies. First, you need to import all the dependency schemas, and then import the schemas that depend on them. For example, first you need to import the object schema, and then the page layout view model, which is dependent on the object schema.

Transferring changes using SVN



Introduction

The version control system is an optional component. Although bpm'online can work without it, the version control system is required in case a user driven application customization is expected. If the development is carried out by a team of developers, using SVN to transfer and merge changes becomes essential.

The purpose of version control system in bpm'online is:

- Transfer of changes between working environments, for example, between development environments.
- Storage of configuration schemas and package versions.

Bpm'online supports the Subversion (SVN) version control system 1.8 and up. Details on how to use SVN can be found in the documentation. SVN repository configuration and bpm'online integration is described in the "**Create repository in SVN server**" article.

Benefits of transferring changes via SVN

- Ability to transfer both the schemas and packages between working environments and configurations.
- Ability to transfer package data, such as lookup or section records.
- Automatic installation of SVN dependency packages.
- Autonomy from the support service in terms of transferring changes in the cloud.

The recommended steps for transferring changes via SVN

ATTENTION

The information below are applicable when working with SVN repositories via the **bpmonline built-in development tools**. The information are not applicable when the file system design mode is turned on (see "**Working with SVN in the file system**"). See "**Working with SVN in the file system**" for more information.

1. Verify that the application to which you want to transfer changes is configured to work with SVN.

For more details about the application setup for working with the version control system, please see the "**Create repository in SVN server**" article.

2. Enable mechanisms for automatic application of changes.

To apply the necessary changes after the transfer, enable mechanisms for automatic application of changes. To do this, you need to set the following keys of the *appSettings* element in the Web.config file (located in the Terrasoft.WebApp directory) to *true*:

```
<add key="AutoUpdateOnCommit" value="true" />
<add key="AutoUpdateDBStructure" value="true" />
<add key="AutoInstallSqlScript" value="true" />
<add key="AutoInstallPackageData" value="true" />
```

The *AutoUpdateOnCommit* key is responsible for automatically updating packages from the SVN before they are committed to the repository. If this key is set to *false*, then, before the commit operation can be run, the application will notify the user about the need to update the local copy from SVN if package schemas have been modified. The *AutoUpdateDBStructure*, *AutoInstallSqlScript* and *AutoInstallPackageData* keys are responsible for automatically updating the database structure, installing SQL scripts, and the data bound to package.

3. Make sure that all necessary data are bound to package.

You need to make sure that all the data you need to migrate is bound to the corresponding package before the package transfer. These data are represented by lookup and section records.

NOTE

If a section wizard was used when creating sections, then certain data is automatically connected to the current package.

4. Make sure that all dependencies of the package can be transferred.

Dependencies on other packages can be added to the custom package during the development process. If the dependency packages are developed by third-party developers, you need to make sure that they are already installed in the application into which the user package will be transferred. If the dependency packages are in an accessible SVN repository, they will be installed automatically if necessary.

5. Install the package from the repository

The sequence of package installation is described in detail in the "**Installing packages from repository**".

Important!

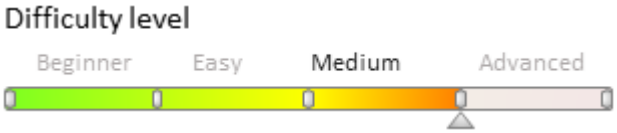
Before updating the application via SVN, you must back up the database. If the application is deployed in the cloud, you should contact support.

Please note that you cannot revert to the previous version of the application via SVN.

ATTENTION

Starting with version 7.11, after installing or updating a package from SVN, bpm'online application requires compilation (the [Compile all items] action in the [Configuration] section). In the process of compilation, the static content will be generated (see "**Client static content in the file system**" article).

Transferring changes using WorkspaceConsole



Introduction

The WorkspaceConsole utility is designed to work with bpm'online packages. Use the utility to:

- Export packages from development environments and migrate them to test environments or production environments (the packages are saved as archives).
- Install new packages when upgrading or migrating from development environments.
- Export and import localization resources of schemas.
- Create and migrate between workspace applications.
- Work with configuration schemas.

Attention!

Bpm'online production environments that actively run business processes should not be updated during business hours. Updating production environments during business hours may cause loss of data.

First, you must set up the WorkspaceConsole before start working with it. The setup procedure is described in the “**WorkspaceConsole settings**” article.

NOTE

WorkspaceConsole needs access to the bpm'online database. If the application is deployed in the cloud, then only employees of the cloud services department are able to work with the WorkspaceConsole utility. Please contact the support if you need to transfer changes to a cloud application.

Benefits of transferring changes via the WorkspaceConsole

- Ability to transfer both the schemas and packages between working environments and configurations.
- Ability to transfer package data, such as lookup or section records.
- Ability to work separately with localization resources, bound data and SQL scripts.

The recommended steps for transferring changes via WorkspaceConsole

1. Check the data binding.

Be sure to check data binding before exporting packages. These data include: system settings, lookups, section filling, etc.

Attention!

If the section was created via the section wizard, then the data necessary for its work are bound to the corresponding package automatically. However, for the section to be displayed in the workplace after import, you must bind the corresponding value of the *SysModuleInWorkplace* object.

2. Backup the application database where you want to transfer the changes

You need to back up the database before updating the application with the WorkspaceConsole. If you use WorkspaceConsole commands incorrectly, the data could be damaged or lost.

3. Export the packages from the application database that contains the changes to transfer

To export the packages from the database, you need to run the WorkspaceConsole with the following parameters (table 1):

Table 1. WorkspaceConsole parameters for exporting database packages

Parameter	Value	Description
operation	*SaveDBContent*	Saves the contents of database to the file system. The content type is determined by the *contentTypes* parameter value. The *destinationPath* parameter determines where to export the contents in the file system.
contentTypes	*Repository*	The database content type to be exported. The *Repository* value specifies that a workspace (with the name specified in the *workspaceName* parameter) must be exported to a specific directory (specified by the *destinationPath* parameter).
workspaceName	*[Workspace name]*	Workspace (configuraion) name where uploaded packages are defined. All users work in the *Default* workspace by default.
destinationPath	*[Path to local directory]*	A path to local directory in the file system. The packages archived in **.gz* format will be will be exported to this directory.

An example of a database export command for the Windows command prompt:

```
[Path to WorkspaceConsole]\Terrasoft.Tools.WorkspaceConsole.exe -
operation=SaveDBContent -contentTypes=Repository -workspaceName=[Workspace name] -
destinationPath=[Path to local directory]
```

After you run this command, the archives of all packages from the specified workspace will be exported into the local directory.

NOTE

Use text editor (such as Windows notepad) to create a batch file (* .bat) with the necessary command in it.

4. Import the packages to the application where the changes must be transferred

To import the database packages, run the *WorkspaceConsole* with the following parameters (table 2):

Table 2. *WorkspaceConsole* parameters for database packages load

Parameter	Value	Description
operation	*InstallFromRepository*	Imports the contents of packages from archives to the database. The bound SQL-scripts, source code generation and installation of bound data are performed if needed. Operation works only with new or modified packages and their elements.
packageName	*[Package name]*	Name of a package from the configuration specified in *workspaceName* parameter. Note that all the packages that the specified package depends on will be installed as well. Optional parameter. If it is not specified, then all packages from the configuration will be imported.
workspaceName	*[Workspace name]*	Workspace (configuraion) name where installed packages are defined. All users work in the *Default* workspace by default.
sourcePath	*[Path to local directory]*	A path to a local directory in the file system. The directory must contain the package archives for importing (*.*gz* files).
destinationPath	*[Path to local directory]*	A path to local directory in the file system. Packages from the directory determined in *sourcePath* parameter will be unpacked here.
skipConstraints	*false*	The option to skip creating foreign keys in the database tables. Can be *true* or *false*.
skipValidateActions	*true*	The option to skip checking for possibility of the creating table indexes during the database structure update. Can be *true* or *false*.
regenerateSchemaSources	*true*	Indicates the need to regenerate the source code after saving the packages in the database. Can be *true* or *false*.
updateDBStructure	*true*	Indicates the need of modify the database structure after the package installation. Can be *true* or *false*.
updateSystemDBStructure	*true*	Indicates the need to modify the structure of the system schema database before the package installation. Also creates all missing indexes in the system tables. Can be *True* or *false*.
installPackageSqlScript	*true*	Indicates the need to run SQL scripts before and after package installation. Can be *True* or *false*.
installPackageData	*true*	Indicates the need to install the data bound to the

		packages after the package installation. Can be *True* or *false*.
continueIfError	*true*	Indicates the need of interrupt or resume the installation on the first error. If this parameter is set to true, the installation process will continue, even if errors are detected. The user will receive the list of errors after the installation. Can be *True* or *false*.
logPath	*[Path to local directory]*	A path to the directory where the log of the completed operation will be created. The file name consists of the date and time of the operation launch.

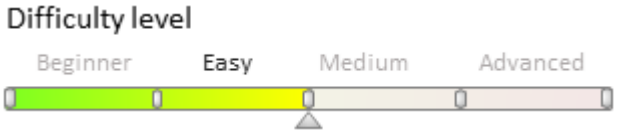
An example of a package installation command for the Windows command prompt:

```
[Path to WorkspaceConsole] -packageName=UsrCustomAuto -workspaceName=Default -
operation=InstallFromRepository -sourcePath=[Path to package archives] -
destinationPath=[Package extraction path] -skipConstraints=false -
skipValidateActions=true -regenerateSchemaSources=true -updateDBStructure=true -
updateSystemDBStructure=true -installPackageSqlScript=true -installPackageData=true -
continueIfError=true -logPath=[Path to the logs]
```

5. Restart the application in IIS

The modifications are made by the *WorkspaceConsole* utility directly into the database and therefore are not available for any application that is already running. Restart the application in IIS for the changes to take effect.

Creating a custom client module schema



Introduction

Client Modules are separate functional blocks, downloaded and run on demand in accordance with the AMD technology. System functions are implemented via client modules. All client modules in bpm’online share description structures that correspond with AMD module description format. Custom module types are described in the "**Client Modules**” article.

The following client module types are available in bpm’online:

- non-visual modules (module schema)
- visual modules (view module schema)
- expanding modules and replacing client modules (Replacing schema of the custom model).

The procedure for creating a custom schema differs for various types of schemas.

Creating a new schema of the non-visual module

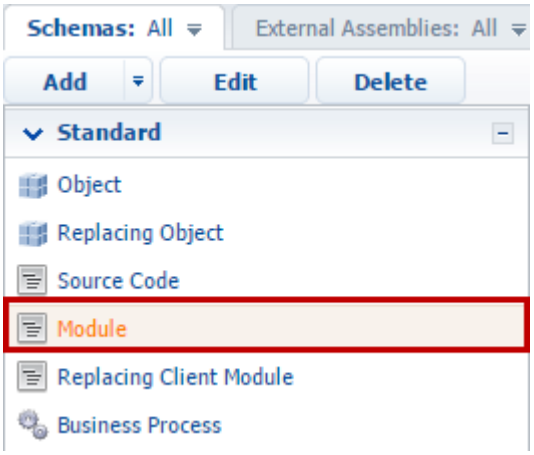
Non-visual modules represent system functionality that is not associated with data binding or data display in the UI. Examples of non-visual modules in the system are business rule modules (BuisnessRuleModule) and utility modules that implement utility functions.

To create a non-visual module schema:

1. Go to the **[Configuration] section** and select custom package to add new schema.

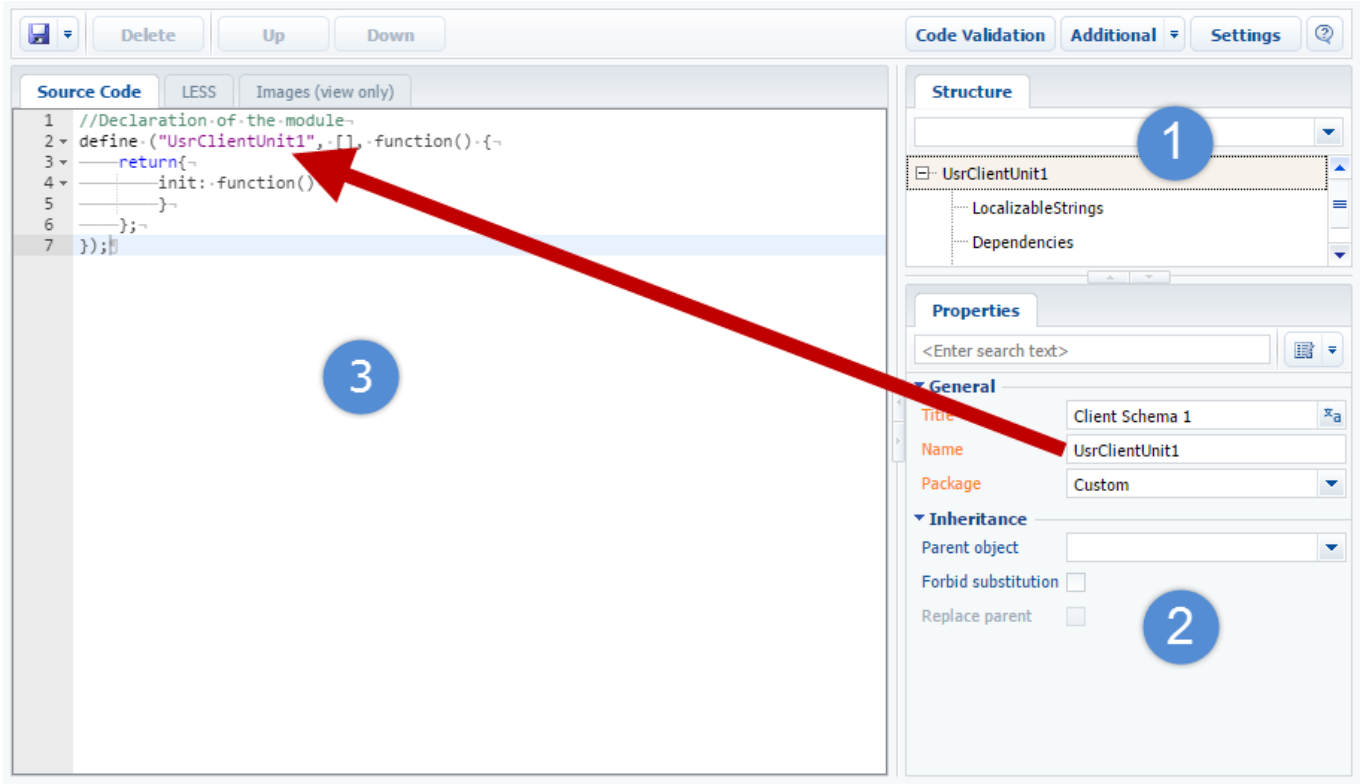
2. On the [Schemas] tab, select [Add] — [Module] (Fig. 1).

Fig. 1. Adding new module schema



3. Select the root element of the structure in the **custom module designer** (Fig. 2, 1) and fill out the properties of the generated schema module (2):

Fig. 2. Client module designer



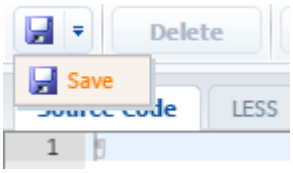
The main properties of the module schema are:

- [Name] – schema name. May contain only Latin characters and numbers. Contains the prefix specified in the [Object name prefix] system setting (SchemaNamePrefix).
- [Title] – schema title. Can be localized.
- [Package] – custom package where the schema is generated.

4. Add the module source code to the [Source code] tab (Fig. 2.3). You need to make sure that the module name in the *define()* function is the same as the module schema name.

5. Save the module schema after making all the changes (Fig. 3):

Fig. 3. Saving the client module schema



Creating a new view model schema.

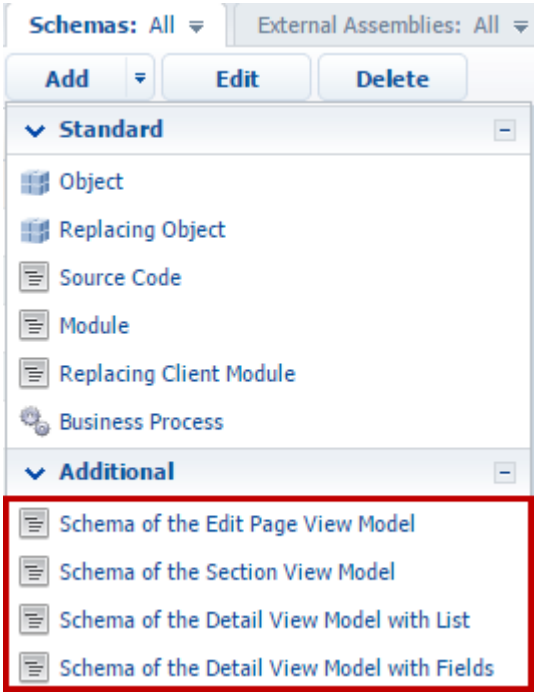
Custom view model schema is a visual module schema. It is a configuration object for generating views and view models by the *ViewGenerator* and *ViewModelGenerator* generators. For more information about the schema structure, please see the "**Client view model schemas**" article.

To create a schema of the visual module:

1. Go to the **[Configuration] section** and select custom package to add a new schema.

2. Select one of the commands to add the schema from the "Additional" commands menu on the [Schemas] tab (Fig. 4):

Fig. 4. Commands for adding view model schemas

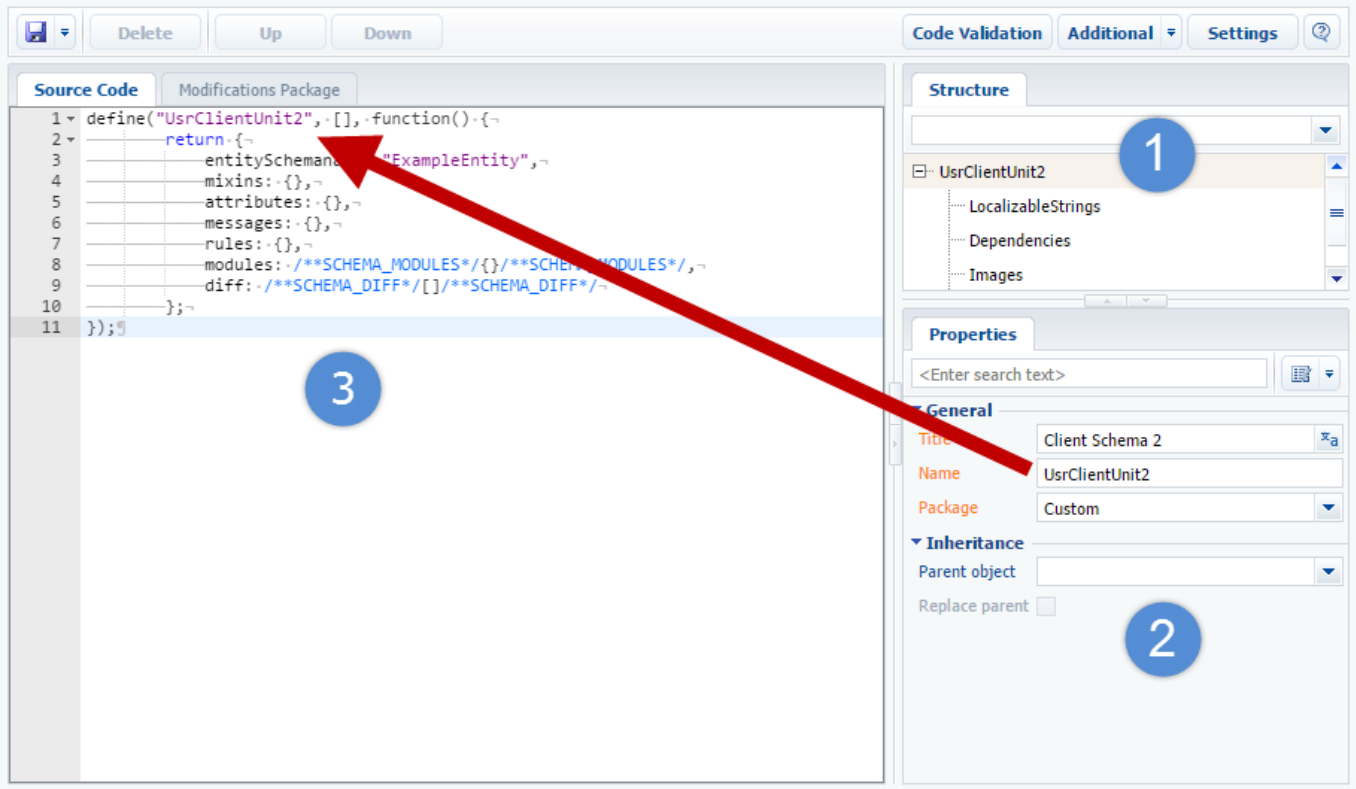


You can add the following types of visual module schemas:

- [Schema of the Edit Page View Model] – a schema of the edit page of section record.
- [Schema of the Section View Model] – a schema of the section page with the list and dashboards.
- [Schema of the Detail View Model with List] – a schema of the edit page of detail with list.
- [Schema of the Detail View Model with Fields] – a schema of the edit page of detail with fields.

3. Select the root element of the structure in the **designer of custom schema model** (Fig. 5.1) and fill the properties of the generated schema (2):

(Fig. 5). custom view model schemas designer



The main properties of the view model schema match with the main properties of the non-visual module schema, mentioned above.

4. Add the model schema source code to the [Source code] tab (Fig. 5.3). You need to make sure that the visual module name in the *define()* function is the same as the view model schema name.

Save the module schema after making all the changes (Fig. 3).

Creating a replacing schema

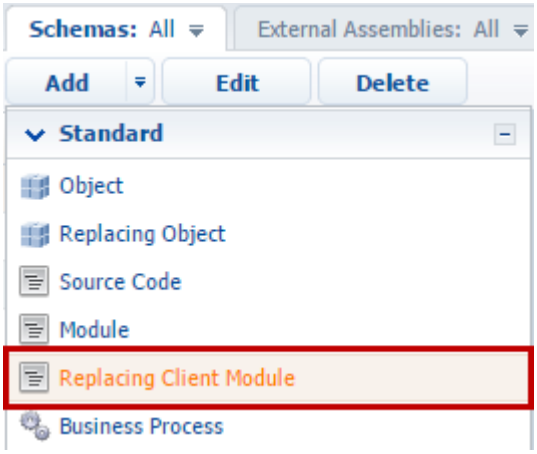
Replacing schemas are used to extend the functions of the already existing schemas. The existing schemas also may be replacing and belong to different packages.

To create the replacing schema of the non-visual or visual modules:

1. Go to the **[Configuration] section** and select custom package to add new module schema.

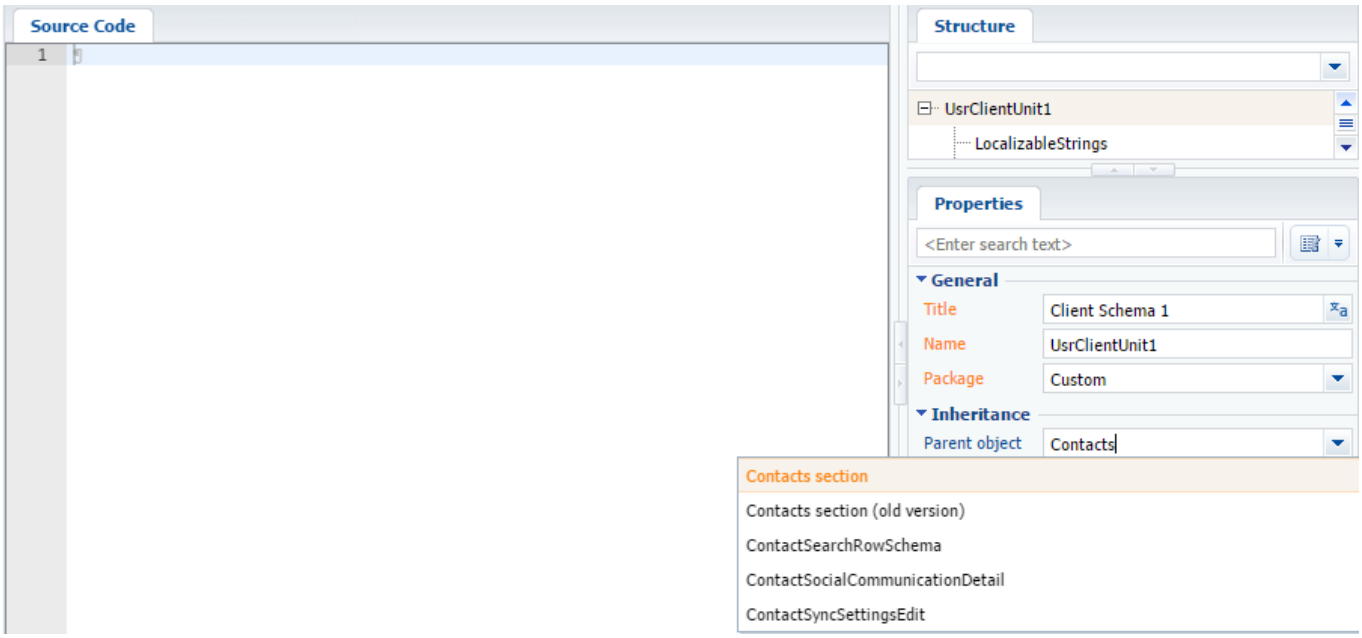
2. On the [Schemas] tab, select [Add] — [Replacing client module] (Fig. 6).

Fig. 6. Creating a replacing schema



3. Select the root element of the structure in the **custom module designer** (Fig. 7.1).

(Fig. 7). Creating a replacing schema



4. To make the replacing module for a specific section or page, specify the title of the base view model schema that you want to replace in the [Parent object] field of the schema properties. For example, to create a replacing schema for the [Contacts] section you need to specify the *ContactSectionV2* as a parent object. Start typing the “Contact Section” schema title in the [Parent object] field of the replacing schema properties and select the corresponding value from the drop-down list.

After confirmation of the selected parent object (Fig. 8), the other property fields are filled out automatically (Fig. 9, 1).

Fig. 8. The confirmation dialog for using the parent schema

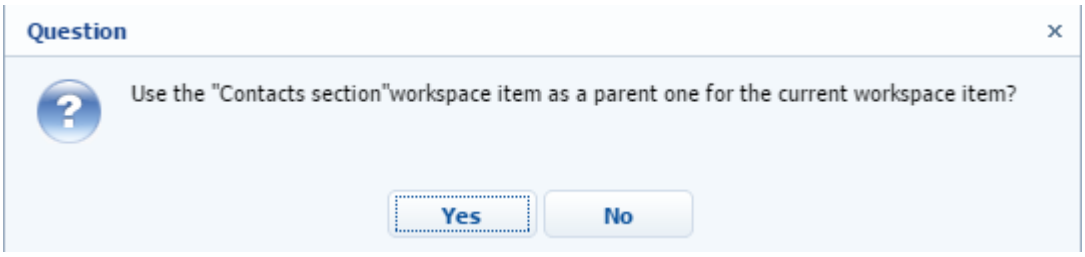
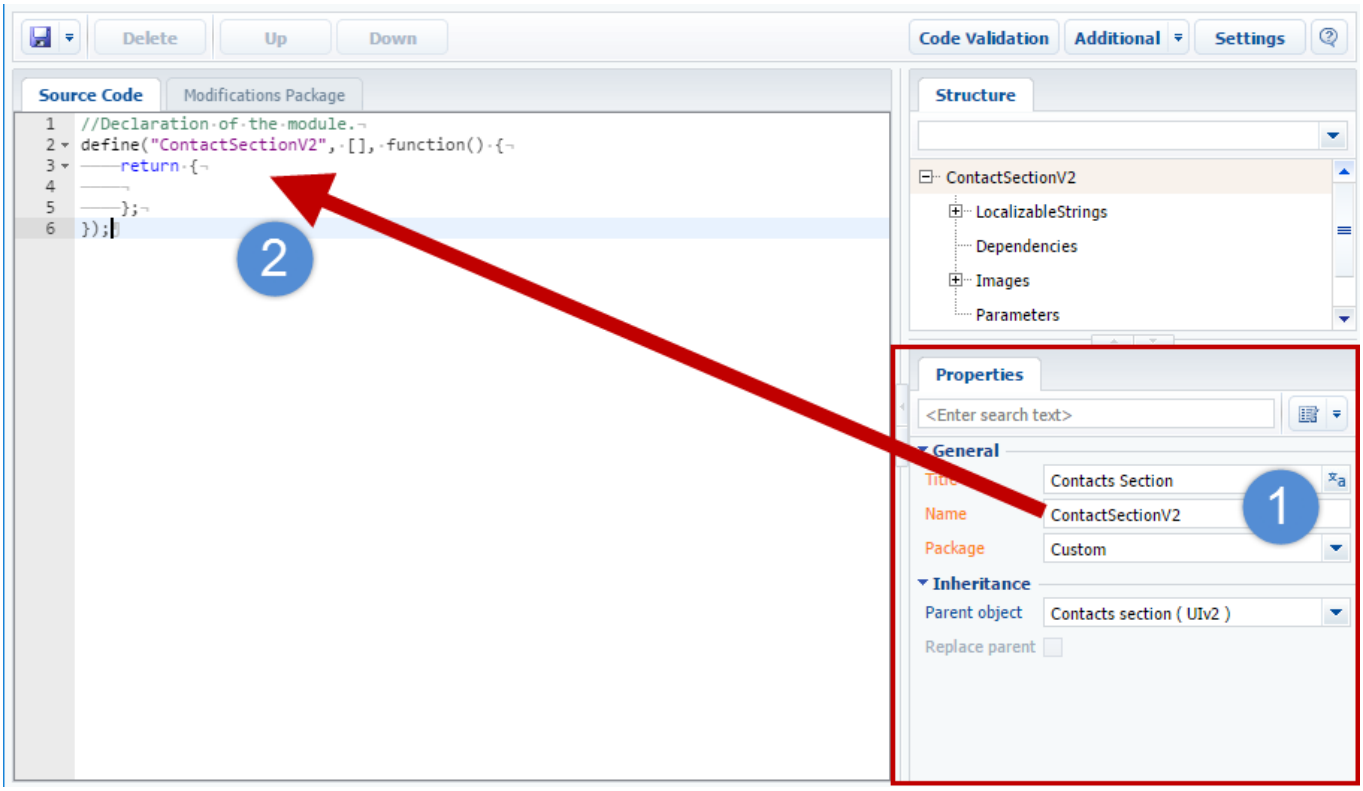


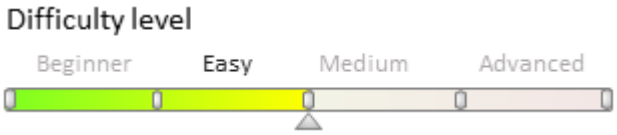
Fig. 9. A replacement client schema in the client schema designer



5. Add the model schema source code to the [Source code] tab (Fig. 9.2). You need to make sure that the replacing module name in the *define()* function is the same as the view model schema name.

6. Save the module schema after making all the changes (Fig. 3).

Creating the entity schema



Introduction

The ORM (Object-relational mapping) objects in bpm'online are based on bpm'online objects — entities. An entity is a business model that allows you to declare a new class of ORM model on the server core level. At the base level, creating an entity means creating a table with the same name and with the same columns as the created object. In most cases, each entity is a system representation of a table in the database.

Bpm'online configuration is based around schemas. Every type of the configuration item is represented by a schema of the appropriate type. From the implementation view point, any type of schema is a kernel class that is inherited from the base *Schema* class. More information about schemas and their properties can be found in the "**Packages, schemas, modules**" article.

An entity as a configuration element is presented by entity schema that is implemented by the *EntitySchema* class. The composition of the columns indexes and methods is described in entity schemas.

There are base and custom entity schemas.

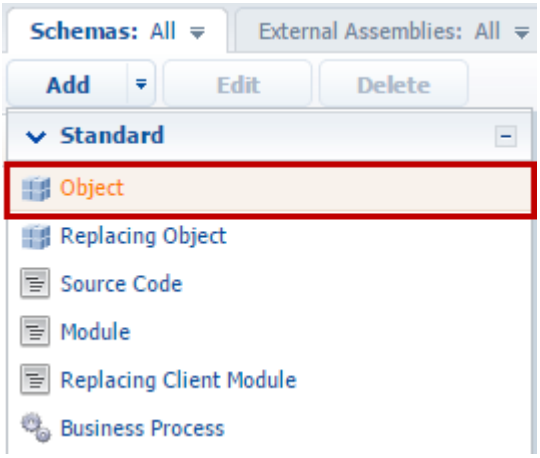
Base entity schemas are not available for editing and are located in the pre-installed packages.

Custom entity schemas can be created as part of configuration and placed in custom packages. Base schemas can be replaced by custom schemas.

Creating a custom object schema

To create a custom schema, open the **[Configuration]** section, select a custom package, go to the [Schemas] tab and select [Add] > [Object] command (Fig.1). As a result, the **Object designer** window will open, where you can configure the created entity.

Fig. 1. Creating a new object schema



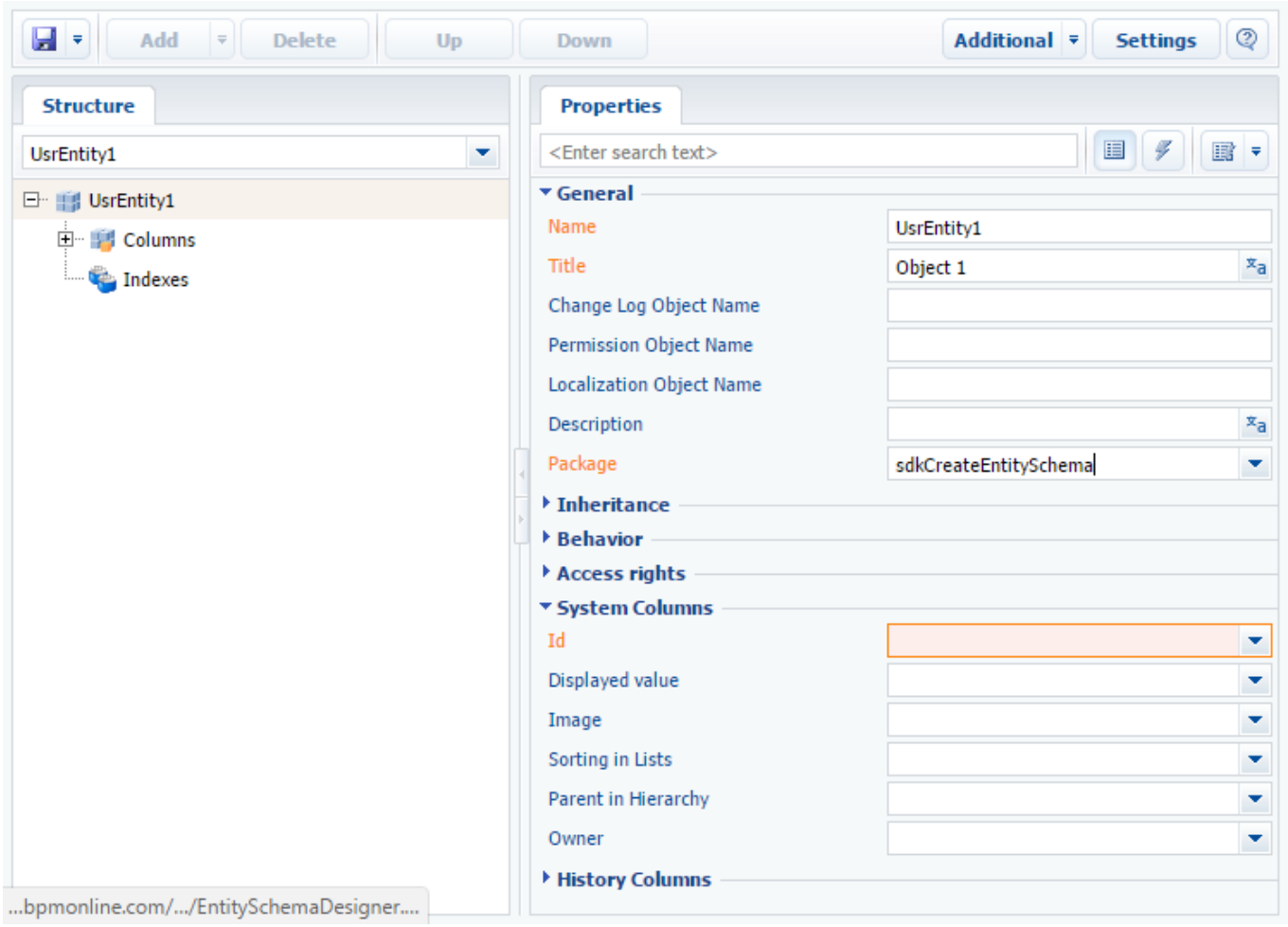
You need to assign values to the following required properties for the created entity schema (Fig. 2):

- [Title] – localized schema title. The default value is set by object designer and can be modified.
- [Name] – schema name. The default value is set by object designer and can be modified. Contains the prefix specified in the [Prefix for object name] system setting (*SchemaNamePrefix*). By default the prefix is “*Usr*”.

> ⚠ Attention!
>
> Prior to 7.9.1 version, the maximum allowed length of the entity name was 30 characters. Starting with version 7.9.1, the maximum length of the entity name is 128 characters.
>
> Entities with names longer than 30 characters can not be used on Oracle databases that are earlier than version 12.2.

- [Package] – a package where the entity schema will be placed after the publication. By default, it contains a package name that was selected before the schema creation. Select one of the available packages from the drop-down list.
- [Id] – a system column used as a primary key in the database table. It is displayed in the extended view of the "**Workspace of the Object Designer**”.

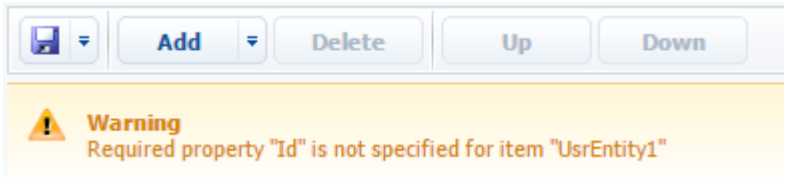
Fig. 2. Required entity schema properties in the object designer



> **NOTE**
>
> To display all schema properties in the object designer, select the [All] option in the display menu of the “**Workspace of the Object Designer**” object properties.

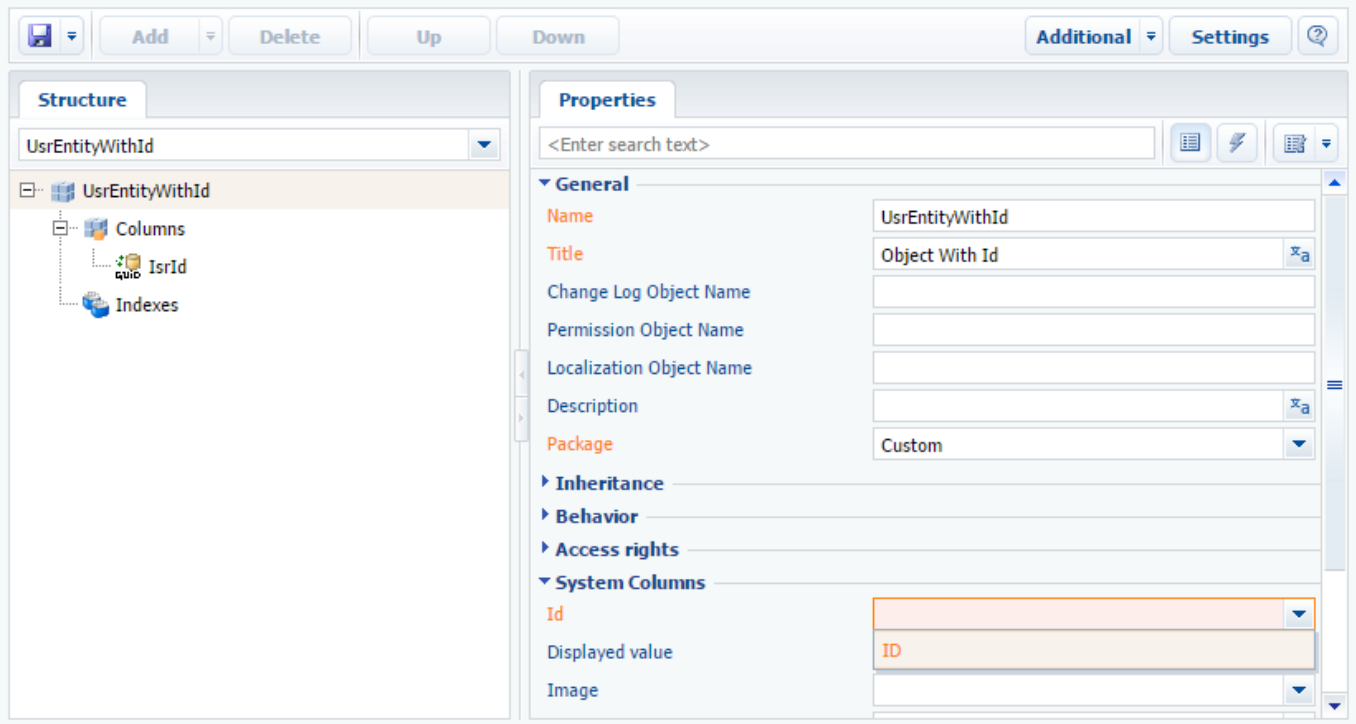
The entity is the representation of the table record in the database and it must have the id column used as a table primary key. If you try to save an object schema without an Id, a warning will pop up (Fig. 3).

Fig. 3. Empty [Id] property warning



You can set the [Id] property value by selecting the column of specific type from the drop-down list (Fig. 4) or by specifying one of base system object as a parent object.

Fig. 4. Setting the Id column



NOTE

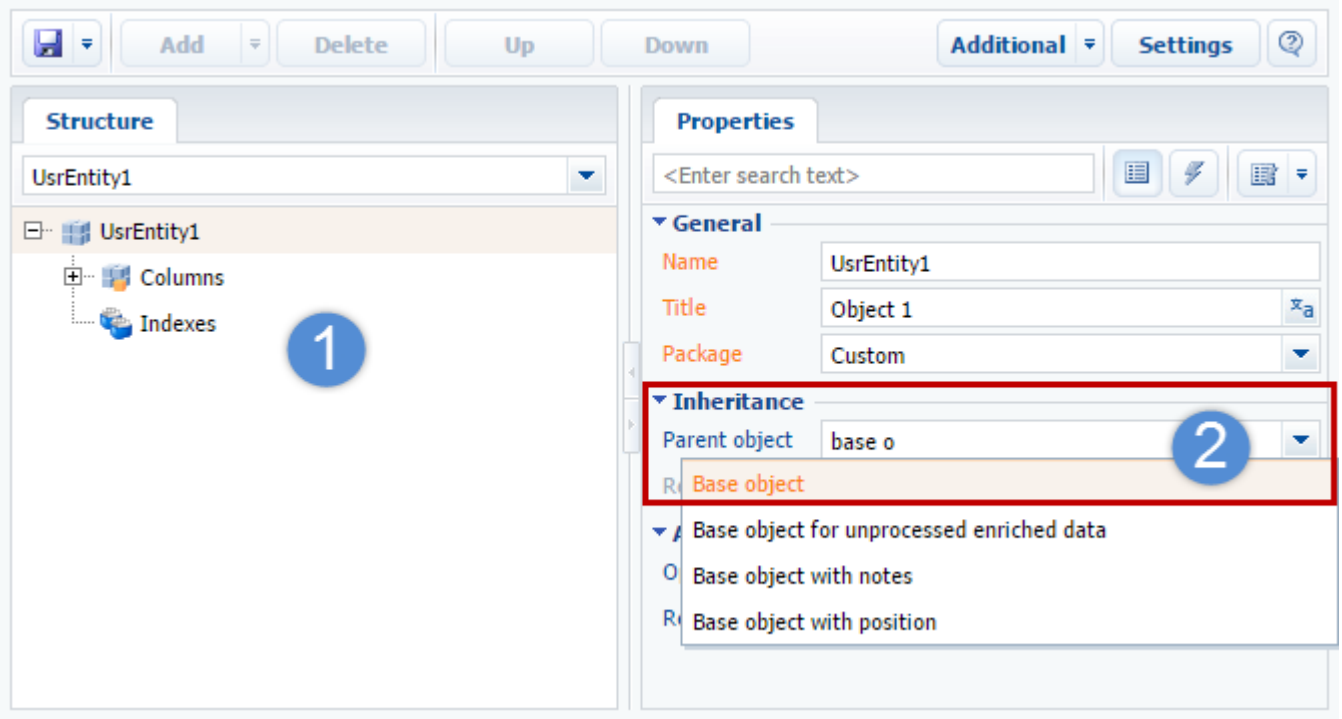
The procedure for adding the column is covered in the “Adding the custom column to the entity” section.

Specifying the parent object

The inheritance mechanism is implemented for bpm'online objects. It is used when the created entity schema must have the functionality already implemented in one of the existing entities. The [Base object] and the [Base lookup] system objects are used as parent objects in most cases.

To implement the inheritance of a new entity schema from an existing entity schema you need to select the root element of the data structure in the entity schema (Fig. 5.1). In the [Parent object] field of the schema properties, select the base entity schema whose functionality you need to inherit. To inherit the functionality of the [Base object] schema, start typing the schema title in the [Parent object] field and select the schema from the drop-down list.

Fig. 5. Selecting the parent schema



After confirming the selected parent object (Fig. 6), the columns inherited from that object will be added to the entity structure.

Fig. 6. The confirmation dialog for using the parent object

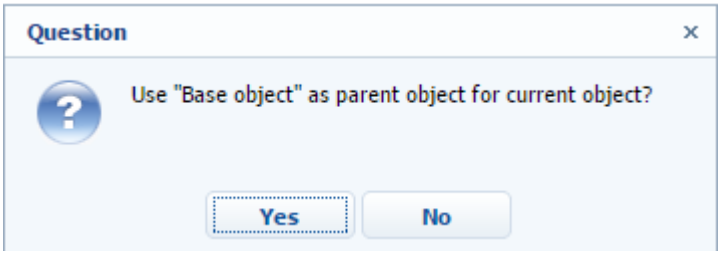
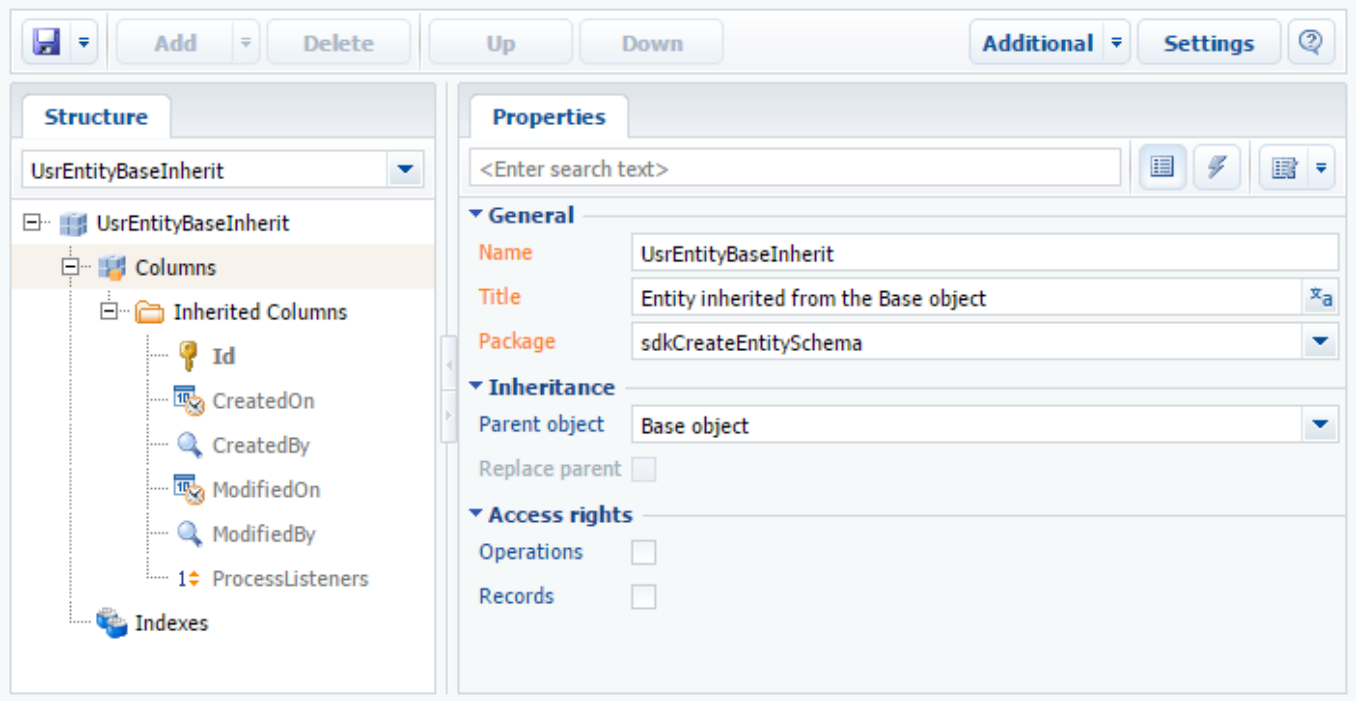


Fig. 7. Columns inherited in the entity structure



NOTE

Select the [Show system columns] checkbox in the settings window of the object designer (see “**Workspace of the Object Designer**”) to display inherited columns.

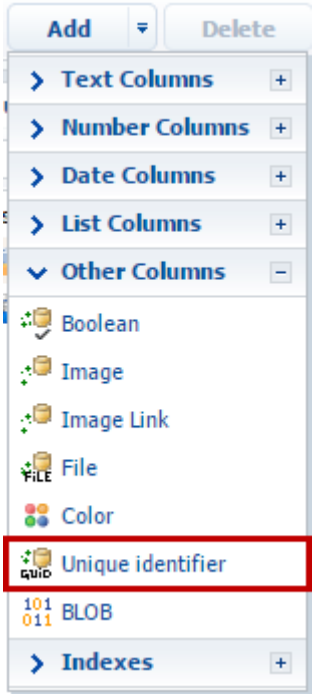
Save the object schema to preserve the metadata changes. Publish the schema to create the corresponding table in the database and make the changes available to bpm’online users.

Adding a custom column to an entity

This section covers adding an Id column to an entity schema.

Use [Add] button (Fig. 8) or [Add] command from the context menu of the *Columns* node in the entity structure to add a custom column. Select the column type from the drop-down list and specify column properties. To add an Id column, execute [Add] > [Unique identifier] command.

Fig. 8. Adding an Id column



NOTE

Enable the [Show Entire List of Column Types] option in the properties window of the "**Workspace of the Object Designer**” to display all types of columns in the “add” menu.

Specify the properties of the Id column (Fig. 9):

- [Title] – column localized title. The default value is set by the object designer and can be modified.
- [Name] – column name. The default value is set by the object designer and can be modified.
- [Data type] – the type of the data in the column. The default value is set by object designer depending on the selected command and can be changed.
- [Required] – specify that the column is required. Since the Id column cannot be empty, select “Application Level” for this property.
- [Default value] – set the column’s default value. Choose “Select from System Variables” from the default value dialog box (Fig. 10). Then select the name of the system variable in the [Name] field. Select the “New Id” variable, which generates unique Ids.
- [Usage mode] – select the “Advanced” mode.

Fig. 9. Adding the Id column

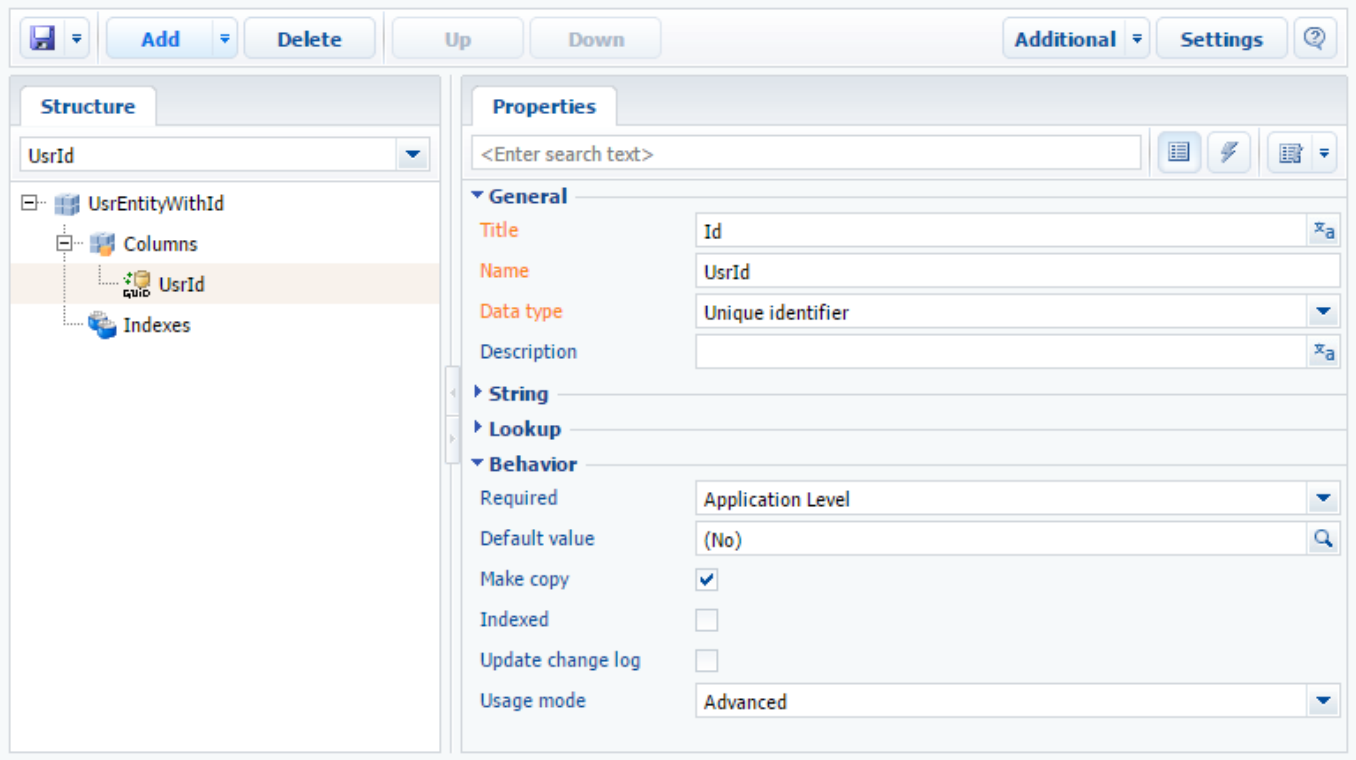
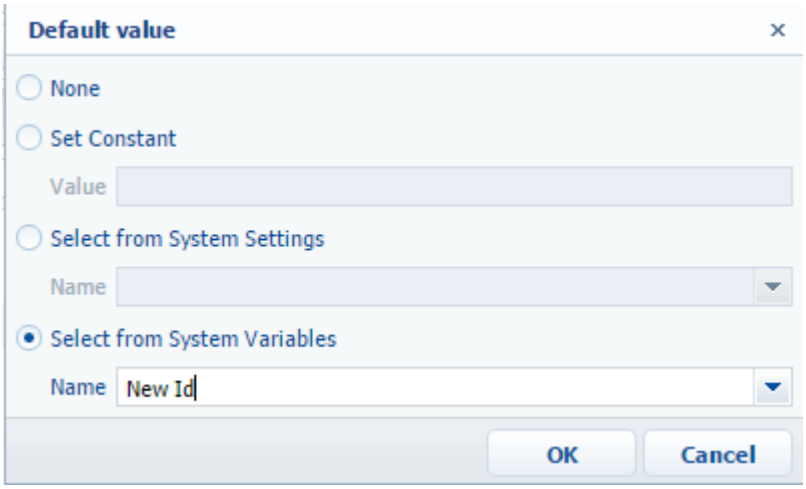


Fig. 10. Setting the default value



Save the schema after setting values for all required attributes.

Adding indexes to the object

Indexes also can be added to the object. They will be automatically created in the database table when the object is published.

To create an index by one column, select the [Indexed] checkbox in the [Behavior] property block. All reference columns are indexed by default.

You can create a composite index in a following way:

1. Select [Add] > [Index] in the context menu of the [Indexes] element. You can specify a custom name for the index or select the [Generate Name Automatically] option. After that, the unique index name will be generated by the system.
2. To implement an integrity constraint for the columns of the index, i.e. to exclude the possibility of inserting duplicate combinations of values, select the [Unique] checkbox for the index.
3. Then add the necessary columns to the index. Select [Add] > [Indexed column] in the context menu of the

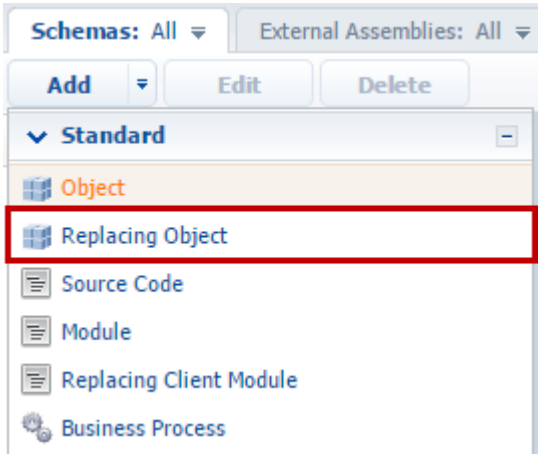
[Indexes] element. Select the object column and specify the sorting direction for the added indexed column.

Creating the replacing object schema

Replacing object schemas are used to extend the functions of the already existing schemas. The existing schemas also may be replacing and belong to the different packages.

To create a replacing object schema, go to [Configuration] section and select custom package to add new module schema. On the [Schemas] tab, select [Add] > [Replacing Object] (Fig. 11).

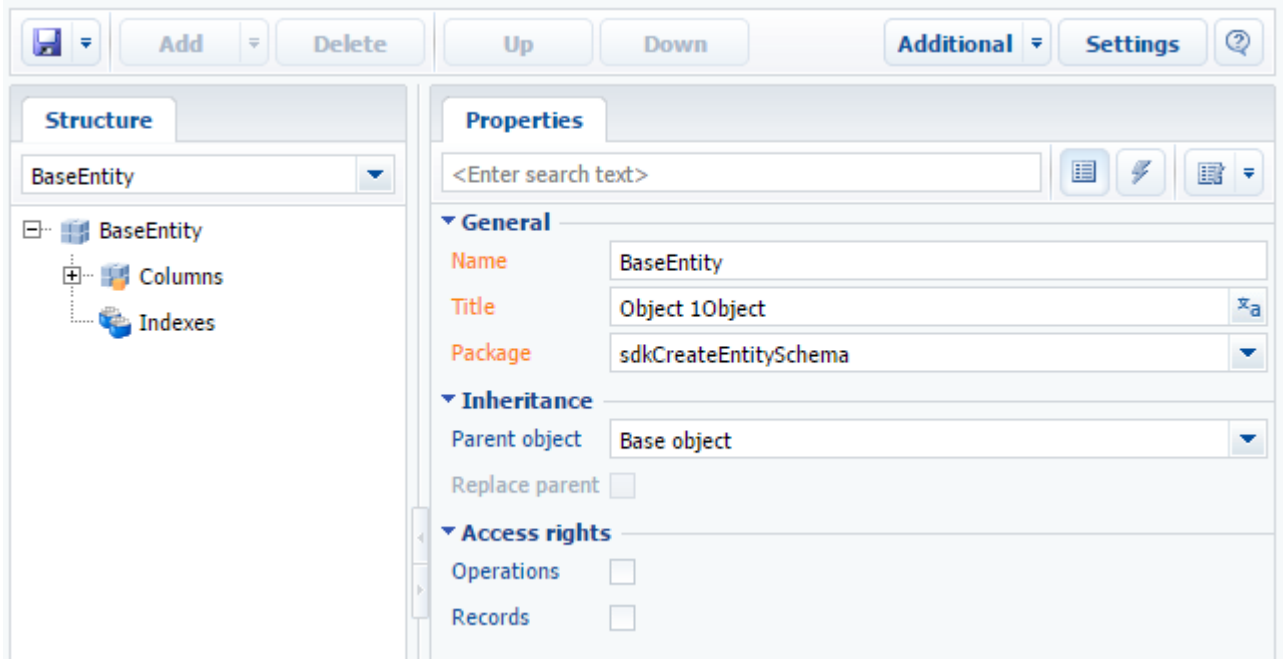
Fig. 11. Command for creating a replacing object schema



To implement the replacement of the new entity schema, select the root element of the data structure in the entity schema (Fig. 5, 1). In the [Parent object] field of the schema properties, select the base entity schema whose functionality you need to replace. To replace the functionality of [Base object] schema, start typing the schema title in the [Parent object] field and select the corresponding value from the drop-down list.

After confirmation of the selected parent object (Figure 6), the other property fields are filled in automatically (Fig. 12, 1).

Fig. 12. Main properties of the replacing object schema



After implementing the changes, publish the replacing object schema.

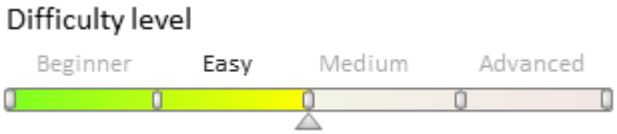
Saving and publishing objects

All structure changes of a business object are stored in RAM.

Save the schema to preserve the changes at the metadata level. To do this, select the [Save] command in the object designer.

To implement changes at the database level, the object must be published. The created (or modified) physical objects in the database (tables, columns, indexes) are the result of successful publication of an object in the [Configuration] section.

Creating the [Source code] schema

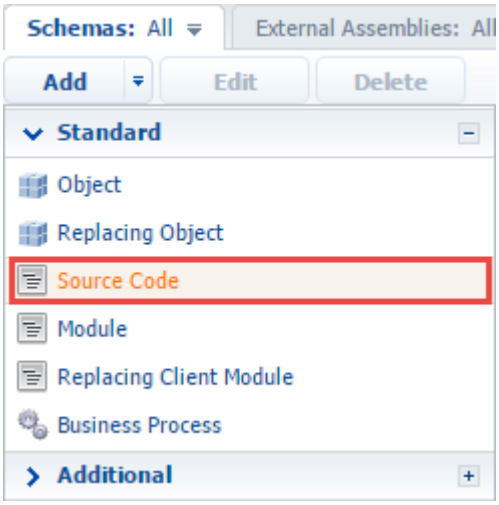


Perform the following actions to create a non-visual module schema.

1. Go to the [Configuration] section and select the custom package to add a new schema.

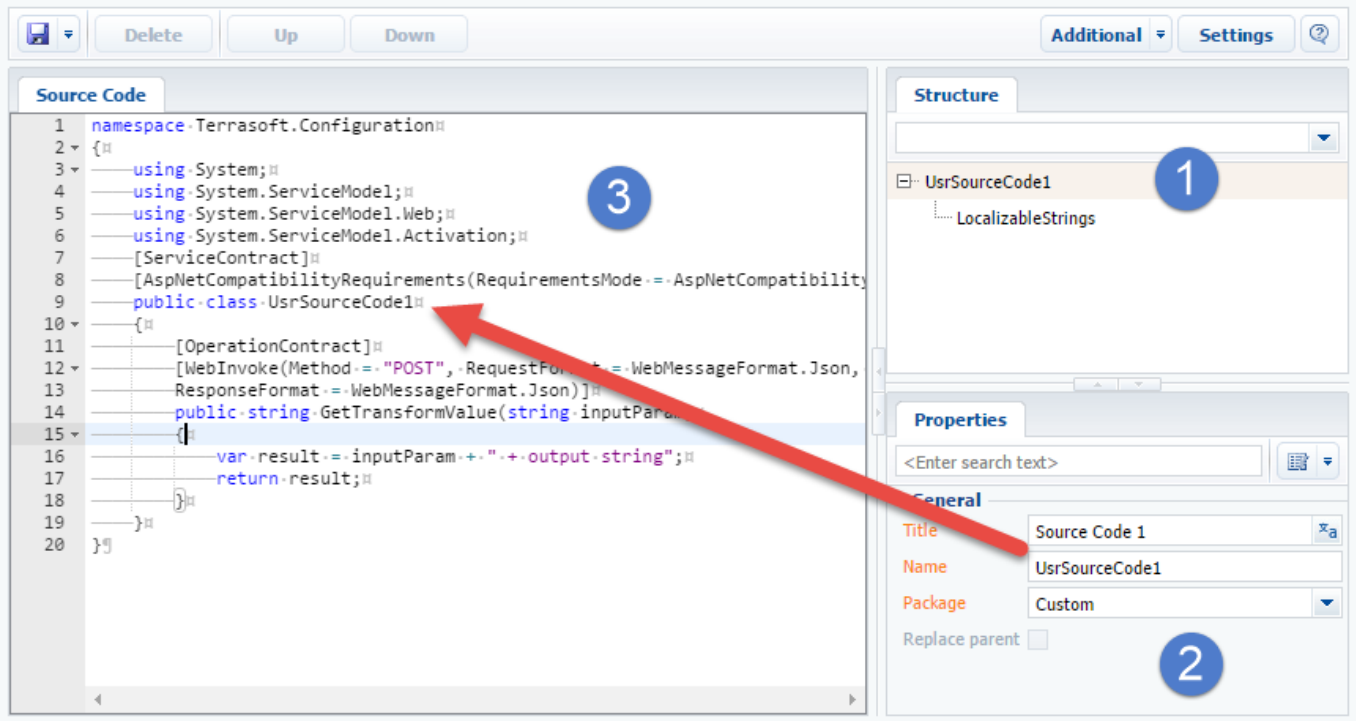
2. On the [Schemas] tab, run the Add > Source Code command (Figure 1).

Fig. 1. Adding a new [Source code] schema



3. Select the root element of the structure (Fig. 2, 1) and fill in the created schema properties (2) in the schema designer.

Fig. 2. The [Source code] schema designer



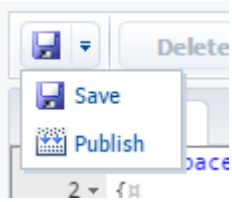
Main [Source code] schema properties:

- [Name] – schema name. May contain only Latin characters and numbers. Includes the [Prefix for object name] system setting prefix (*SchemaNamePrefix*).
- [Title] – schema title. May be localized.
- [Package] – a custom package used to create a schema.

4. Use the [Source Code] tab of the schema designer to add the source code (Fig. 2, 3). Make sure that the source code declares a class with a name that matches the schema name.

5. Publish the schema (Fig. 3):

Fig. 3. Saving and publishing a schema



ATTENTION

The schema designer uses RAM to process changes. Save the schema to apply changes to the schema metadata. To do this, click [Save] in the object designer. Publish the schema to apply changes to the database.

Development resources

Contents

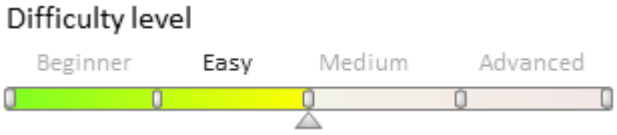
- **Built-in development tools**
- **Development in the file system**
- **Working with WorkspaceConsole**
- **Client code debugging**
- **Server code debugging**

Built-in development tools

Contents

- **The [Configuration] section**
- **The [Configuration] section. The [Data] tab**
- **Source code and metadata viewport**
- **Designers of configuration items**

The [Configuration] section



Introduction

The [Configuration] section is designed for managing configuration elements that implement bpm’online configuration functions.

The [Configuration] section tools enable:

- Managing packages that comprise system functions, as well as managing the package contents.
- Expand and modify bpm’online functions.
- Organize the integration with subversion control systems.
- Manage the development processes and transfer changes between the working environments.
- Manage workspaces.

To start working with the [Configuration] section, go to the [System designer] – [Advanced settings] – [Configuration].

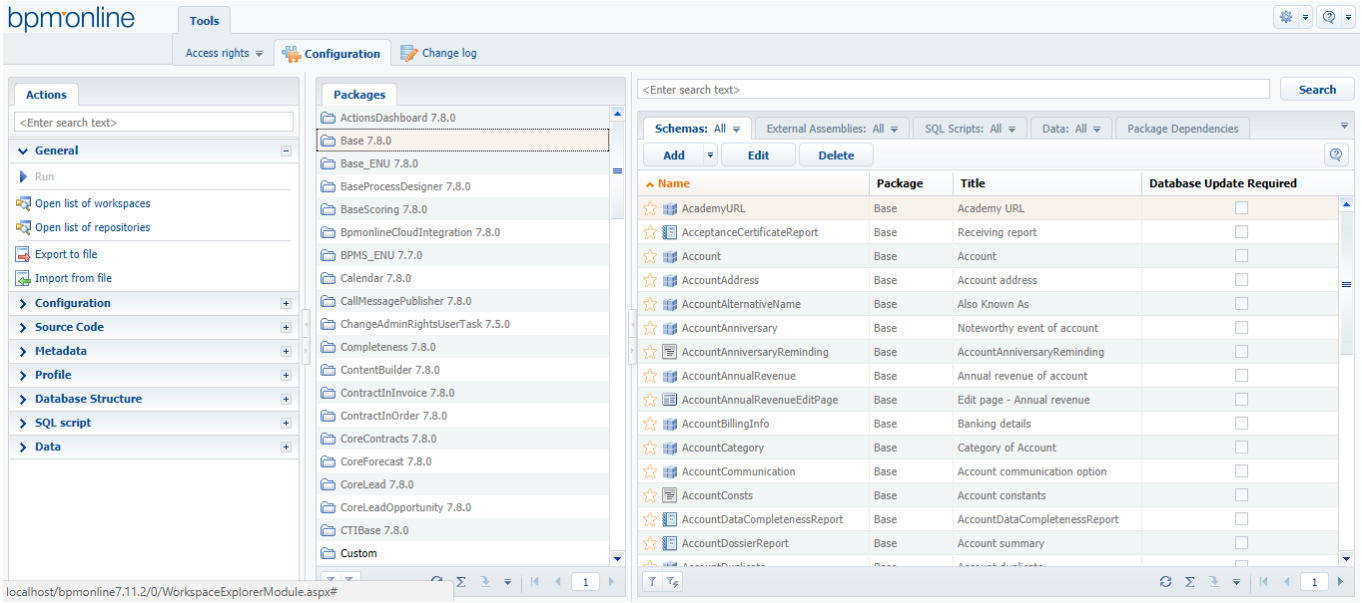
NOTE

You can open the [Configuration] section using a direct link: [application website address]/[Workspace number]/WorkspaceExplorerModule.aspx, for example: http://my.bpmonline.com/0/WorkspaceExplorerModule.aspx.

Starting with version 7.8.4, the WorkspaceExplorerModule.aspx has become available via alternative paths: /dev.aspx or /dev. For example, http://my.bpmonline.com/0/dev.

The section interface is available on Fig. 1.

Fig. 1. The [Configuration] section



Actions in the [Configuration] section

The actions are available on the [Actions] tab of the [Configuration] section side panel, as well as in the section’s context menu. The actions are divided into several groups.

General

[Run] – opens a selected page configuration element in a separate window. The opened page will work as if was opened from the regular UI. The actions is available only for schemas of the “Page” type.

[Open list of workspaces] – opens the [List of custom workspaces] window used for creating, setting up and deleting workspaces.

[Open list of repositories] – opens the [List of repositories] window used for creating, configuring and deleting links to subversion control repository.

[Export to file] – saves the selected schema to an *.md file, which can be imported to a different workspace (see “**Transferring changes using schema export and import**”).

[Import from file] – imports the specified *.md file of a configuration element to a package currently selected in the [Packages] list. As a result:

- If the workspace does not yet have a schema with the same name, a new schema will be added. It will be identical to the one originally saved to the imported *.md file.
- If the workspace already contains an element with the same name, it will be replaced with the imported element.

ATTENTION

If the workspace already contains a schema with the same name as the imported one, the current schema will be completely overwritten. The new schema will be completely identical to the imported one, match its properties and logic, including the inherited logic.

Configuration

[Compile modified items] – publishes changes from the configuration status whose status is “changed”. As a result, application’s executable files will be updated. Changes will become available to the users who work in this workspace.

[Compile all items] – publishes all configuration elements and compiles them. As a result, application’s executable files will be updated. The [Compile all items] action also exports static content to the ...\Terrasoft.WebApp\conf (see “**Client static content in the file system**”).

[Restore from repository] – cancels all changes in the current workspace and restores its state to the last committed state. Unavailable in the file system development mode (see “**Development in the file system**”).

[Verify configuration] – verifies the workspace for errors and invalid links (data, script, access to the base packages from the dependent ones, etc). The verification results are presented to the user in the form of a report.

> ⚠ ATTENTION
>
> Starting from version 7.12.1, the [Verify configuration] action has been deleted from the configuration.

[Download packages to file system] – exports the packages from the application database to the following directory: *...\Terrasoft.WebApp\Terrasoft.Configuration\Pkg*. Available in the file system development mode only (see “**Development in the file system**”).

[Update packages from file system] – imports the packages from the following catalog: *...\Terrasoft.WebApp\Terrasoft.Configuration\Pkg* to the database. Available in the file system development mode only (see “**Development in the file system**”).

Source Code

The actions in the [Source Code] group are designed for viewing and generating source code of bpm’online schemas.

> ⚠ ATTENTION
>
> After updating the workspace from the repository and before compilation, run source code generation via the [Generate where it is needed] action.

[Open] – opens the source code of the currently selected schema. The source code is opened in the **source code viewing window**.

[Generate for selected items] – re-generates source code for selected configuration elements (“schemas”) only. Changes in the selected schemas, as well as changes in their parent schemas will take effect.

[Generate for modified items] – generates source code only foe schemas that have been modified in the current configuration.

[Generate where it is needed] – generates source code for all schemas that require source code generation.

[Generate for all items] – generates source code for all schemas in the current workspace. This operation may take some time (longer than 10 minutes).

Metadata

The actions in the [Metadata] group are designed for viewing metadata in which the structure of bpm’online schemas is saved.

[Open] – opens the **metadata view window** for the selected schema.

Profile

The actions in the [Profile] group are designed for deleting the data that was automatically saved in the current user’s profile.

The profile data includes page area status (expanded/collapsed), status of page area splitters, designer view preferences, list settigs (columns and their arrangement), etc.

[Clear for selected pages] – deletes all current user’s profile data for the selected page schemas.

[Clear for all pages] – deletes all current user’s profile data for all pages.

[Clear all] – deletes all current user’s profile data for all pages and designers.

Database Structure

The actions in the [Database Structure] group are designed for making changes in the bpm’online database structure. For example, the database structure requires updating when adding new columns to objects, so that the corresponding column appears in the database as well.

[Update for selected items] – updates the database structure according to the changes in the selected schemas, where the [Database Update Required] checkbox is selected.

[Update where it is needed] – updates the database structure according to the changes in all schemas, where the [Database Update Required] checkbox is selected.

SQL script

The actions in the [SQL script] group are designed for making changes in the bpm’online database via SQL scripts. These actions are available for items on the [SQL scripts] tab.

[Install selected items] – runs the scripts selected on the [SQL scripts] tab.

[Install where it is needed] – runs all SQL scripts of a package, where the [Database Update Required] checkbox is selected.

Data

The actions in the [Data] group are designed for installing package data (such as lookup records) to the database.

[Install selected items] – install the data selected on the [Data] tab.

[Install where it is needed] – installs package data, where the [Needs to be installed in database] checkbox is selected.

Configuration elements

Primary configuration element types are schemas, external libraries, SQL scripts and data. All configuration elements are grouped on the corresponding tabs.

Schemas

The [Schemas] tab contains configuration element schemas. It displays complete list of configuration element schemas or the list of schemas of the package currently selected on the [Packages] tab (Fig. 1).

Different types of schemas are identified with different icons:

 – objects

 – business processes and their actions

 – reports

 – source code and client modules

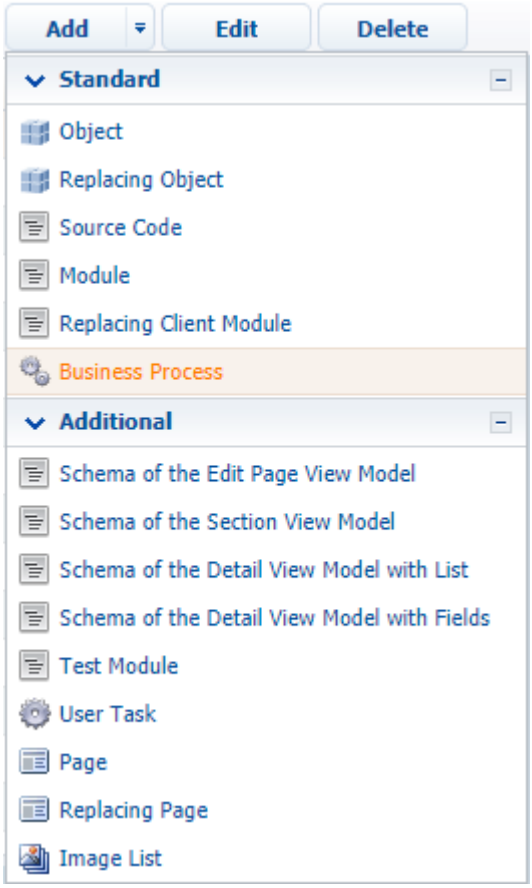
 – image lists

 – pages (legacy configuration element).

Use the list toolbar to add, edit and delete schemas.

[Add] – create a new schema. Use the menu commands of the [Add] button to add different types of schemas. Clicking a schema type in the menu opens the corresponding schema designer (see “**Designers of configuration items**”). For example, select the [Object] command in the [Add] menu to add a new object. The object designer will open.

Fig. 2. Selecting the schema type



[Edit] – opens selected schema in the corresponding designer for editing.

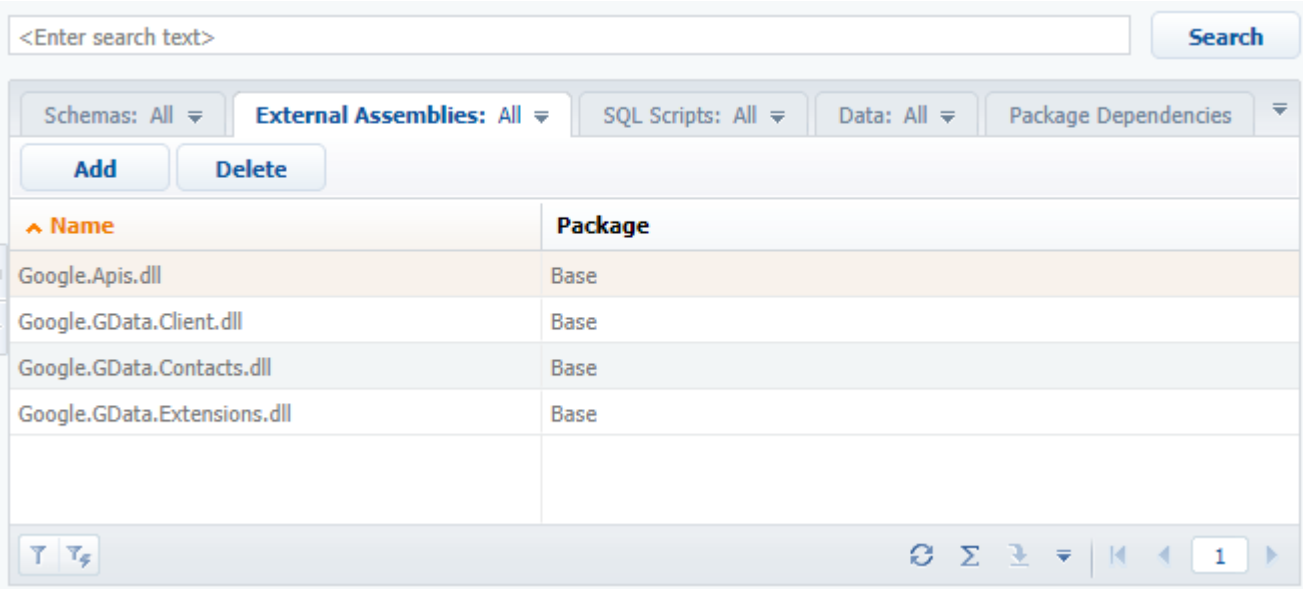
[Delete] – deletes the schema in the current working copy of the current workspace. Running the [Commit package to repository] action will delete the schema in both the workspace and the repository.

> Attention
>
> After deleting an object schema from the configuration, the database still contains the table connected with the object. Use correspondent SQL query to remove the table from the database.

External Assemblies

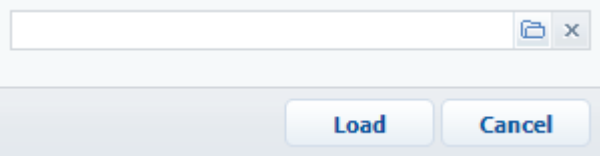
The [External Assemblies] tab contains the list of external libraries used in the configuration schemas (Fig. 3).

Fig. 3. The list of configuration items on the [External Assemblies] tab



To add a new element on the tab, select a package to save the library in; click the [Add] button and select the needed library (Fig. 4) and click [Load].

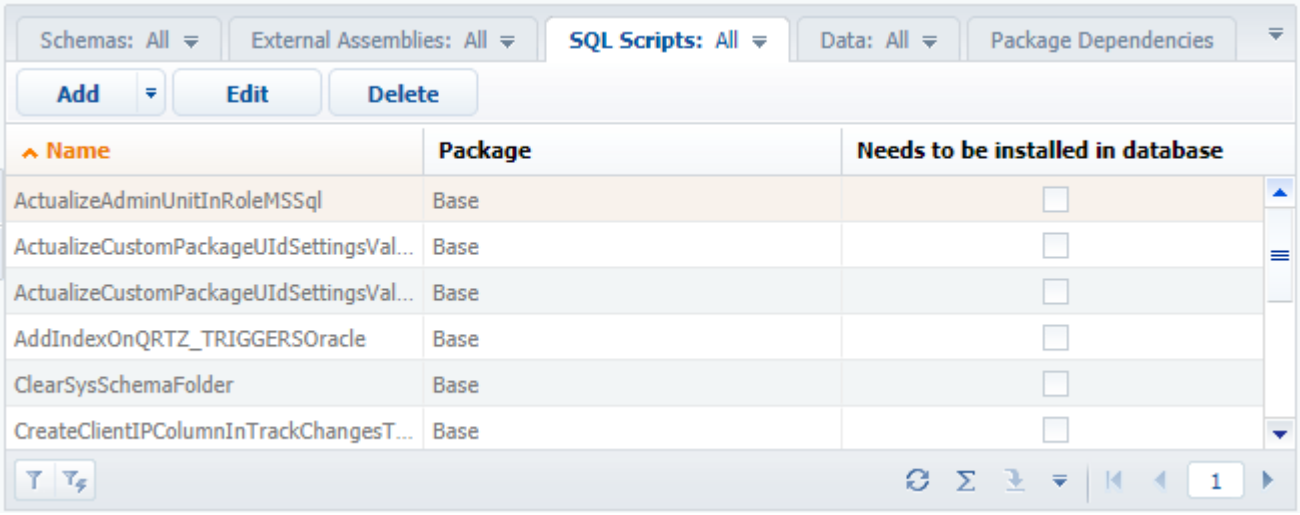
Fig. 4. Adding an external library to a package



SQL Scripts

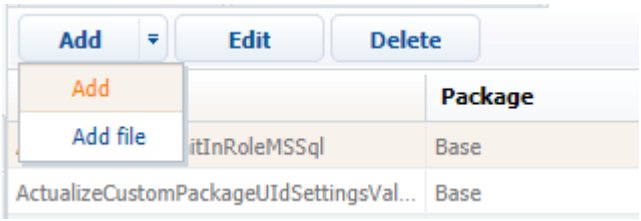
The [SQL Scripts] tab contains the list of SQL scripts bound to the package.

Fig. 5. List of the configuration elements on the [SQL scripts] tab



To add an SQL script, click [Add] and select an item from the menu (Fig. 6).

Fig. 6. The [Add] menu on the [SQL scripts] tab



- [Add] – opens an SQL script binding window, where you can add the script code and set binding parameters.
- [Add file] – opens a window for selecting a script file, which will be loaded to the SQL script binding window.

The toolbar also contains buttons for editing:

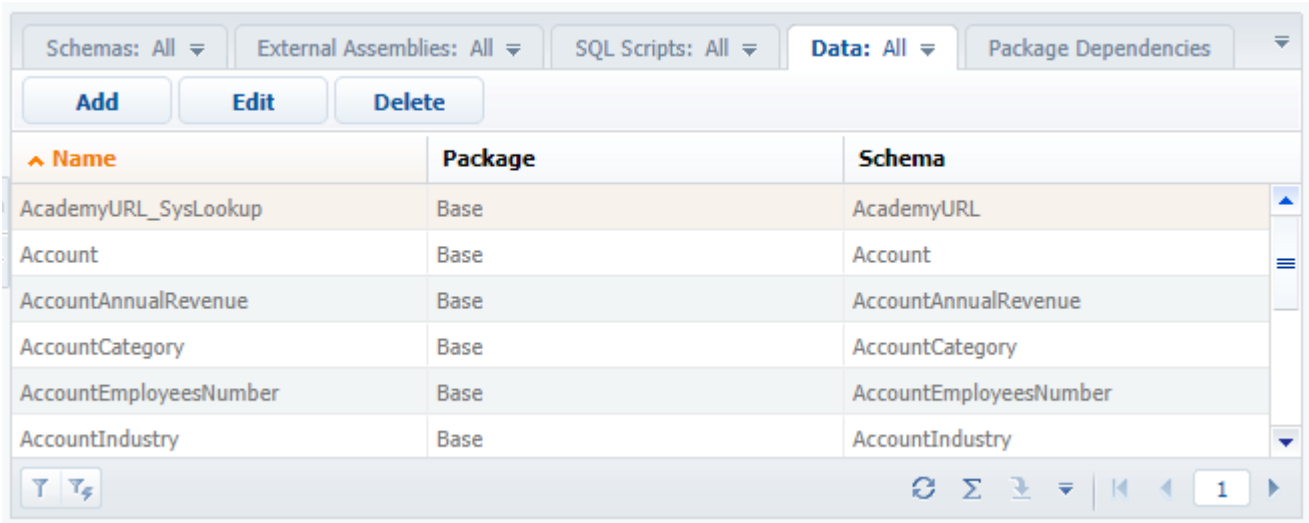
[Edit] – edit a previously added SQL script.

[Delete] – delete a previously added SQL script.

Data

The [Data] tab contains information on the package-bound data.

Fig. 7. List of the configuration elements on the [Data] tab



Use the list toolbar to add, edit and delete data.

[Add] – create a new “data” configuration element. Clicking this button opens the data adding window.

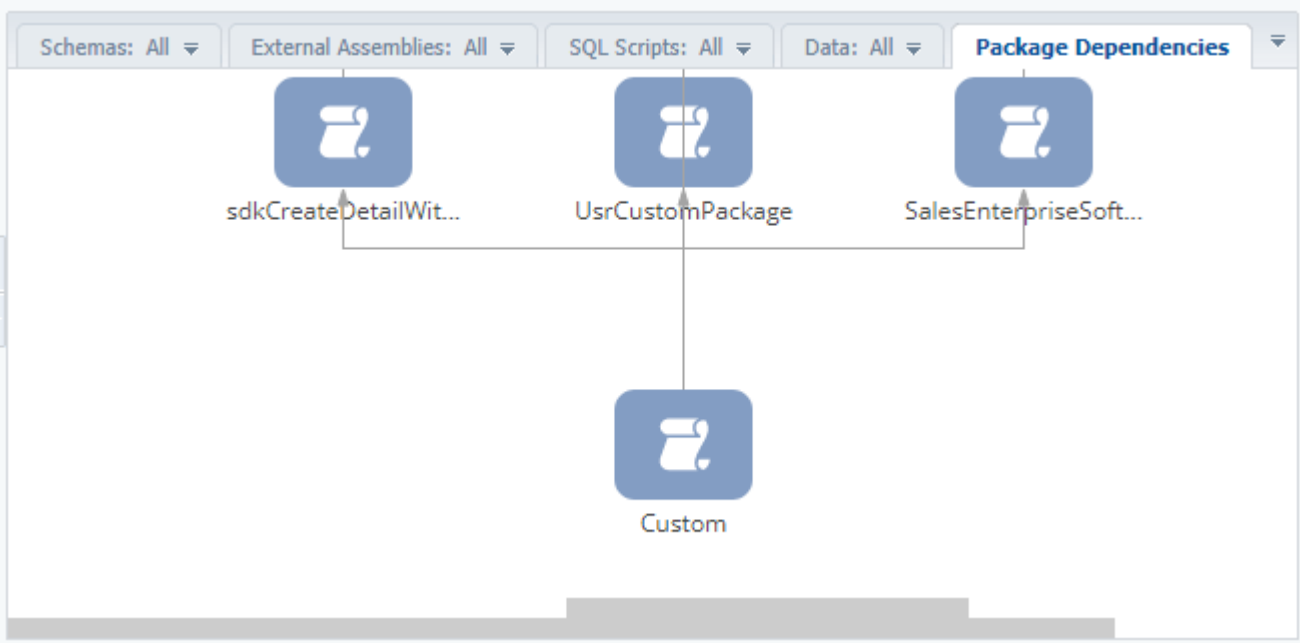
[Edit] – edit previously added data.

[Delete] – delete previously added data.

Package Dependencies

The [Package dependencies] tab displays hierarchy of all packages installed in the current workspace (Fig. 8).

Fig. 8. The [Package Dependencies] tab



The [Configuration] section. The [Data] tab

Difficulty level

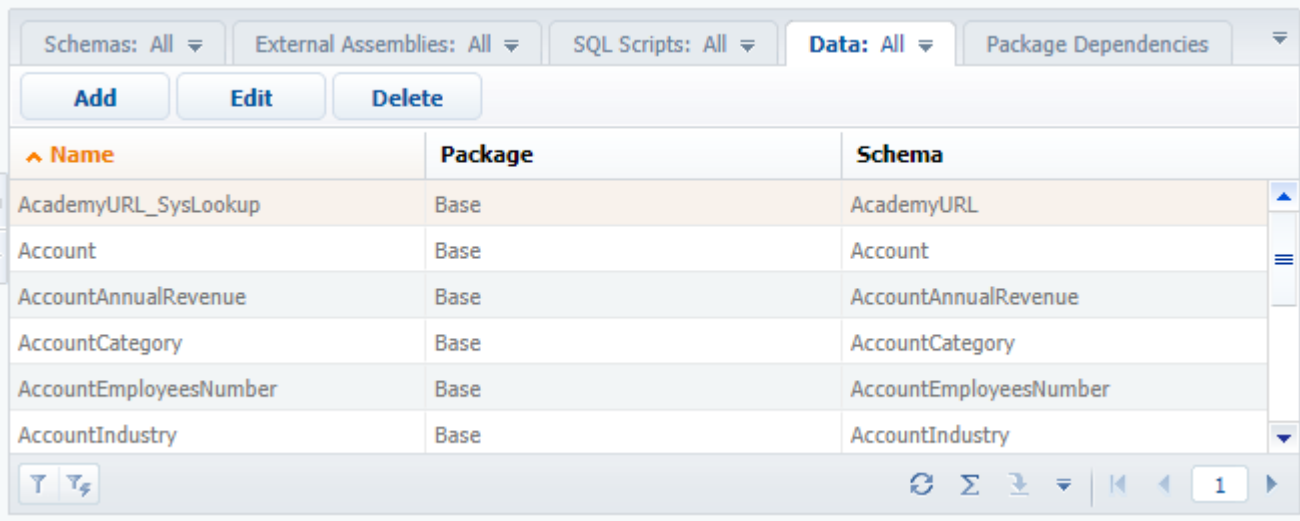
Beginner Easy Medium Advanced

Introduction

When delivering packages to customers, it is sometimes necessary to install additional data for correct operation of all functions. Binding data to a package enables you to achieve this purpose.

The [Data] tab of the [Configuration] section displays information about the data bound to a package.

Fig. 1. List of configuration elements on the [Data] tab



The [Data] tab actions

You can use the following actions on the tab (Fig.1):

[Add] – add new data. The action opens a new page of binding data to package.

[Edit] – edit previously bound data. The action opens a page of binding data to package for editing selected data from the list.

[Delete] – delete previously bound data.

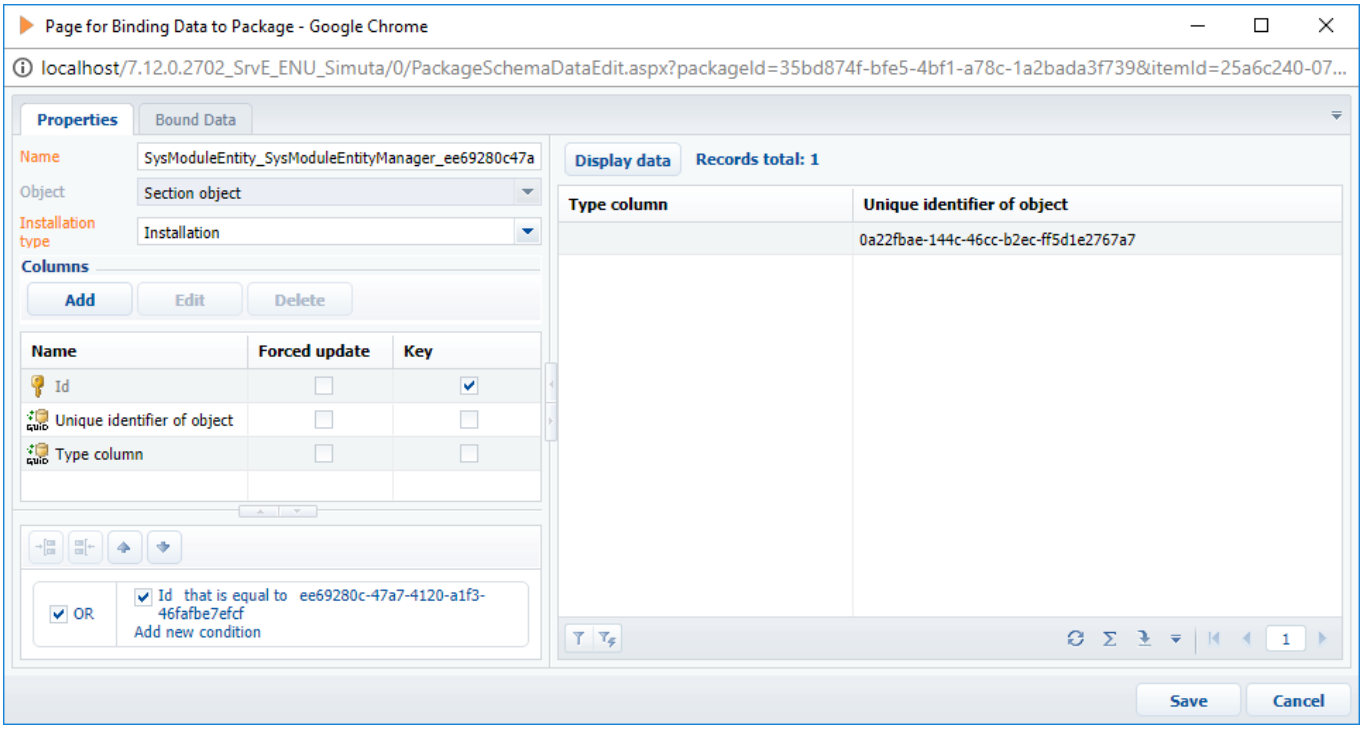
The page of binding data to packages

Displays properties of the data bound to a package. It contains the [Properties] and [Bound data] tabs.

The [Properties] tab

The [Properties] tab is used for installing the properties of package bound data (Fig.2).

Fig. 2. The [Properties] tab



The tab contains the following fields and groups:

1. [Name] – the name of the data bound to a package.

2. [Object] – the object that package bound data are connected to. Use the caption and not the object name when you select it.

3. [Installation type] – specificities of adding the bound data to the application during package installation. The following types are available:

- [Initial installation] – the data will be added to the corresponding tables during the first package installation. This installation type is enabled only if the package **is installed via WorkspaceConsole**. Not recommended.
- [Installation] – the data will be added to the application during the first package installation or updated during the package update process. This installation type is the most common one and is used by default.
- [Update existing] – only the object columns with the selected [Forced update] checkbox in the [Columns] group will be updated during the package update process. This installation type is used, for example, when delivering the hotfix updates.

4. [Columns] – the object bound columns selected in the [Object] field. All object columns are added by default. You can edit the list of columns and their properties via actions. Column properties:

- [Name] – the name of an object column.
- [Forced update] – the checkbox indicating that data update is required when you update the package. It is recommended for installation in case of using the [Update existing] installation type.
- [Key] – the key column checkbox. By default the key is installed for the primary column of an object. If compound keys are used in the database, select the columns that make up a compound key. The primary column should not be included into a compound key.

5. Data filter – enables creating conditions that will be used when filtering the selected object records that are being bound to a package. If the filter is not set, all the application data connected to the selected object will be bound. The filtered data can be displayed via the [Display data] action.

The tab actions:

1. [Display data] – displays data bound to a package.

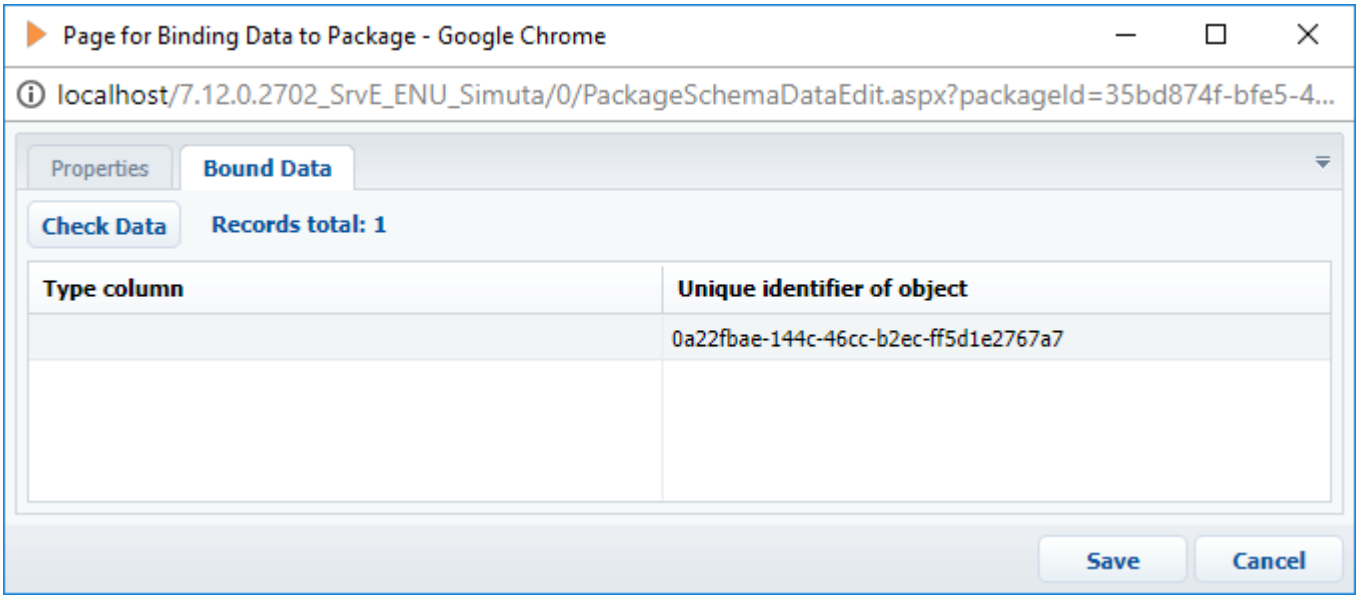
2. [Save] – saves the data bound to a package and closes the page of binding data.

3. [Cancel] – closes the page of binding data without binding the data.

The [Bound data] tab

The [Bound data] tab is used to display the data that have already been bound (Fig.3).

Fig. 3. The [Bound data] tab



The [Check data] action enables checking the correctness of already bound data and the data being bound.

Recommendations

1. The data installation type depends on how a customer will further work with data after the update. The [Installation] type should be used in most cases.

2. We recommend to delete the [Created by], [Modified on], [Modified by], [Active processes] standard columns from the list of columns to bind. Leave only the [Created on] service column.

3. We recommend you to be very careful with using the [Forced update] property of the bound columns. Avoid selecting it for system setting values, external system keys,web-service URLs, i.e. for all columns that affect bpm-online operation capacity.

4. Filter the bound data by identifier or object name (code). We do not recommend using object captions, modification dates, etc.

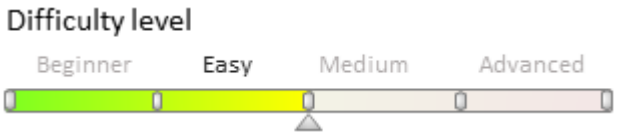
Typical mistakes in data binding

1. Additional data, such as custom lookup or system setting values are not bound for new functions created in custom packages. New functions do not work after installation of a package into a new application, since the necessary data are missing.

2. Filters are not set during binding data of a new custom section. As a result, all data including those used for testing will be bound to the package. Such data volume can be quite big, which leads to longer package installation.

3. The bound data were not applied during package installation via SVN, for example, when automatic data applying is disabled (see "**Installing packages from repository**”).

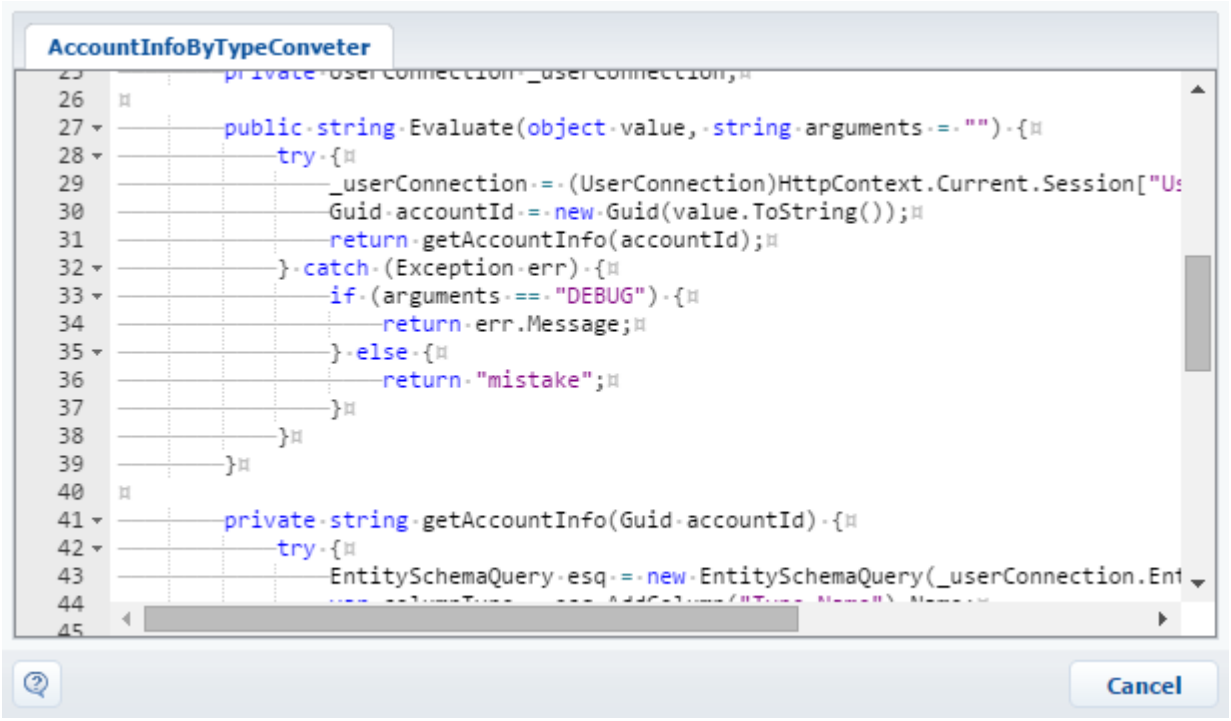
Source code and metadata viewport



You can open the source code and metadata viewport by the following actions *[Source code] –› [Open]* and *[Metadata] –› [Open], respectively*. These viewports also can be opened from item designers.

Source code viewport (Figure 1) displays schema source code.

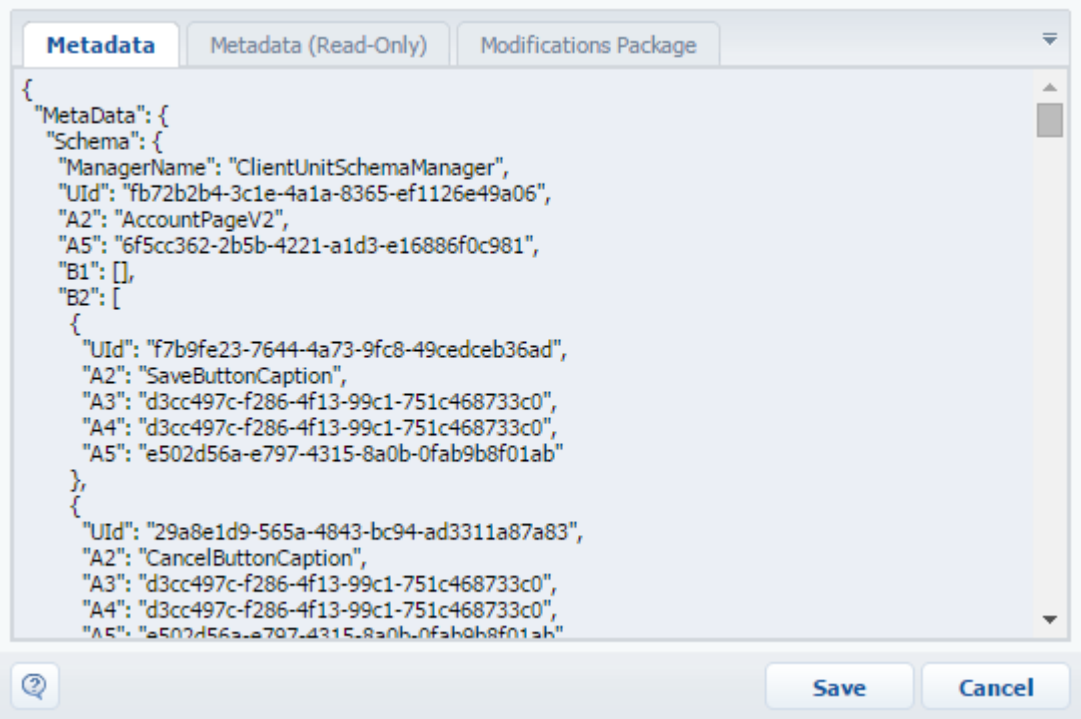
Fig. 1. — Source code viewport



The source code of the schema is generated by the system automatically and can be edited manually.

The metadata viewport (figure 2) is designed for viewing and manual editing of metadata of selected schemas.

Fig. 2. — Metadata viewport, tab [Metadata]

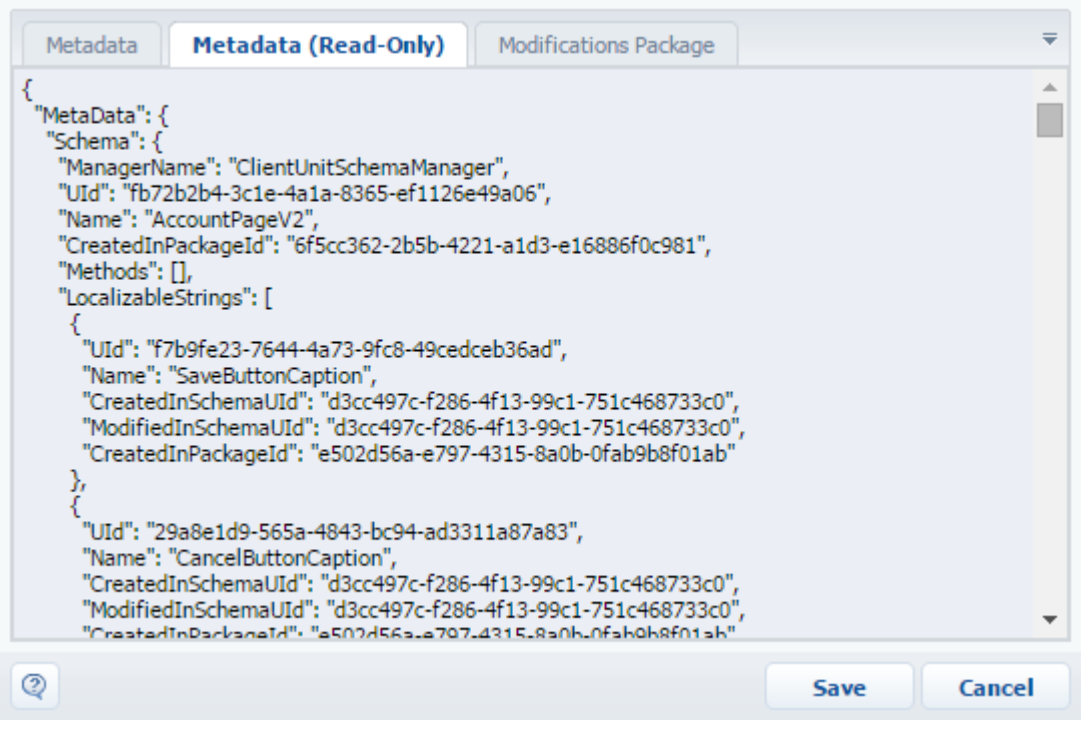


The *[Metadata]* tab shows metadata in their initial view. Use this tab in order to edit metadata manually.

> **NOTE**
>
> The system generates metadata automatically upon saving schemas and it is not recommended to edit them manually. Schema with incorrectly saved metadata can't be opened for editing in the designer unless metadata errors are corrected.

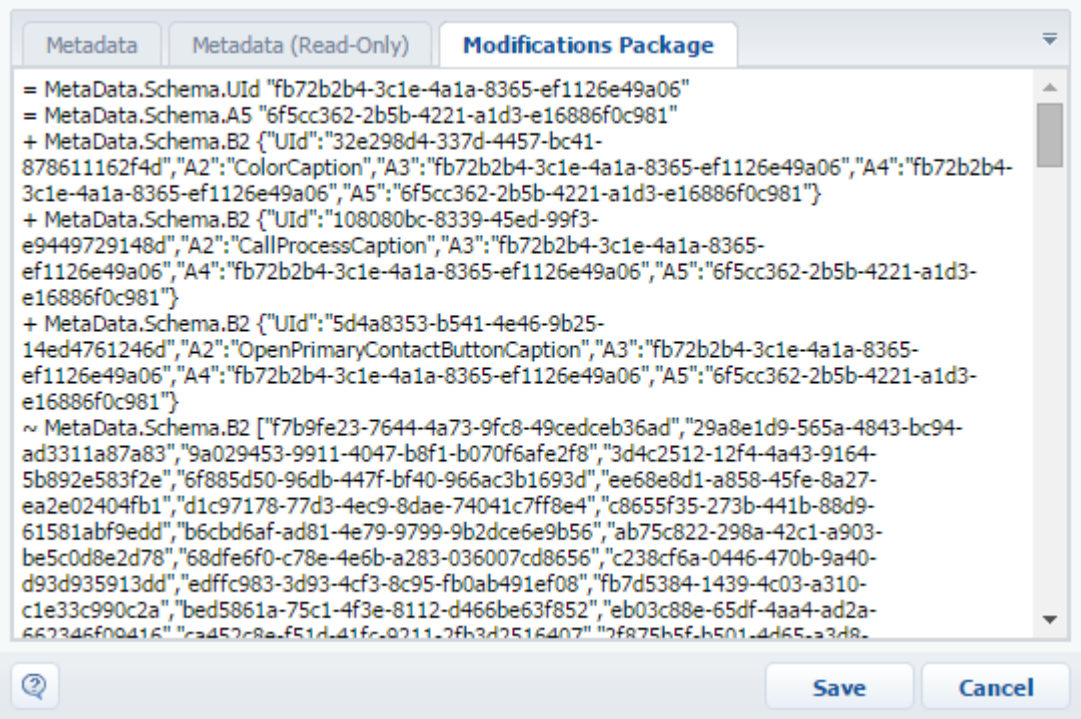
The *[Metadata (Read)]* (Figure 3) displays data that is similar to that displayed by the *[Metadata]* tab, but in a form suitable for reading. Internal identifiers (for example, "A2") are replaced with actual values of items, specified in the [Name] property field (for example, "AccountName"). This tab can be used for manual editing of metadata.

Fig. 3. — Metadata viewport, tab [Metadata (Read-Only)]



The *[Modifications Package]* (Figure 4) shows the list of differences in metadata between the current schema and its parent schema.

Fig. 4. — Metadata viewpoint, tab [Modifications Package]

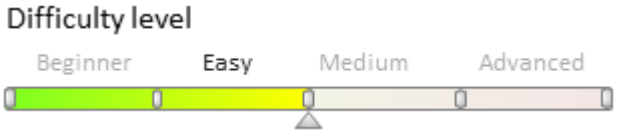


Designers of configuration items

Contents

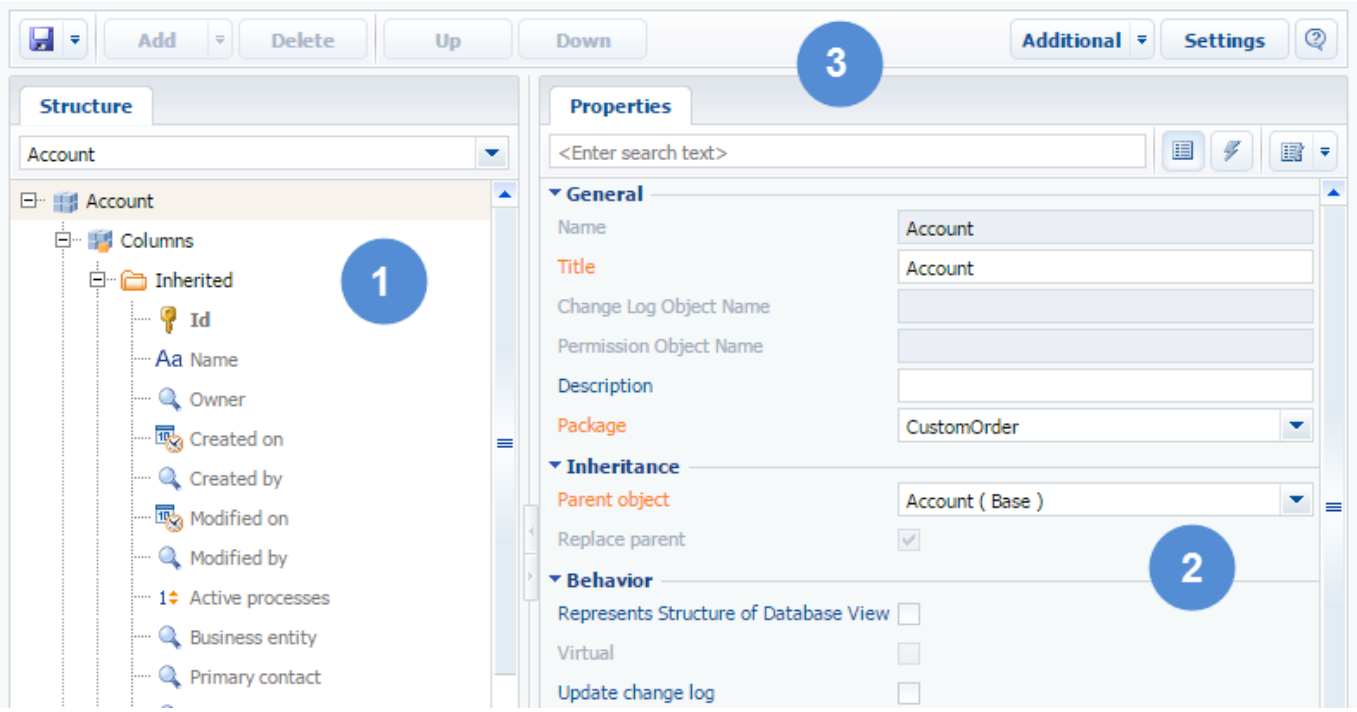
- **Workspace of the Object Designer**
- **Source code designer**
- **Module designer**
- **Process designer workspace**
- **User task designer workspace**
- **Workspace of image list designer**
- **Report designer**

Workspace of the Object Designer



The workspace of the Object Designer (figure 1) consists of several functional areas and contains controls and tools used for creating objects.

Fig. 1. — The object designer workspace



Object structure area (1)

The object structure area shows columns and indexes added to the object. For example, the structure of the "Account" object contains the "Name", "Ownership Type", "Primary Contract", "Parent Account" and other columns.

Column types in the object structure depend on the type of data in the columns. Column indexes are designed to speed up operations with the columns, such as search and filtering.

You can add necessary items to the object structure using the [Add] menu that contains the list of all available object components.

Properties and events area (2)

You can change the set of individual characteristics of the object and its items on the [Properties] tab. This includes setting the default value or making columns required.

This tab also provides the possibility for the generation of events, processing of which allows creating operating logic of the object when the user takes certain actions, for example, filling of required fields before saving entries in the course of event processing.

Toolbar (3)

In addition to the standard buttons, the Object Designer toolbar includes the following buttons:

Add	Add an item to the object structure. The menu contains the list of all available types of columns and indexes.
Delete	Delete of columns from the object.

> NOTE
>
> Deleting columns from an object is similar to deletion of columns from the corresponding table of the system database.

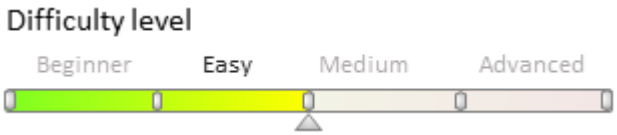
Up	Move the item up in the object structure.
Down	Move the item down in the object structure.

Settings window

In addition to the standard items, the configuration window of the Object Designer also contains the following items:

Show Indexes	Display indexes in the object structure.
Show entire list of column types	Display full list of structure items in menu [Add] (menu shows only basic items by default).
Show system columns	Display the columns, the [Use Mode] property of which contains an "Extended" or "Never" value. For example, columns with information on primary keys ("ID") of object records are not shown in the object structure by default.

Module designer

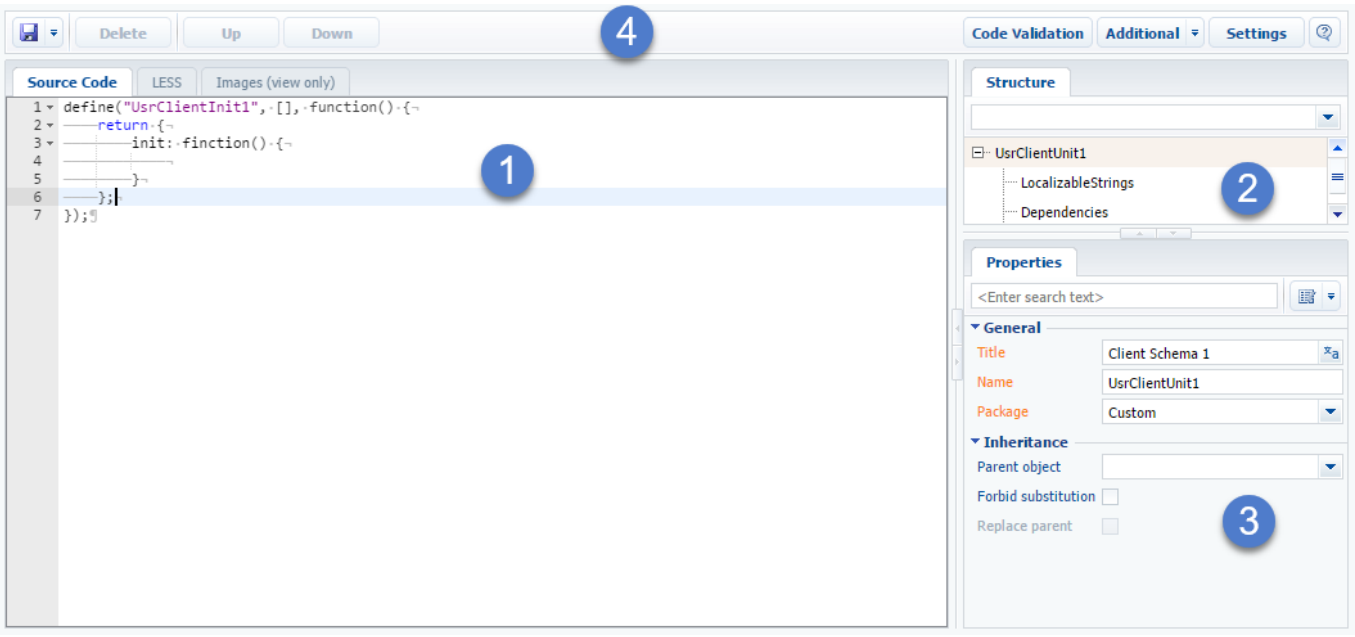


Introduction

The module designer is used to configure the [*ClientUnitShema*] schema. You can add a source code of the JavaScript modules, their dependencies, localized strings, images, messages, parameters and CSS styles to the schema.

A user interface of the module designer (Fig. 1) has several function areas, controls and instruments to create module schemas.

Fig. 1 User interface of the module designer



Source code editor

The area of the source code editor (1) is used to edit the source code of the user JavaScript classes. With the editor you can add, delete and format the source code of the added functions. The debugging of the source code is not provided in the editor. The editor has three tabs:

[Source code] – the source core of the module.

[LESS] – the source code of the CSS styles connected with the module. Styles can be added with the LESS language

compiled to CSS.

[Images (view only)] – allows to view the collection of the images added to the module resources.

Schema structure window

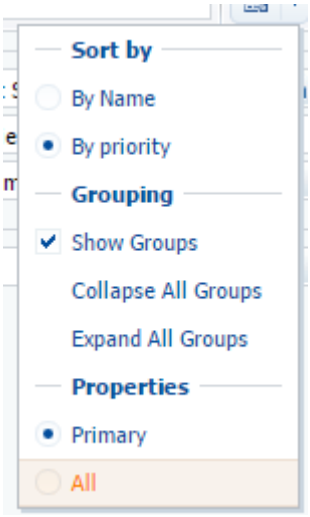
The schema structure area (2) is used to display the schema structure; the root element, resources, dependencies, messages and parameters.

Property window

The properties of the element selected in the structure area (2) are displayed in the properties window (3). If the root element of the structure is selected, then the main properties of the schema source code are displayed. If the localized string is selected, then the properties of the localized string are displayed.

Configuration of the properties view is performed with menu commands (Fig. 2). Select the [All] command to display all properties of the selected element of the structure.

Fig. 2 View properties configuration commands

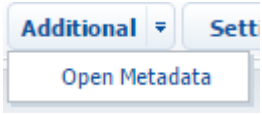


Toolbar

The module designer toolbar (Fig. 1,4) has the following menu and buttons:

Name	Purpose
Save	Menu with save and publish commands.
Delete	Deletes the selected element of the object structure.
Up	Moving an element to a position higher from its current position in the object structure.
Down	Moving an element to a position lower from its current position in the object structure.
Code validation	Performs source code check for formatting errors. Result is displayed on the [JS errors] tab.
Additional	Contains commands for opening metadata view windows (Fig. 3).
Settings	Opens the settings window

Fig. 3 Additional module designer commands

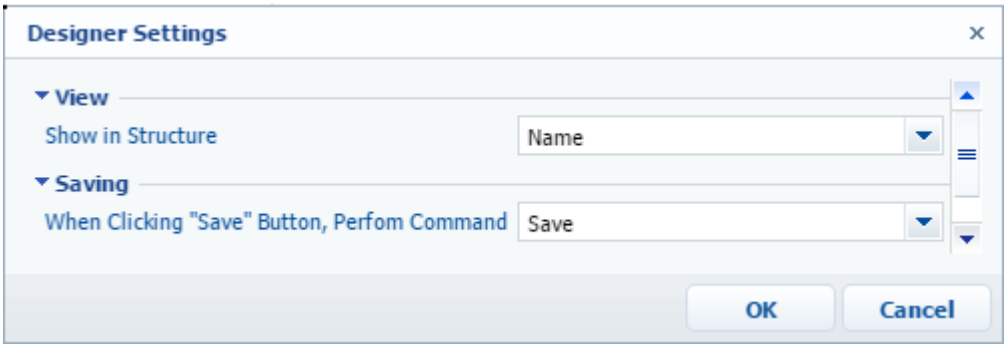


Settings window

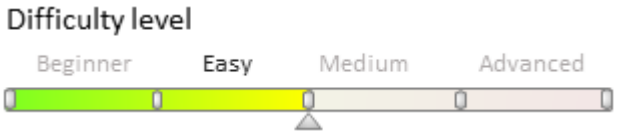
The window of the module designer settings (Fig. 4) has the following items:

Name	Purpose
Show in Structure	Select to show a name or a title of the schema element in the structure.
When Clicking "Save" Button, Perform Command	Select the command that will be performed after clicking the [Save] button. Possible options include saving metadata and publishing a schema.

Fig. 4 Module designer settings



Source code designer

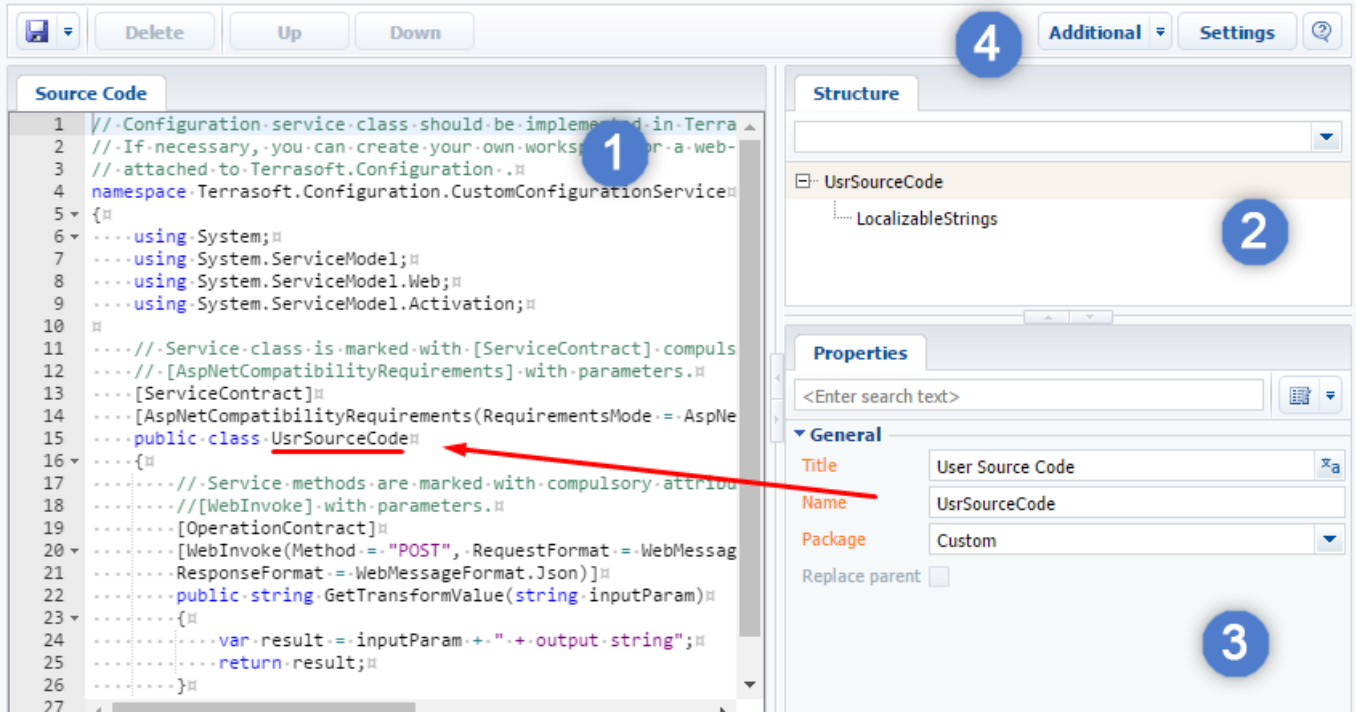


Introduction

Source code designer is used to configure schemas of the *SourceCodeShema* type. Use it to add the C# source code of classes to the schema and localizable strings used to localize UI text implemented by these classes.

The working area of the source code designer (Fig. 1) consists of several functional areas and contains the controls and tools necessary for creating source code schemas.

Fig. 1. Source code designer



Source code editor

Use the source code editor area (1) to edit the C# source code of custom classes. Add, delete and format the source code of custom functions. Note: Source code debugging is not available in the editor.

Schema structure window

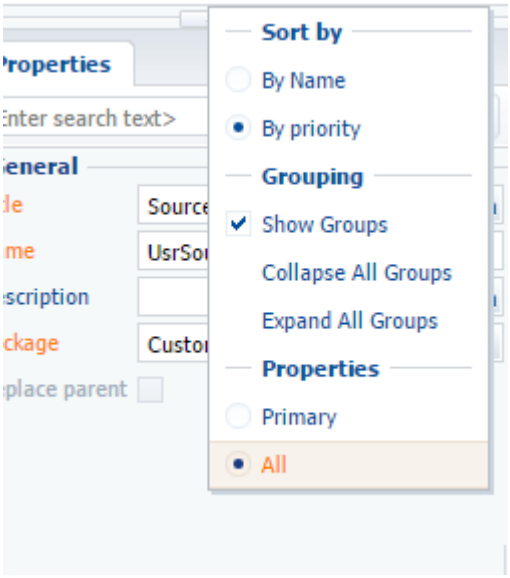
The schema structure (the root element and localizable strings of the source code) is located in the structure area (2).

Properties window

The properties window (3) displays the properties of the element selected in the schema structure (2). If you select the root element of the structure, you will see the general properties of the source code schema on the right-hand side. If you select a localizable string, you will see its properties.

Use the context menu (Fig. 2) to display all available properties of the selected element in the schema structure. Click [All] to switch to advanced mode.

Fig. 2. Context menu

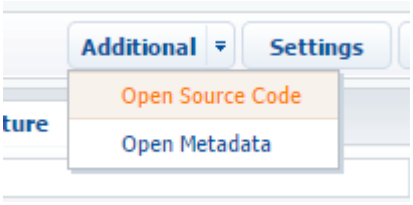


Toolbar

The source code designer toolbar features the following menus and buttons:

Name	Purpose
Save	Saving and publishing the schema.
Delete	Deleting the selected object.
Up	Moving the selected element up.
Down	Moving the selected element down.
Additional	Opening the source code or the metadata window (Fig. 3).
Settings	Opening the settings window.

Fig. 3. Source code designer settings

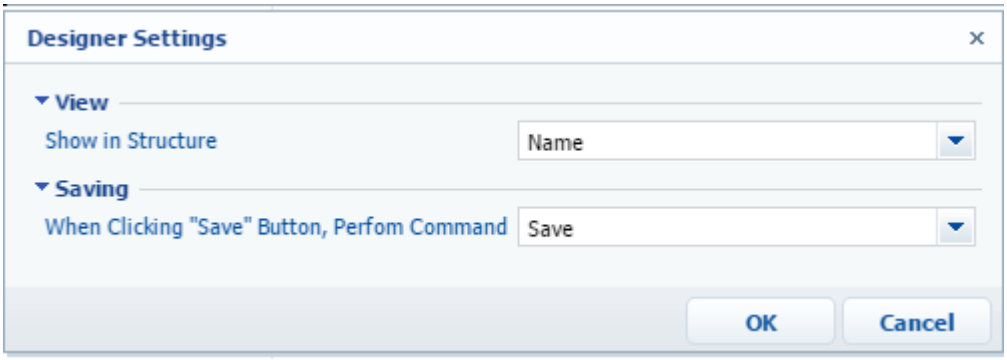


Settings window

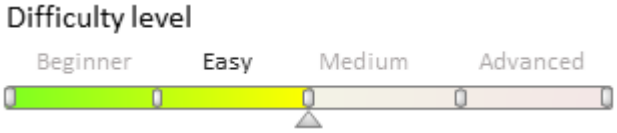
The source code designer settings window (Fig. 4) includes:

Name	Purpose
Show in Structure	Choosing to display either the “Name” or the “Title” of the schema elements in the structure.
When Clicking "Save" Button, Perform Command	Choosing what happens when you click the [Save] button. Possible options – “Save” or “Publish” the schema.

Fig. 4. Source code designer settings



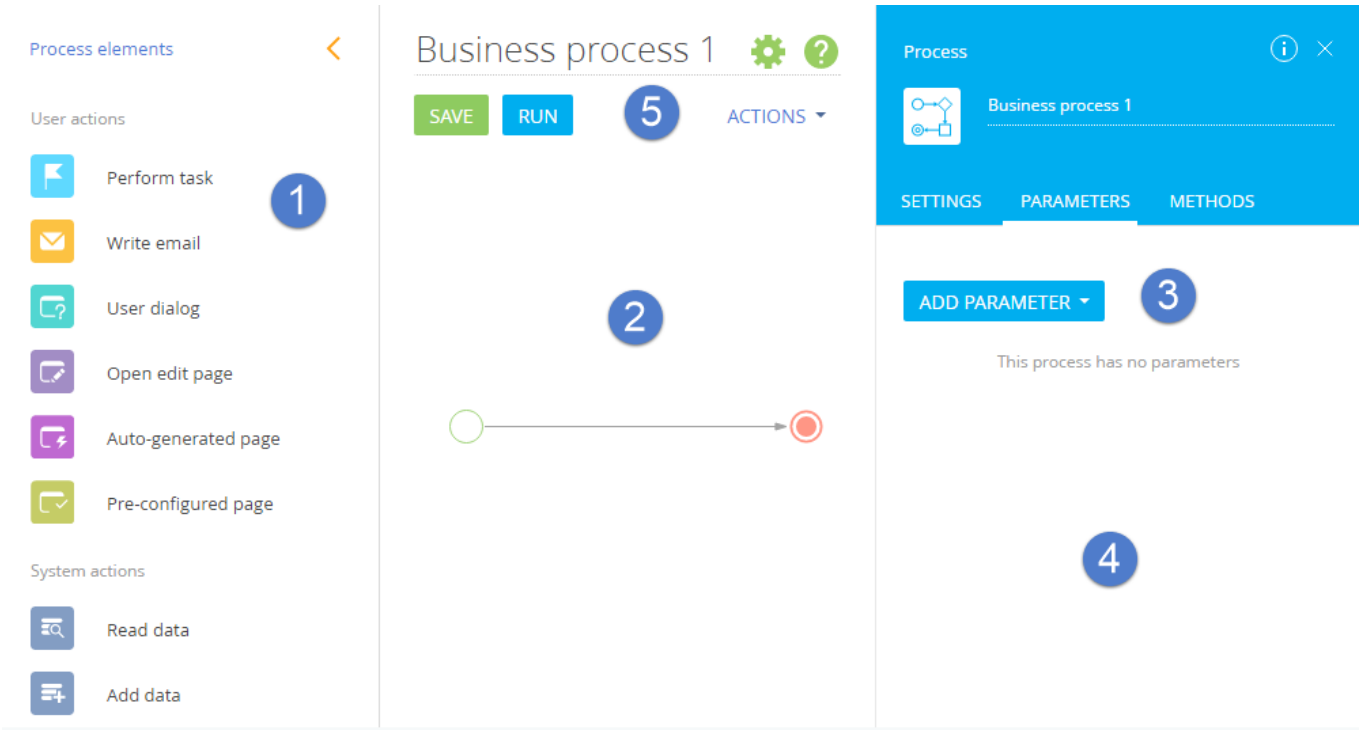
Process designer workspace



Overview

The process designer workspace (Fig. 1) consists of several functional areas and contains control elements and tools to create processes.

Fig. 1 The process designer workspace



Process elements workspace (1)

The [Elements] workspace contains a list of elements which a process can cosnsist of.

Depending on the purpose, elements are divided into several groups: [Actions], [Events], [Boolean operators].

A description of the items, their purpose and properties refer to the process setup documentation.

Designer workspace (2)

A designer workspace displays a graphical representation of a process. You can edit captions and other properties of the process elements. The elements are placed on the workspace by highlighting the desired item.

Process structure workspace (3)

The [Structure] workspace displays a tree structure of process elements, which are displayed on the workspace, and those that appear only in the structure, such as process parameters.

Process structure can include the following groups of items:

• [Links] — displays a list of flows and process connecting objects.

• [Parameters] — sets the parameters of the process elements and their values.

• [Methods] — adds methods used in the process scripts.

• [Messages to user] (LocalizableStrings) contains a list of messages displayed to users in the system interface. These messages can be localized.

• [Namespaces] (Usings) — adds a namespace to the process helps developers to simplify the work with the source code of the process script.

Properties and events workspace (4)

A set of common characteristics of a process and each of its element is available in the [Properties] tab.

Each process element has individual properties. A set of properties depends on the element type. For example, in a conditional flow the property is [Condition].

Toolbar (5)

The process designer menu  contains the following commands:

• [Save] — saves changes made to the process schema. If no changes that require publication have been made to a process, then after saving users will start work with the updated process.

• [Publish] — appears in the menu if changes that require compilation of the updated bpm'online executables have been made to a process.

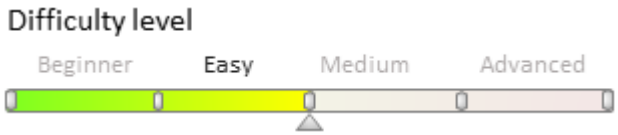
 — switch cursor to regular mode. Use this button to exit the vertical or horizontal displacement of the process elements and to deselect an item in the [Items] workspace.

The [Advanced] process designer menu contains the following commands:

• [Open source code] — opens a window with the process source code.

• [Open metadata] — opens a window of the process metadata.

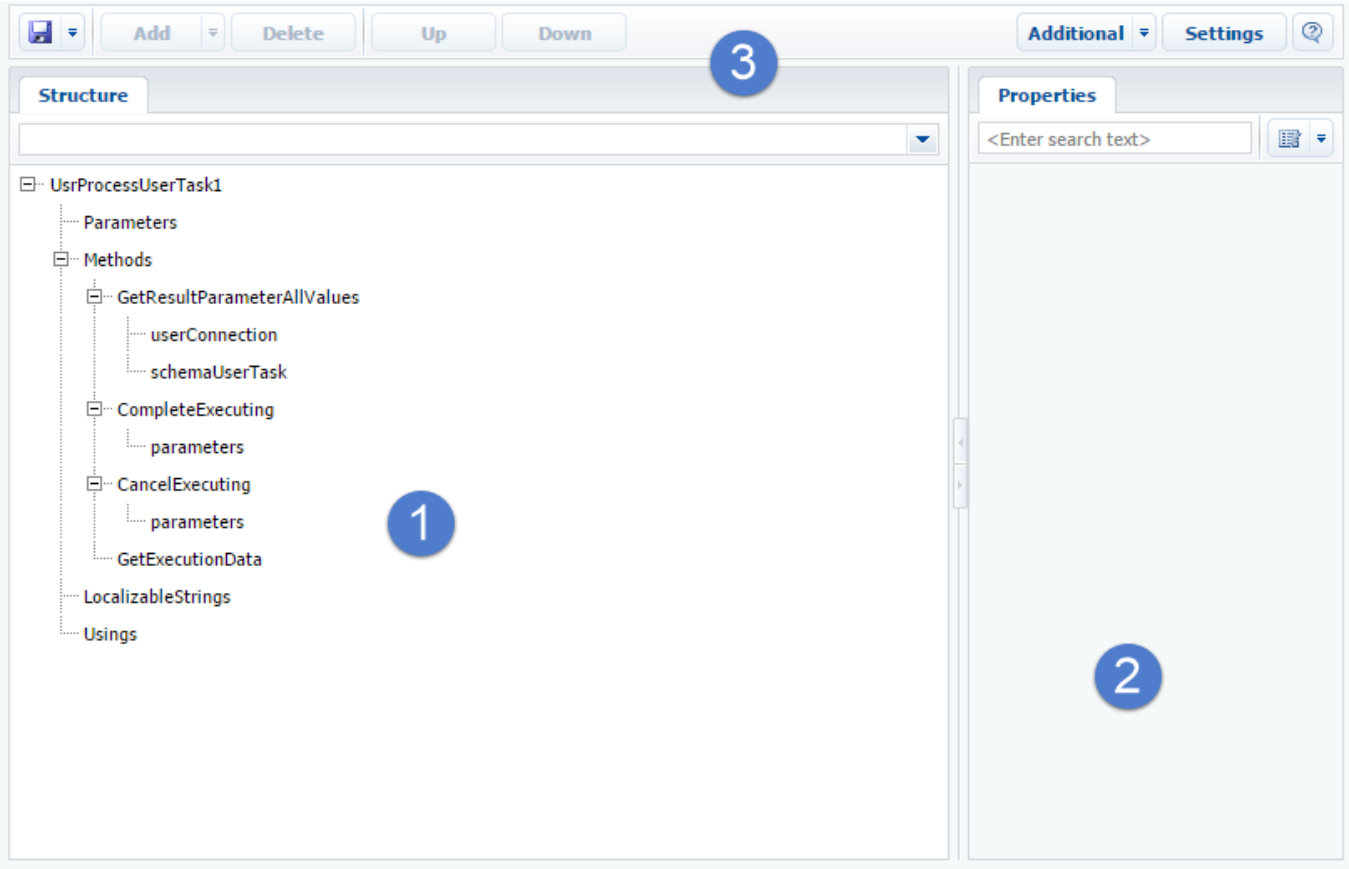
User task designer workspace



Overview

The User Task Designer (Fig. 1) consists of a number of functional areas and contains tools for creating custom activities for use in business processes.

Fig. 1. User Task Designer work area



Structure area (1)

The [Structure] area contains a tree-like structure of the business process elements.

The Properties Area (2)

Use the [Properties] area to modify the number of separate characteristics of a user task and any of its items.

The Toolbar (3)

In addition to the standard buttons, the toolbar of the user task designer also contains the following buttons:

[Add] – adds an item to the user task structure. The item currently selected in the structure will determine the type of an item that will be added by clicking the button. For example, if the [Parameters] group or a parameter is selected in the structure, clicking the [Add] button will add a new parameter. The [Add] button menu also contains the following commands:

- [Add Parameter] – adds a parameter to the user task structure.
- [Add Method] – adds a method to the user task structure.

You can also add an item by using the [Add] command of the right-click menu in the [Structure] area.

[Delete] – deletes the selected item from the user task structure.

[Up] – moves an item up the list in the user task structure.

[Down] – moves an item down the list in the user task structure.

Workspace of image list designer

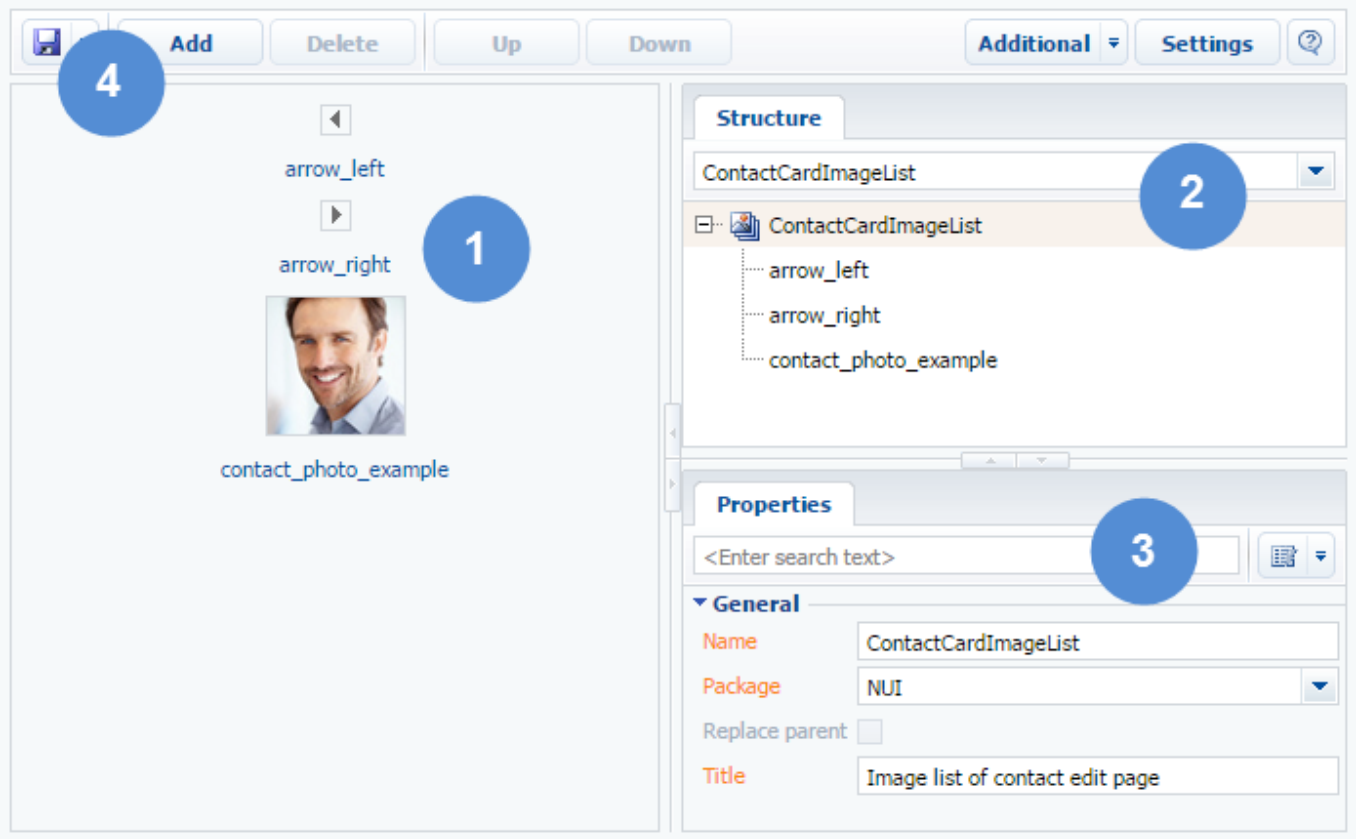
Difficulty level

Beginner Easy Medium Advanced

Overview

The workspace of the image list designer (figure 1) consists of four main functional areas and contains necessary tools for creating image lists.

Fig. 1. — Working area of image list designer



Specifics of the image list designer are described in chapter, "Specifics of handling of image list designer".

Designer image area (1)

List items in the form of image sketches are located in image areas.

Image list structure area (2)

A tree-type structure of image list items is displayed in the [Structure] area.

Property area (3)

You can change a set of individual characteristics of an image list and also each item in the [Property] area. Image files are downloaded into the list through the same area.

Toolbar (4)

In addition to the standard buttons, the toolbar of image list designer also includes the following buttons.

Add	Add a new item into the list. The item doesn't include images upon adding.
Delete	Delete selected elements from image list.

Up	Movement of the item above its current position in the object structure.
Down	Movement of the item below its current position in object structure.

Report designer

Difficulty level

Beginner Easy Medium Advanced

Overview

The [Report] configuration elements are used to form analytic reports to build printables.

Report designer is a separate application supplied along with the bpm’online and used to create and edit [Report] configuration elements. Report designer should be installed on local PC.

ATTENTION

Zip archive with report designer can be downloaded by this link https://academy.bpmonline.com/sites/default/files/documents/downloads/ReportDesigner/ReportDesigner_7_10_en.zip

Contents

- **Setting up the report designer connection with server**
- **Report designer workspace**
- **Report designer features**

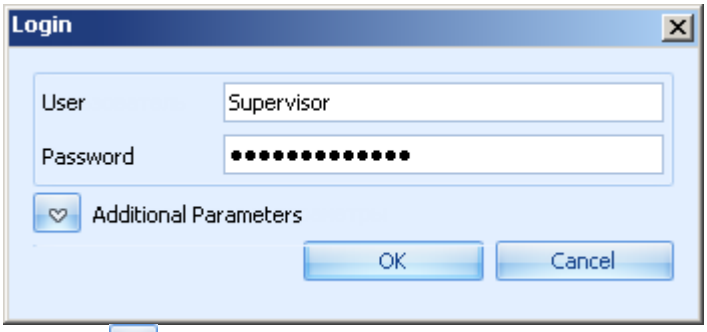
Setting up the report designer connection with server

Difficulty level

Beginner Easy Medium Advanced

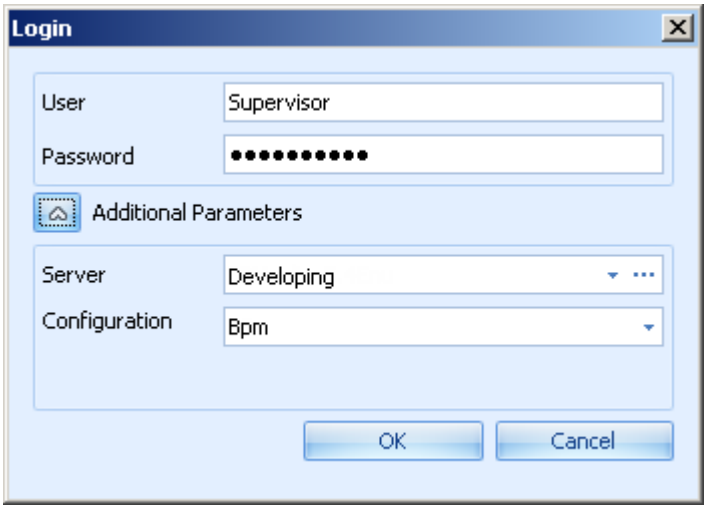
The report designer application starts with the login window (Fig. 1).

Fig. 1. Login window



Click the button to open additional login parameters used to select server and configuration (Fig. 2).

Fig. 2. Login window with additional parameters



Description of all fields of login window is given in the Table 1.

Table 1. Login window fields

Name	Description
User	The user name that is used to log in to bpm’online system.
Password	The password used to log in to bpm’online system.
Server	Server with bpm’online application. If server is not present in the list, click the ... button and add corresponding server in the [Available Servers] window.
Configuration	Name of the bpm’online configuration.

ATTENTION

Starting with version 7.10.0 only the *Default* configuration is available.

[Available Servers] window

[Available Servers] window (Fig.3) is used to configure parameters of connection to server with bpm’pnline application. A list of commands enabled in the [Available Servers] window is given in the Table 2.

Fig. 3. [Available Servers] window

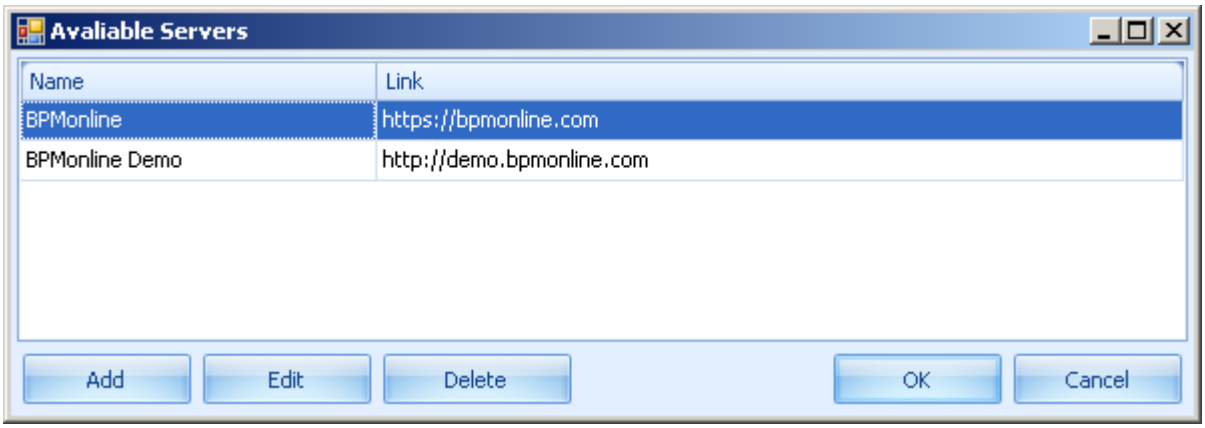
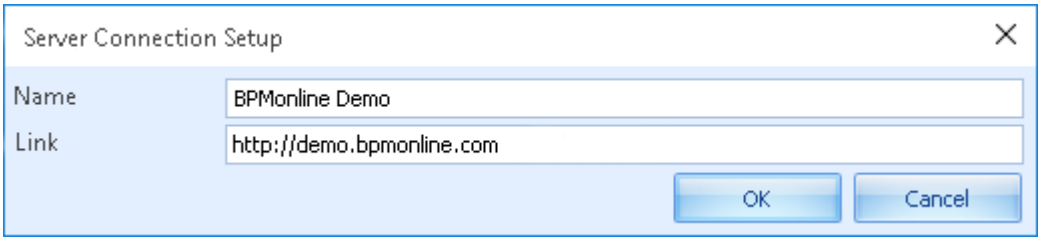


Table 2. Edit commands of the available servers list

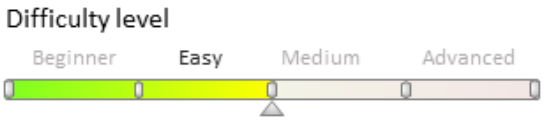
Name	Description
Add	Adding a new server to the list. Click the button to open [Server Connection Setup] window (Fig. 4). Here you can add a name and a link to the new server.

	Server link is the web address of the server (for example, http://mywork.bpmonline.com).
Edit	Modifying connection parameters of the selected server.
Delete	Deleting the selected server from the list,

Fig. 4. Server Connection Setup window



Report designer workspace



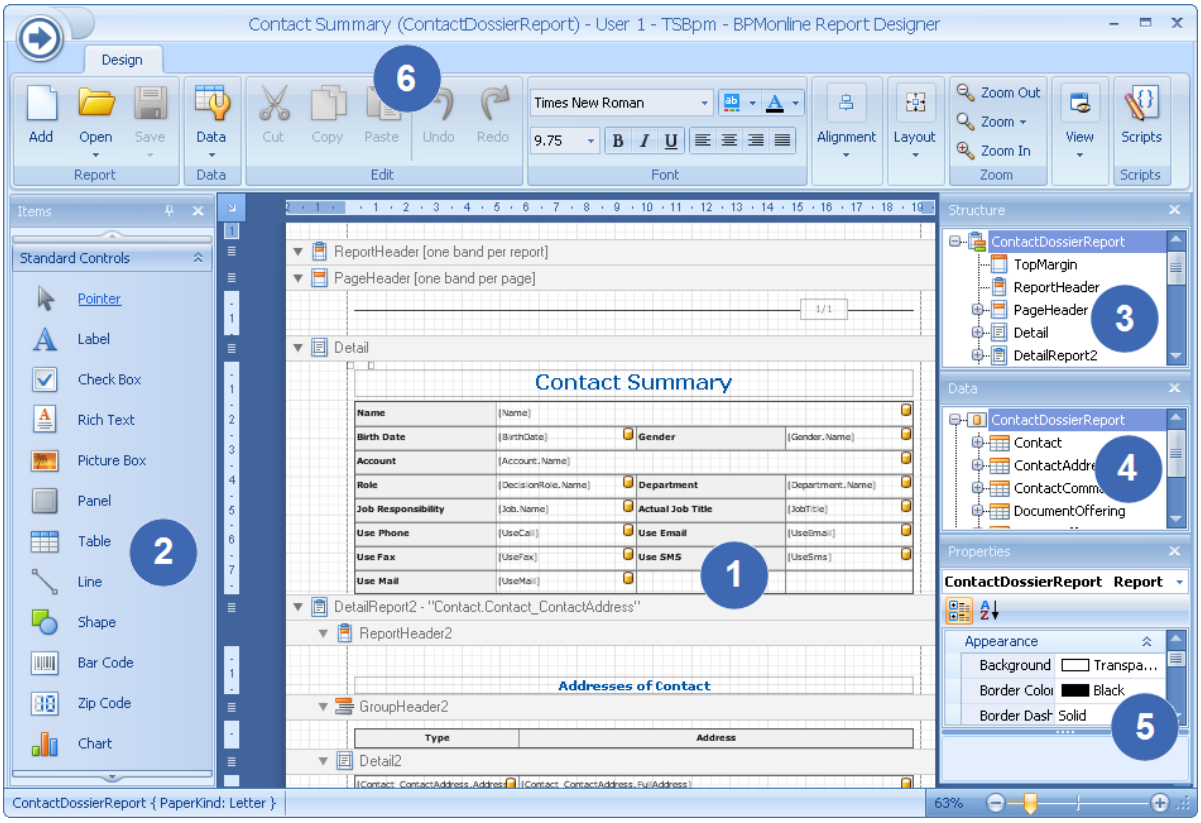
Overview

ATTENTION

A Zip archive with the report designer can be downloaded via this link https://academy.bpmonline.com/sites/default/files/documents/downloads/ReportDesigner/ReportDesigner_7_10_en.zip

The workspace of the report designer (Fig. 1) consists of several functional areas and contains tools that are necessary for creating reports.

Fig. 1. Report designer workspace



Report layout area (1)

This area is the breadboard layout of the report page. Any element from the [Items] window can be placed on this area.

The [Items] window (2)

The window contains a list of elements that can be added to the report page (labels, picture boxes, charts, page breaks, etc.).

> Note
>
> The position of the [Items], [Structure], [Data] and [Properties] windows can be arbitrary in the designer workspace. In addition, some windows can be hidden. Click the [Windows] button and select a corresponding window to display or hide it.

The [Structure] window (3)

The window displays a tree-like structure of elements that were added to the report.

The [Data] window (4)

The window displays the structure of the report source data. For example, this area can display bpm'online section columns. Columns displayed in the [Data] window can be placed to the report designer area.

The [Properties] window (5)

The window displays individual characteristics of the selected report element. Values of the properties can be modified.

The ribbon (6)

Ribbon is a set of toolbars at the top of the report designer workspace. Ribbon helps to find commands to perform tasks.

The [Report] toolbar

The [Report] tollbar is used to create, edit and save reports (Fig. 2). Description of the commands of the [Report] toolbar is given in the Table 1.

The [Report] toolbar

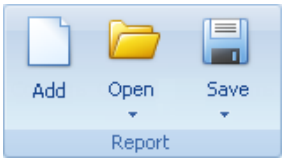


Table 1. Commands of the [Report] toolbar

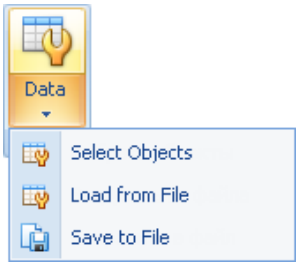
Command	Description
Create	Creates a new report If another report was opened when the [Create] button was clicked, you will receive a warning message to save the opened report before creating a new one. The report will be saved under its current name.
Open	Opens the selected report in the designer. Button menu contains the following items: • [Open] – opens the report from the bpm’online application database. After clicking this menu item, the [Open Report from Server] window with a list of configuration elements of the “Report” type will open. The selected report will be opened in the designer. • [Open from file] – opens the report from the .repx file.
Save	Saves the report. Button menu contains the following items: • [Save] – saves the report to the system database without updating the bpm’online executable files When you save a new report, a new configuration element of the "Report" type is created. • [Publish] – saves the report to the system database and updates the executable files. • [Save to file] – saves the report to the .repx local file . The .repx files can be opened only in the report designer.

The [Data] toolbar

Connection of the report with objects and database is carried out via report data source. The report data source may include columns of system objects and additional custom columns. For example, to add a list of accounts and their industries to the report, the data source must contain the [Name] and [Industry] columns of the [Account] object.

The report data source is created with the [Data] toolbar (Fig. 3).

Fig. 3. [Data] toolbar



The [Data] button menu contains the following commands:

- [Select Objects] – opens the [Objects for Report] window to select object columns. Data of these columns will be used to create a report.
- [Load from file] – loads the report data source from the .xsd file.
- [Save to file] – saves current report data source to the .xsd file. For example, to edit the data source outside the report designer.

The [Objects for Report] window will open by clicking the [Data] button.

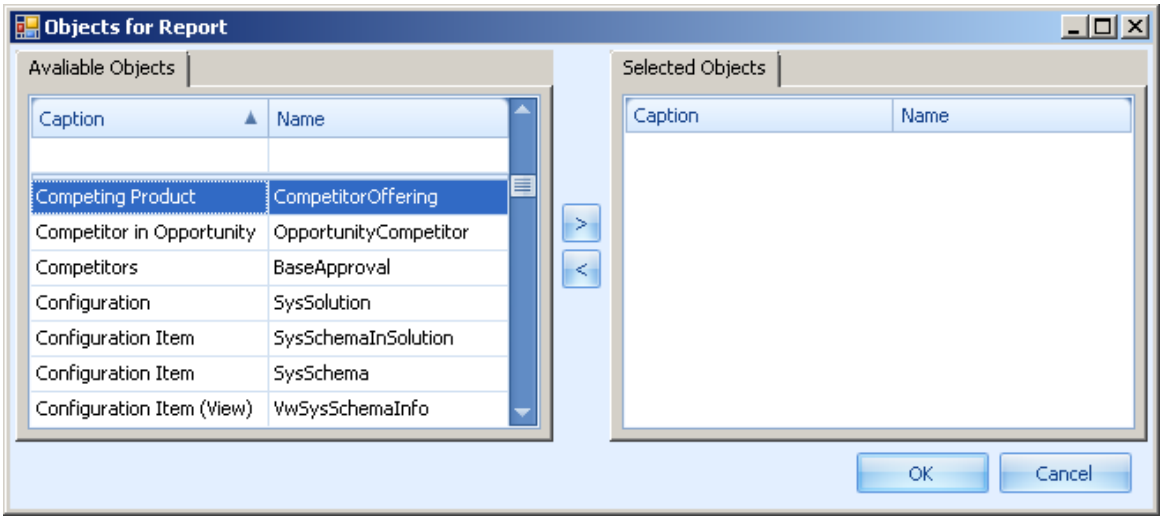
NOTE

Before start the [Select Object] command, select a custom bpm’online package in the [Package] report property. The report will be saved to this package. For example, this can be the [Custom] package.

The [Objects for Report] window

The [Objects for Report] window (Fig. 4) is used to add configuration object columns to the report data source.

Fig. 4. [Objects for Report] window

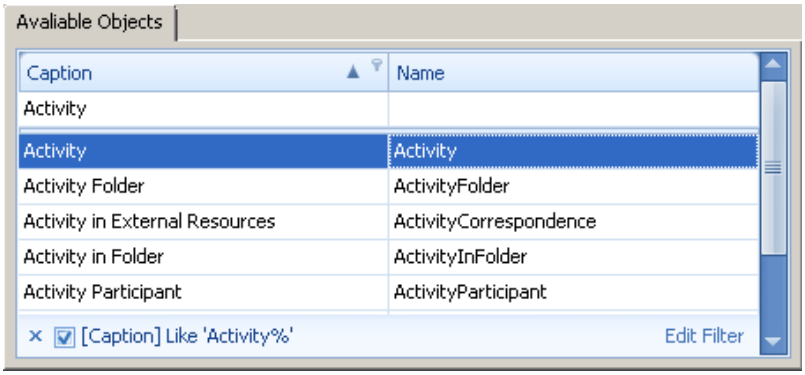


The [Available objects] area

The [Available objects] area contains a list of system objects, the information from which can be used to build the report.

The string under the [Title] and [Name] is used for search of the objects. Enter a part of the name or a title to filter objects (Fig. 5). Filter conditions that were applied to the area will be displayed in a separate string.

Fig. 5. the [Available objects] area with applied filter



To configure filter conditions click the [Edit filter] link. To disable filter without deleting filter conditions, disable the filter checkbox. Click the × button to clear the filter.

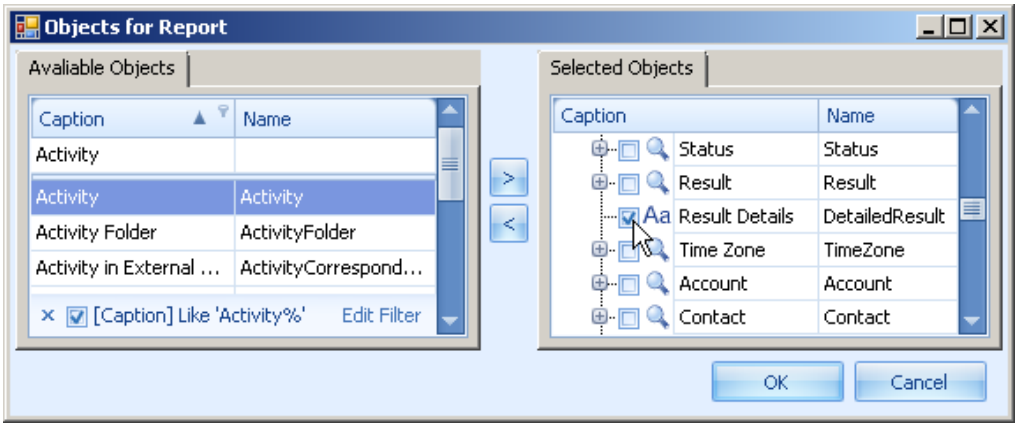
The [Selected objects] area

The [Selected objects] area displays structure of object columns that were added to the current report.

Click the > button to add an object to the [Selected objects] area or click the < to remove the object. For each object you need to specify columns to be added to the report data structure.

Enable the checkbox at the left of the object column to add it to the data source.

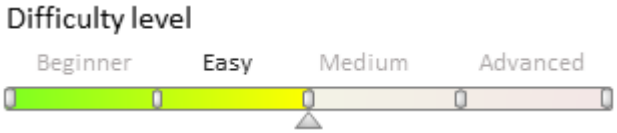
Fig. 6. Adding object column to the report data source



For example, if the report will contain names and main contact addresses, you need to add the [Contact] object and enable checkboxes for the [Name] and [Address] columns in the [Selected objects] area.

The data structure tree in the [Selected objects] area contains information of the connected data. If an object column connects it with other object, it will contain all structure of columns of the connected object and will be marked in the structure of the current object with the icon. For example, for the [Contact] object it will be the [Owner] column.

Report designer features



Introduction

Building a report includes two main components: creating a data report structure and a layout of the report page. Use the [Data] toolbar to create data source structure.

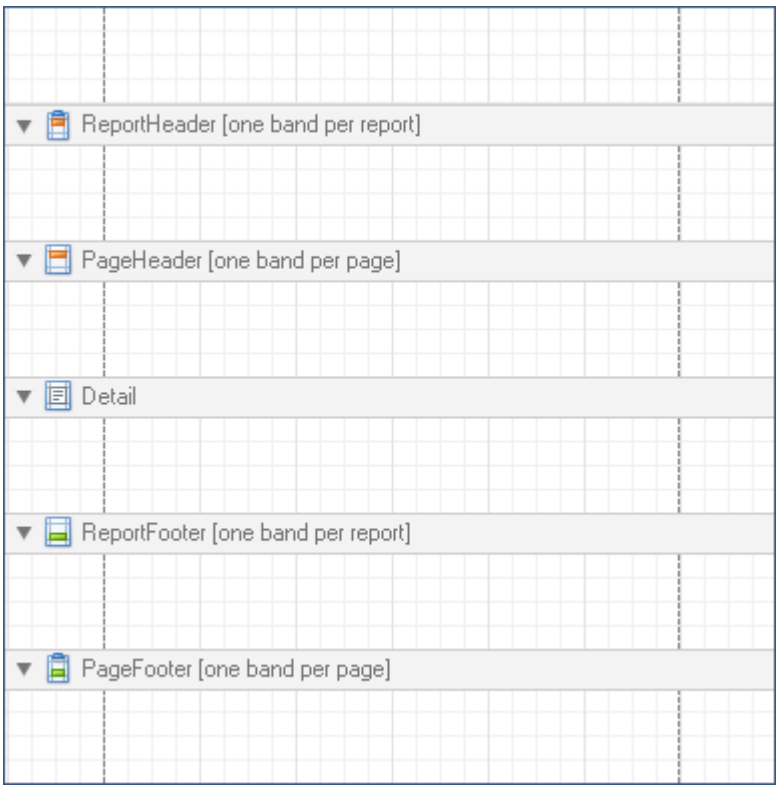
> ATTENTION
>
> When creating the report, the mechanism for filling out reports with data must be manually set by editing the data source of the report.

The page layout is created by adding lines, columns and other report elements to the workspace. Description of the functional elements of the report designer can be found in the “**Report designer workspace**” article.

Report page layout

Layout of the report page is performed with bands that correspond to the page report area and displays elements within it (Fig. 1). Elements placed on headers and footers are displayed on each page of the report and the elements placed on the [Report Header] band will be displayed only on the first page of the report.

Fig. 1. Report page layout example



NOTE

To add a line, right-click the workspace and click the [Insert Band] command. Then select the band type.

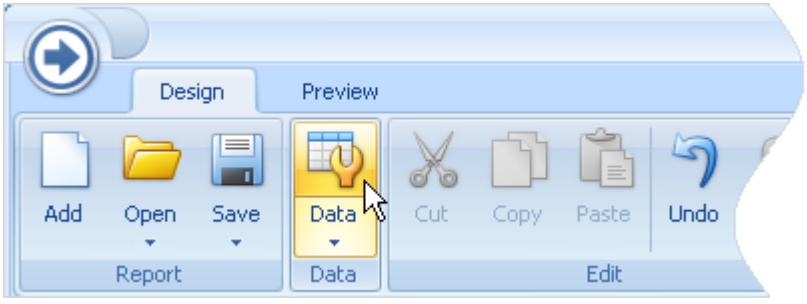
The [Detail] band is used to display a table in the report and is considered a table row template. One report can only have one [Detail] band.

To add more tables, place nested reports to the report. For example, to display a table with the account name and the account’s primary contact, place two columns on the [Detail] band.

Creating a data report structure

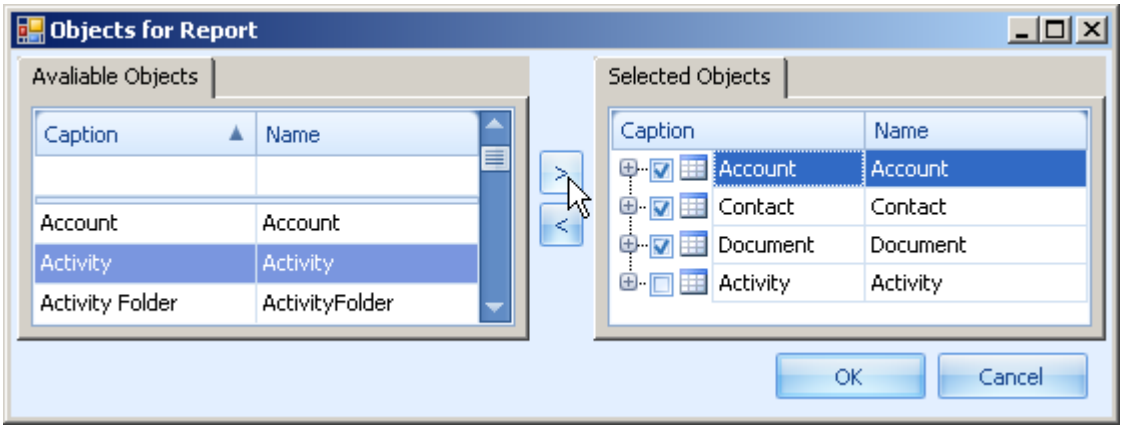
1. Click the [Data] button on the control panel (Fig. 2).

Fig. 2. Adding data to a report



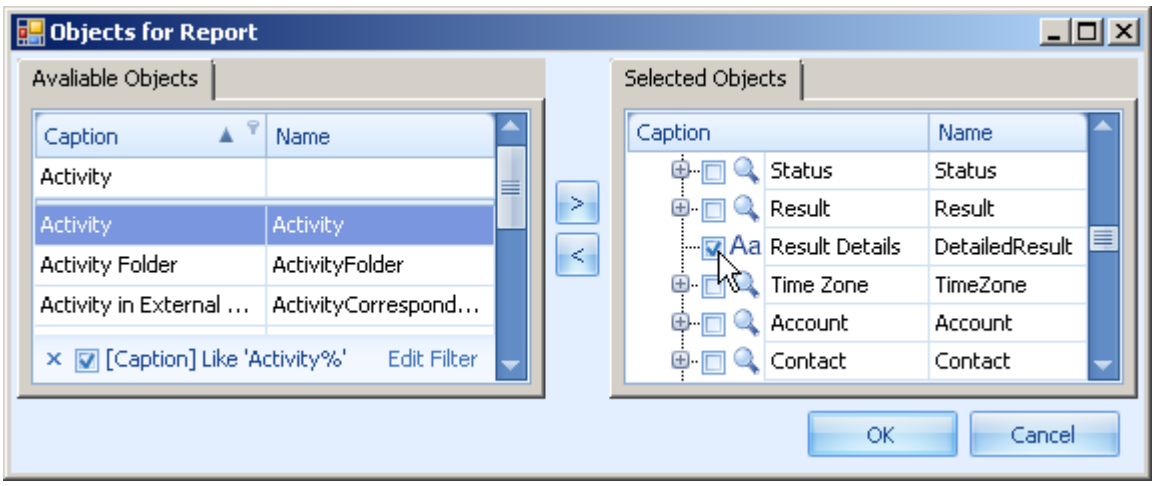
2. Add all objects the data of which will be used in the report to the [Selected objects] area, and enable checkboxes for them (Fig. 3).

Fig. 3. Adding objects to a report



3. Select object columns by enabling or disabling checkboxes on the [Selected objects] area.

Fig. 4. Selecting columns to use in the report



4. Click the [OK] button.

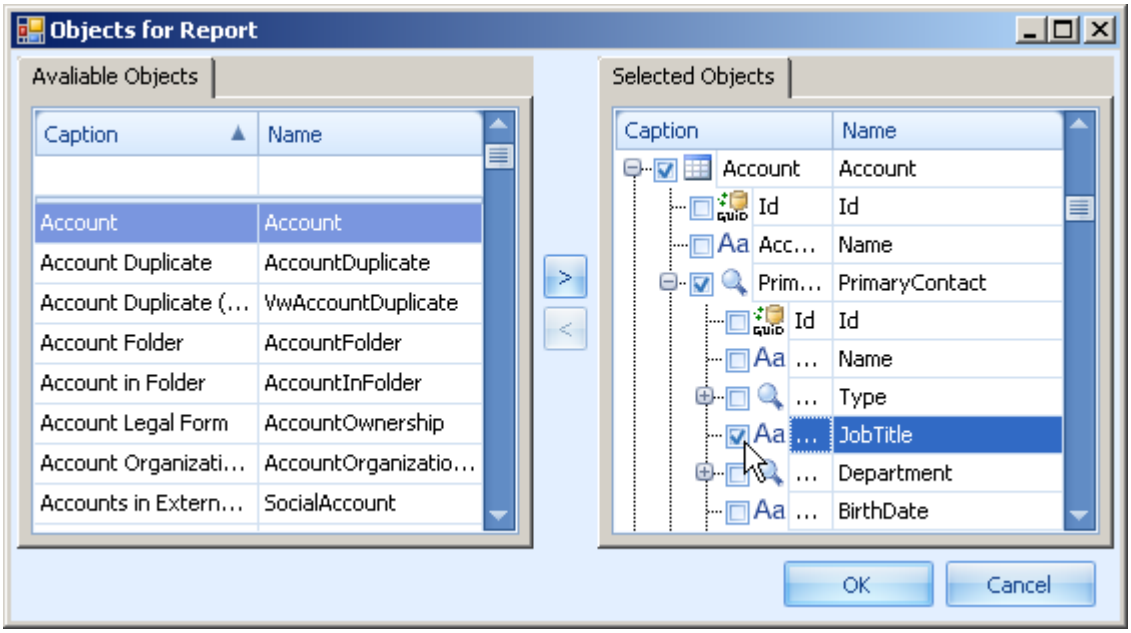
Added objects and their columns will be displayed in the [Data] window.

Adding connected data

You can add to the report columns of any available object and columns of connected objects.

For example, the [Contact] object is connected with the [Account] object via the [Primary contact] column. By adding the [Account] object to the [Selected objects] area, you can add columns of the [Contact] connected object (e.g., [Full name], [Email], etc.). (Fig. 5).

Fig. 5. Adding a connected object column to the report data source



ATTENTION

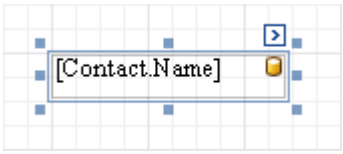
When you add connected data, enable the checkbox for the object the field of which you would like to use. Otherwise the added data will not be connected.

Columns of the connected object that were added this way will display the information connected with corresponding record of the main object. For example, the [Job] column will display the job of the contact connected with the account by the [Primary contact] column.

Adding data to the report page

Report data structure is displayed in the [Data] window. You can move the columns from the data area to the report designer workspace. Column moved to the workspace is a text object (Fig. 6).

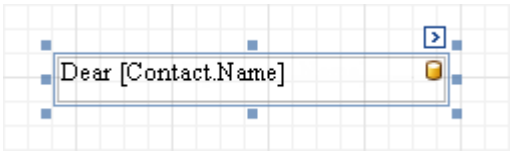
Fig. 6. Text object connected with the object column of the report data



While generating the report, the data from the corresponding database column will be displayed in the place of this object. For example, place a [Full name] column on the designer workspace to display a contact name in the report.

You can place a column on the other report text object. I this case, a variable displaying data of this column will be added to the text of the object (Fig. 7).

Fig. 7. Text object with the added variable



Editing the report data source

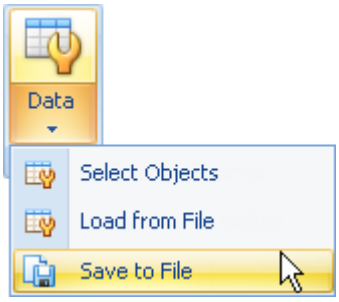
Edit the data source manually to fill the report with data from the database and for the advanced report configuration. For example, you can use columns that are not enabled in the system database in the report.

To edit the report data source in the third-party application, you need to save it in the separate file. To do this:

1. Add all necessary data from the system database to the report.

2. Save the report data structure in the .xsd file with the [Data] > [Save to file] menu command (Fig. 8).

Fig. 8. Saving the data source to the .xsd file

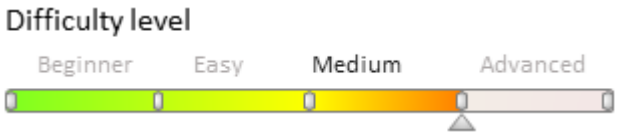


3. Open the .xsd file in the third-party application (for example, Microsoft Visual Studio) and preform the required modifications in it.

4. Load edited data source from the .xsd file to the report with the [Data] > [Load from file] menu command.

As a result, all changes made in .xsd file via the third-party application will be imported to the report.

Development in the file system



Introduction

Using an Integrated Development Environment (IDE) maximizes development speed. Examples of the IDE include Visual Studio, WebStorm and other tools. An IDE usually enables you to create, modify and compile the source code, debug it, run team development, use version control systems, etc. IDEs usually use text files stored in the file system to work with the source code.

> NOTE
>
> For development in the file system, you can use Microsoft Visual Studio Community, Professional and Enterprise version 2017 (with latest updates) and higher.

You can configure bpm’online **configuration packages** in the file system. With this mechanism, you can export the packages from the database to a set of files, edit the package content using an IDE and upload the updated packages back to the database. Using Visual Studio, you can debug custom source code of the schemas of the “Source code” (*SourceCodeSchema*) type.

Use bpm'online **built-in tools** if there is no need or possibility to develop in the file system.

Main limitations of development in the file system

When the development in the file system mode is enabled, a full-fledged development is supported only for schemas of the “Source code” (*SourceCodeSchema*) and “Client schema” (*ClientUnitSchema*) types.

For other package elements (such as resources and SQL scripts), the following rules are used:

- When exporting packages from the database to the file system, the package elements that are stored in the database will replace the corresponding items in the file system. The source code schemas and client schemas will not be replaced.
- When uploading packages to the database, source code schemas and client schemas will replace the corresponding items in the database. The application will keep using source code schemas and client schemas from the file system.

> ATTENTION

Starting with version 7.11.2, when the file system development mode is enabled, resources of these schemas are also saved in the file system after saving client schemas (*ClientUnitSchema*) and source code schemas (*SourceCodeSchema*) in the corresponding designers (see the **[Configuration] section**).

Integration with the version control system (SVN) with enabled development in the file system is performed with third-party tools. Bpm'online built-in mechanism of working with SVN is not used. It is still possible to install packages from the SVN repository in the [Configuration] section (this simplifies working with related packages). Use third-party utilities, such as TortoiseSVN to install separate packages.

NOTE

To use the built-in capabilities of working with the SVN repository, disable the development mode in the file system.

Application settings for development in the file system

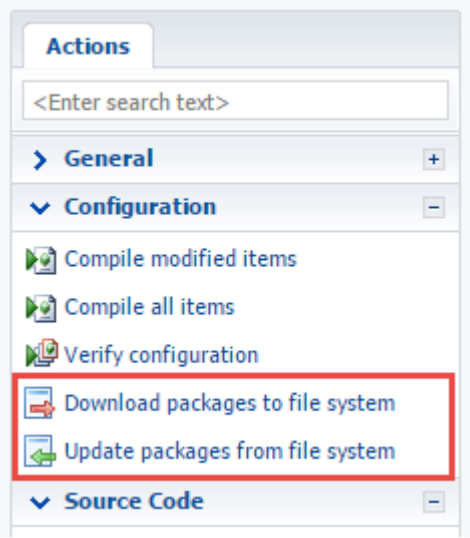
To enable development in the file system, edit the Web.config file (located in the root folder with the installed application) and set the *enabled* attribute of the *fileDesignMode* element to *true*.

```
<fileDesignMode enabled="true"/>
```

After enabling the development in the file system, two buttons will appear on the [Actions] tab in the [Configuration] section.

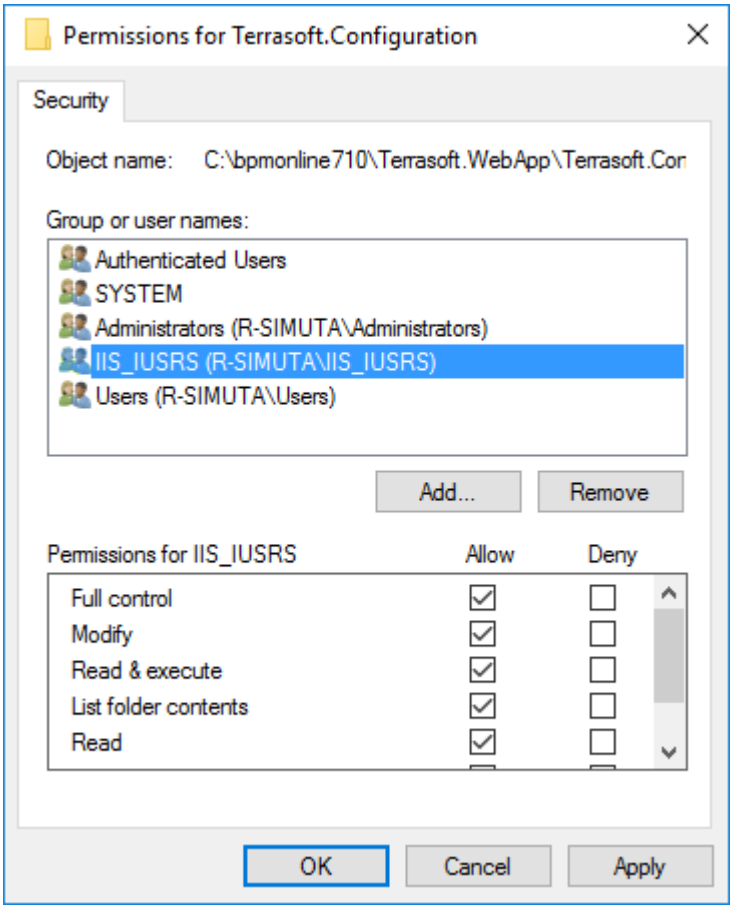
- [Download packages to file system] – exports the packages from the database to the following folder: *[path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg*.
- [Update packages from file system] – uploads the packages to the database from the following folder: *[path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg*.

Fig. 1 Actions of the [Configuration] section for development in the file system



To integrate the application with the configuration project, grant full access to the *[path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg* folder for the OS user, who runs the IIS application pool (Fig. 2). Usually, this is the built-in IIS_IUSRS user.

Fig. 2 Setting up access rights for the Terrasoft.Configuration folder



Terrasoft.Configuration package

A configuration project is a Visual Studio solution supplied with bpm'online setup files. The solution can be found here: *[path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration*.

To start development in the file system, open the following file in Visual Studio: *[path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Terrasoft.Configuration.sln*.

The configuration project structure is available in table 1.

Table 1. Configuration project structure

Folder	Purpose
Lib	The folder where package-bound third-party class libraries are exported.
Autogenerated\Src	The folder with files that contain auto-generated source code of the preset package schemas. These files cannot be edited.
Pkg	The folder where the packages for development in the file system are exported from the database.
bin	A folder for compiled configuration and third-party libraries.

Getting Started with the configuration

Creating a package

If you do not intend using SVN in the development process, then the process of creating a package is the file system development mode is the same as that in the normal mode. For more information on creating packages please refer to the **Creating and installing a package for development** article.

Attention!

The working with SVN mode is enabled in the bpm'online by default. If the [Version control system repository] field is empty when creating a package, then the package will not be bound to the repository. The versioned development of this package can be performed only after manually binding it to the repository from the file system.

More information about creating a custom package and binding it to the SVN repository can be found in the "**Creating a package in the file system development mode**” article.

Working with new package elements

It is recommended to add new elements (schema or resource) to the package only from the **[Configuration] section**. To create and edit a new item in a custom package:

- Select a custom package in the [Configuration] section and add a new element in it (see **Creating a custom client module schema**, “**Creating the [Source code] schema**”).
- Add resources (such as localized strings) to the schema if needed.
- Click [Download packages to file system] (Fig. 1).
- Use an IDE (such as Visual Studio) to edit the source code of the schema or localized resource in the files (located in the [Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\ [Package name] folder). The package properties are described in the "**Package structure and contents**” article.
- Click [Update packages from file system] to upload changes to the application database (Fig. 1).

Attention!

Changes made in client schemas are available in the application immediately, without uploading to the database. You only need to update the page in the browser.

- If you changed a source code schema, then you must compile the application.

NOTE

When developing source code schemas in C #, compile them directly in Visual Studio. More information about compilation and debugging in Visual Studio can be found in the "**Working with the server side source code in Visual Studio**” article.

See also

- **Visual Studio settings for development in the file system**
- **Working with the client code in the file system**
- **Working with the server side source code in Visual Studio**
- **Developing the configuration server code in the user project**
- **Automatic displaying of changes in the development of the custom logic**
- **Working with SVN in the file system**
- **Packages file content**
- **Localization of the file content**
- **How to create Unit-tests via NUnit and Visual Studio**
- **How to use TypeScript when developing custom functions**

Visual Studio settings for development in the file system

Difficulty level

Introduction

Using an Integrated Development Environment (IDE), Microsoft Visual Studio maximizes development speed. Microsoft Visual Studio IDE usually enables you to create, modify and compile the source code, debug it, run team development, use version control systems, etc.

Development in Microsoft Visual Studio became possible after the implementation of the **configuration packages in the file system** mechanism in bpm'online. With this mechanism, you can export the packages from the database to a set of files, edit the package source code using an IDE and upload the updated packages back to the database.

NOTE

For development in the file system, you can use Microsoft Visual Studio Community, Professional and Enterprise version 2017 (with latest updates) and higher.

The WorkspaceConsole utility integrated into Visual Studio is used to compile applications. The WorkspaceConsole has the following benefits:

- Significantly speeds up the compilation process, because the whole configuration assembly is split into independently compiled modules. Only the modules that contain modified packages are compiled.
- Compilation does not require exiting the debugging mode or disconnecting from the IIS process.

Attention!

You can also compile the configuration project using the Visual Studio compiler. However, it may not take into account the dependencies and the position of packages.

Visual Studio settings for development in the file system:

1. Enable compilation mode in the IDE.
2. Configure the WorkspaceConsole to compile the application.
3. Configure Microsoft Visual Studio.

Visual Studio configuration steps

1. Enable compilation mode in the IDE

To enable compilation mode in the IDE, edit the Web.config file (located in the root folder with the installed application) and set the *enabled* attribute of the *fileDesignMode* element to *true*.

```
<fileDesignMode enabled="true" />
```

Attention!

Enable the **development mode in the file system** to compile in the Visual Studio.

Attention!

Currently, the development in the file system is no compatible with getting client content from preliminary generated files. For the correct work of the development in the file system you need to disable getting static client content from the file system. Set the “false” for the *UseStaticFileContent* flag in the Web.config file to disable this functions.

```
<fileDesignMode enabled="true" />
...
<add key="UseStaticFileContent" value="false" />
```

Attention!

If the development mode in the file system is enabled, IIS will use the *Terrasoft.Configuration.dll* library only from the file system.

After switching to the file system development mode for the first time, upon logging in, the user is redirected to the "Configuration" section. At this time, "The "Default" workspace assembly is not initialized" error appears. To eliminate this error, run the "Compile all items" action.

2. Configure the WorkspaceConsole to compile the application.

Configuration projects are compiled via the **WorkspaceConsole** utility, which is included in the application setup files. The utility should be configured before using. In addition to the settings covered in the "**WorkspaceConsole settings**” article, you must also enable the development mode in the file system in the Terrasoft.Tools.WorkspaceConsole.exe.config configuration file.

```
<fileDesignMode enabled="true" />
```

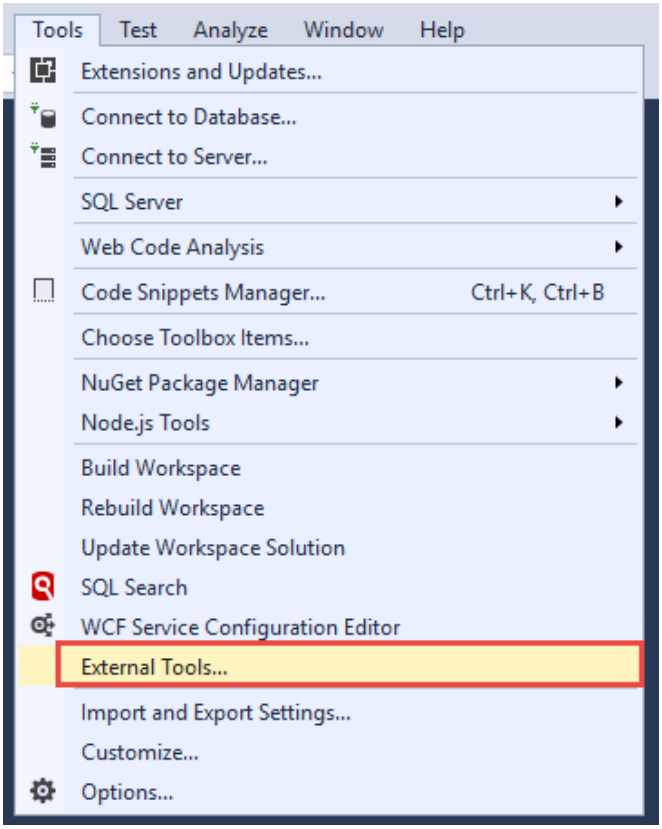
To speed up the compilation by splitting the configuration assembly into independent compiled modules, set the *CompileByManagerDependencies* setting to "*true*" in the "internal" Web.Config (located in the Terrasoft.WebApp directory) and in the Terrasoft.Tools.WorkspaceConsole.exe.config file of the WorkspaceConsole utility.

```
<appSettings>
    ...
    <add key="CompileByManagerDependencies" value="true" />
    ...
 </appSettings>
```

3. Configure Microsoft Visual Studio

To use the WorkspaceConsole utility for compilation in Visual Studio, set up *External Tools*. To do this, execute the Tools > External Tools... command in the Visual Studio (Fig. 1).

Fig. 1 Adding external tools in the Visual Studio



In the opened dialog window (Fig 2), add and set up three commands for calling the *WorkspaceConsole* utility. The *Build Workspace* and *Rebuild Workspace* commands initiate compilation of changes and full compilation of configuration projects. The *Update Workspace Solution* command updates the Visual Studio solution of the configuration package from the application database. It applies all changes made by the users within the application. The properties and arguments of commands are available in tables 1, 2 and 3. Select the *Use Output window* checkbox (Fig. 2) for all three commands.

Fig. 2 Setting properties and arguments for an external Visual Studio command

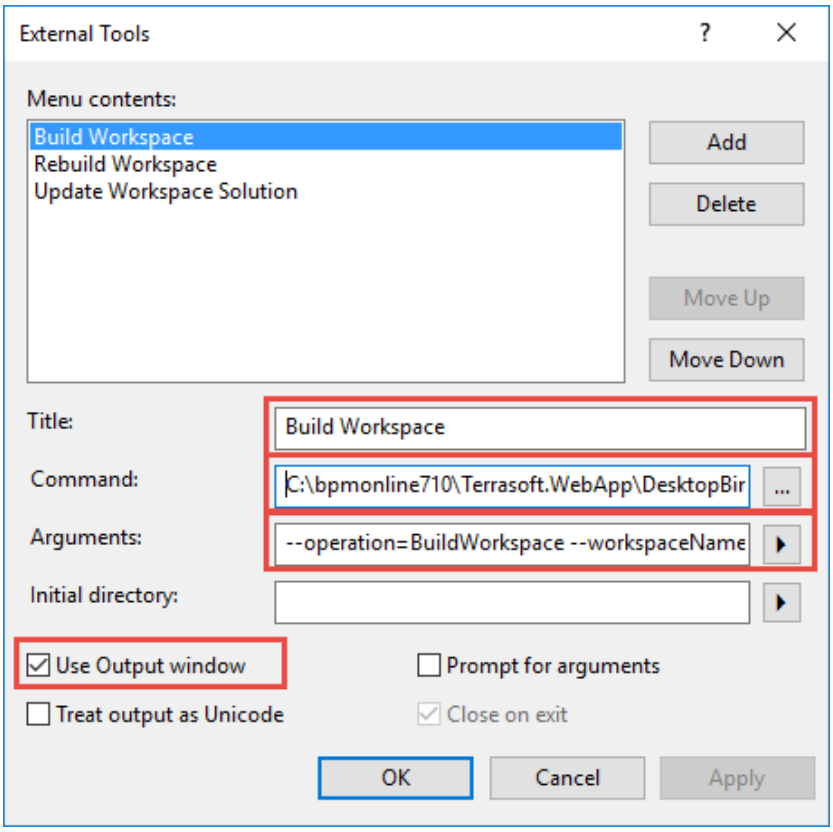


Table 1. Update Workspace Solution command properties

Title	Update Workspace Solution
Command	[Path to the catalog with installed application]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe Example: C:\bpmonline710\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe
Arguments	--operation=UpdateWorkspaceSolution --workspaceName=Default --webApplicationPath="[Path to the catalog with installed application]\Terrasoft.WebApp\" Example: --operation=UpdateWorkspaceSolution --workspaceName=Default --webApplicationPath="C:\bpmonline710\Terrasoft.WebApp\"

Table 2. Build Workspace command properties

Title	Build Workspace
Command	[Path to the catalog with installed application]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe Example: C:\bpmonline710\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe
Arguments	--operation=BuildWorkspace --workspaceName=Default --webApplicationPath="[Path to the catalog with installed application]\Terrasoft.WebApp\" Example: --operation=BuildWorkspace --workspaceName=Default --webApplicationPath="C:\bpmonline710\Terrasoft.WebApp\"

Table 3. Rebuild Workspace command properties

Title	Rebuild Workspace

Command [Path to the catalog with installed application]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe

Example:

C:\bpmonline710\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe

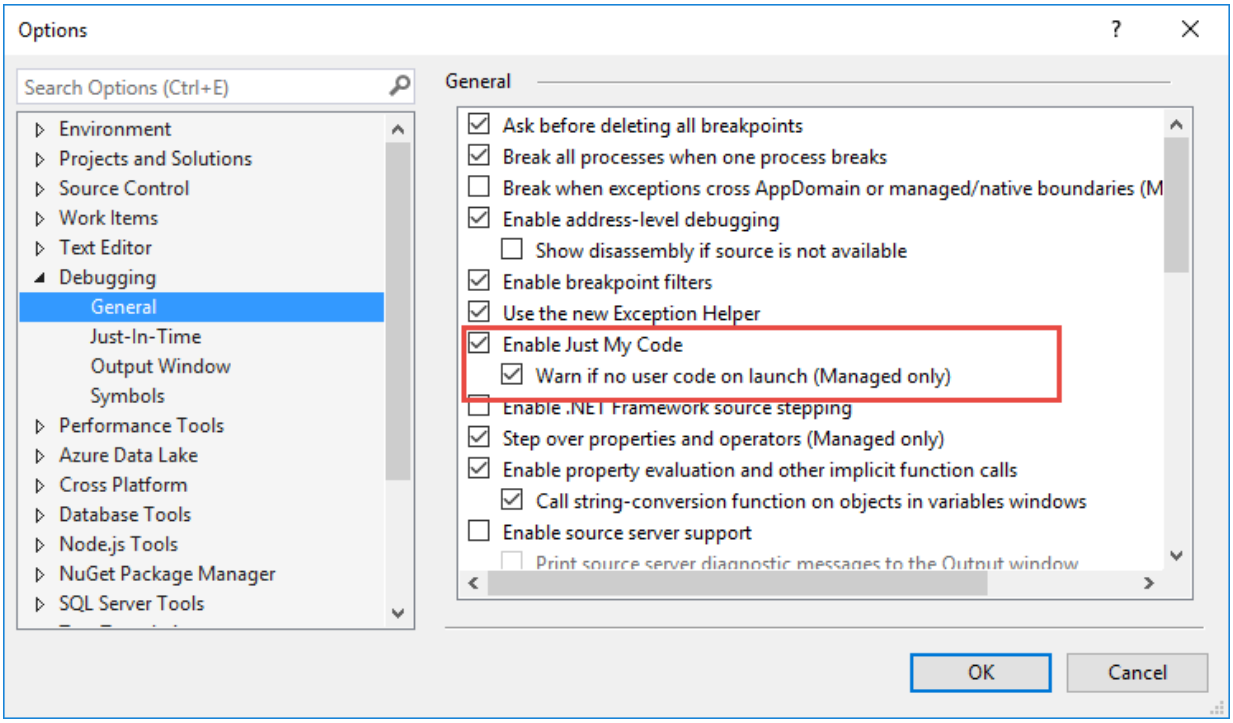
Arguments --operation=RebuildWorkspace --workspaceName=Default --webApplicationPath="[Path to the catalog with installed application]\Terrasoft.WebApp\"

Example:

--operation=RebuildWorkspace --workspaceName=Default --webApplicationPath="C:\bpmonline710\Terrasoft.WebApp\"

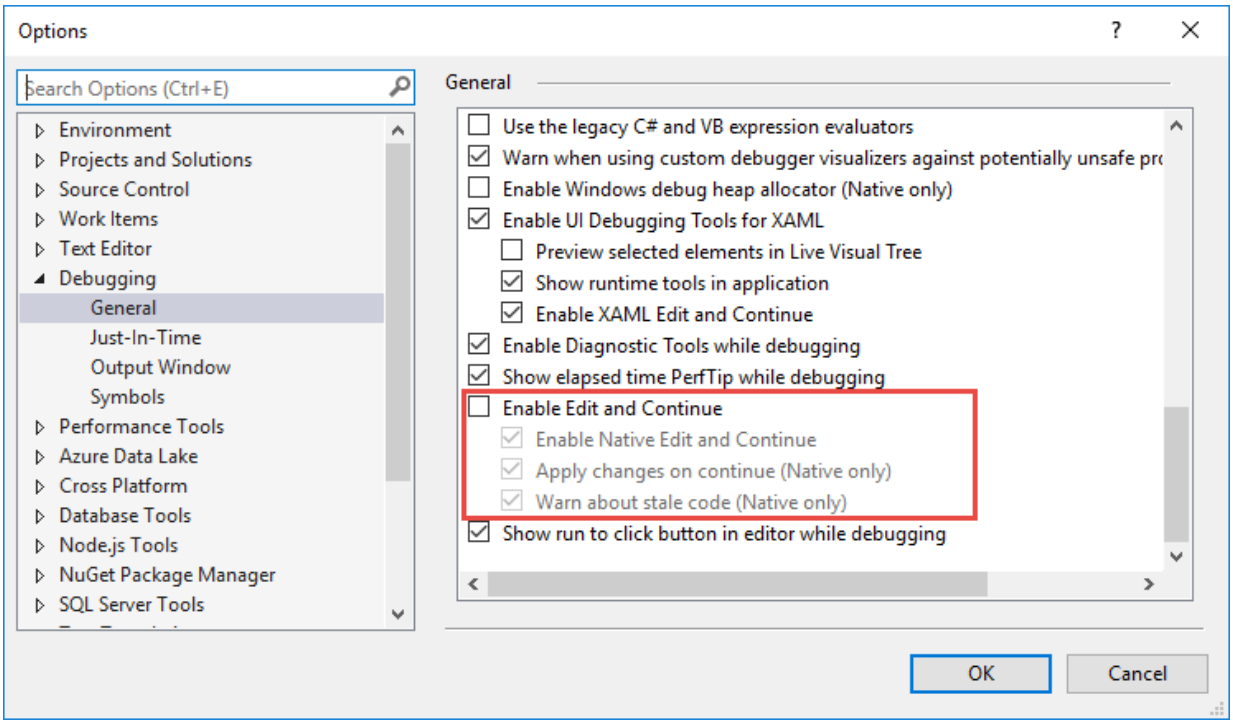
To prevent the debugger from accessing the source code that is disabled in the project, execute the Debug > Options... menu command and enable the *Enable Just My Code* option in the opened dialog (Fig. 3). For more information about the *Enable Just My Code*, please refer to this page.

Fig. 3 Enable Just My Code option



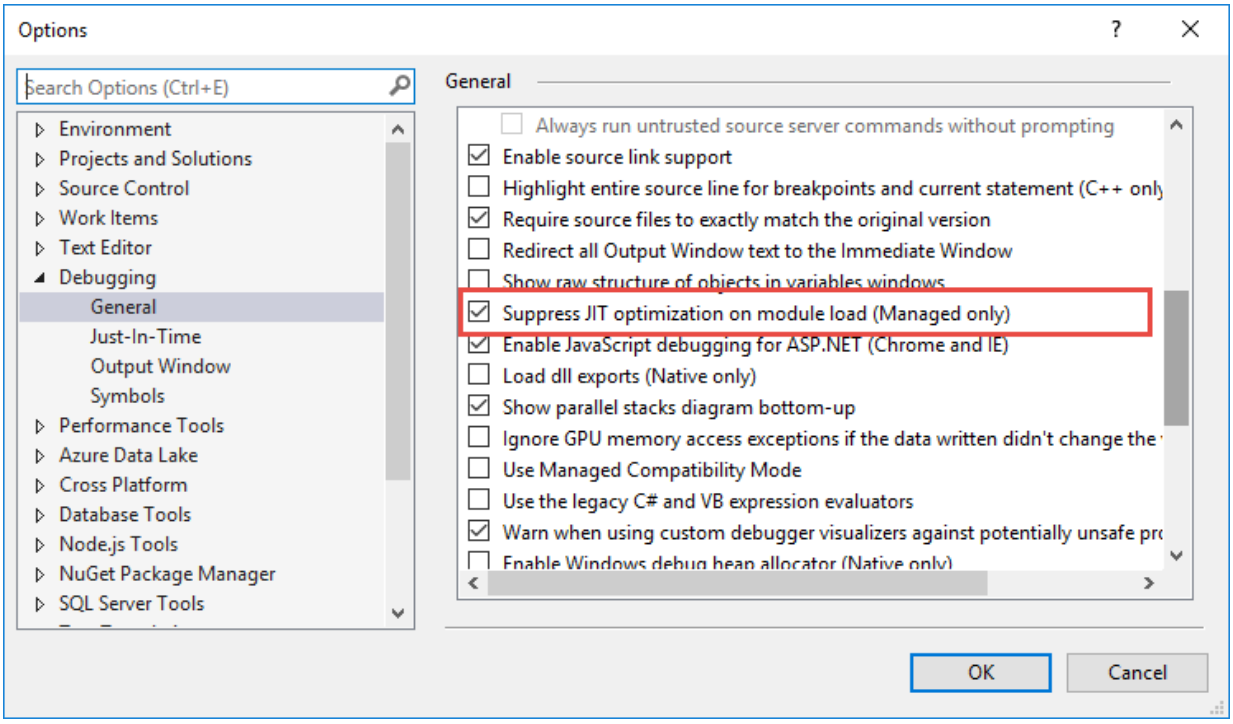
After the configuration is compiled, the application is automatically restarted. The *Enable Edit and Continue* option is not supported and should be disabled (Fig. 4).

Fig. 4 Edit and Continue option



For the debugger to stop correctly on breakpoints, make sure that the *Suppress JIT optimization on module load* option is enabled (Fig. 5).

Fig. 5 Suppress JIT optimization on module load



Working with the server side source code in Visual Studio

Difficulty level

Beginner Easy Medium Advanced

Introduction

Bpm'online has the ability to debug program code using the integrated functions of the Visual Studio development environment. The Visual Studio debugger enables developers to freeze the execution of program methods, check variable values, modify them and monitor other activities performed by the program code.

The general procedure for bpm’online development in Visual Studio is as follows:

1. Perform preliminary settings in bpm'online and Visual Studio.
2. Create, obtain or update a package from the SVN repository.
3. Create a [Source code] schema for development.
4. Perform development in Visual studio.
5. Save, compile and debug the source code.

ATTENTION

After successful compilation, the resulting *Terrasoft.Configuration.dll* assembly will be placed to the Bin catalog, while IIS will automatically use it in bpm'online application.

General procedures for developing bpm'online solutions in Visual Studio

1. Perform preliminary settings

Bpm'online and Visual Studio setup for development in the file system is described in the “**Development in the file system**” and “**Visual Studio settings for development in the file system**” articles.

NOTE

For development in the file system, you can use Microsoft Visual Studio Community, Professional and Enterprise version 2017 (with latest updates) and higher.

2. Create, obtain or update a package from the SVN repository

Creating a custom package with or without SVN is described in the “**Creating and installing a package for development**” and “**Creating a package in the file system development mode**” articles. Installing and updating packages is described in the “**Installing packages from repository**” and “**Updating package from repository**” articles.

NOTE

We recommend using Tortoise SVN or Git for working with version control repositories.

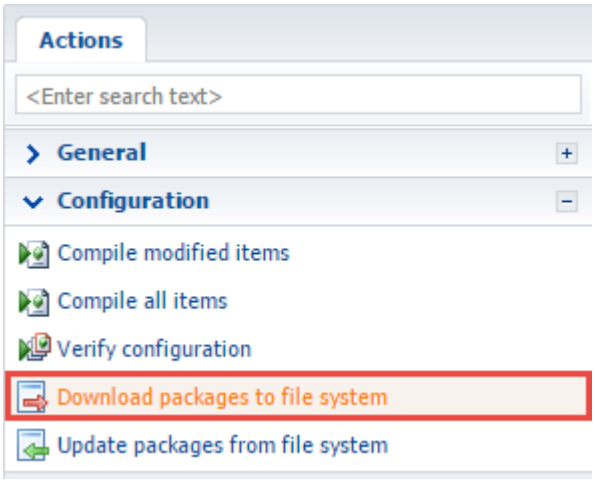
3. Create a [Source code] schema

Learn more about the process of creating the [Source code] schema in the “**Creating the [Source code] schema**” article.

4. Conduct development in the Visual Studio

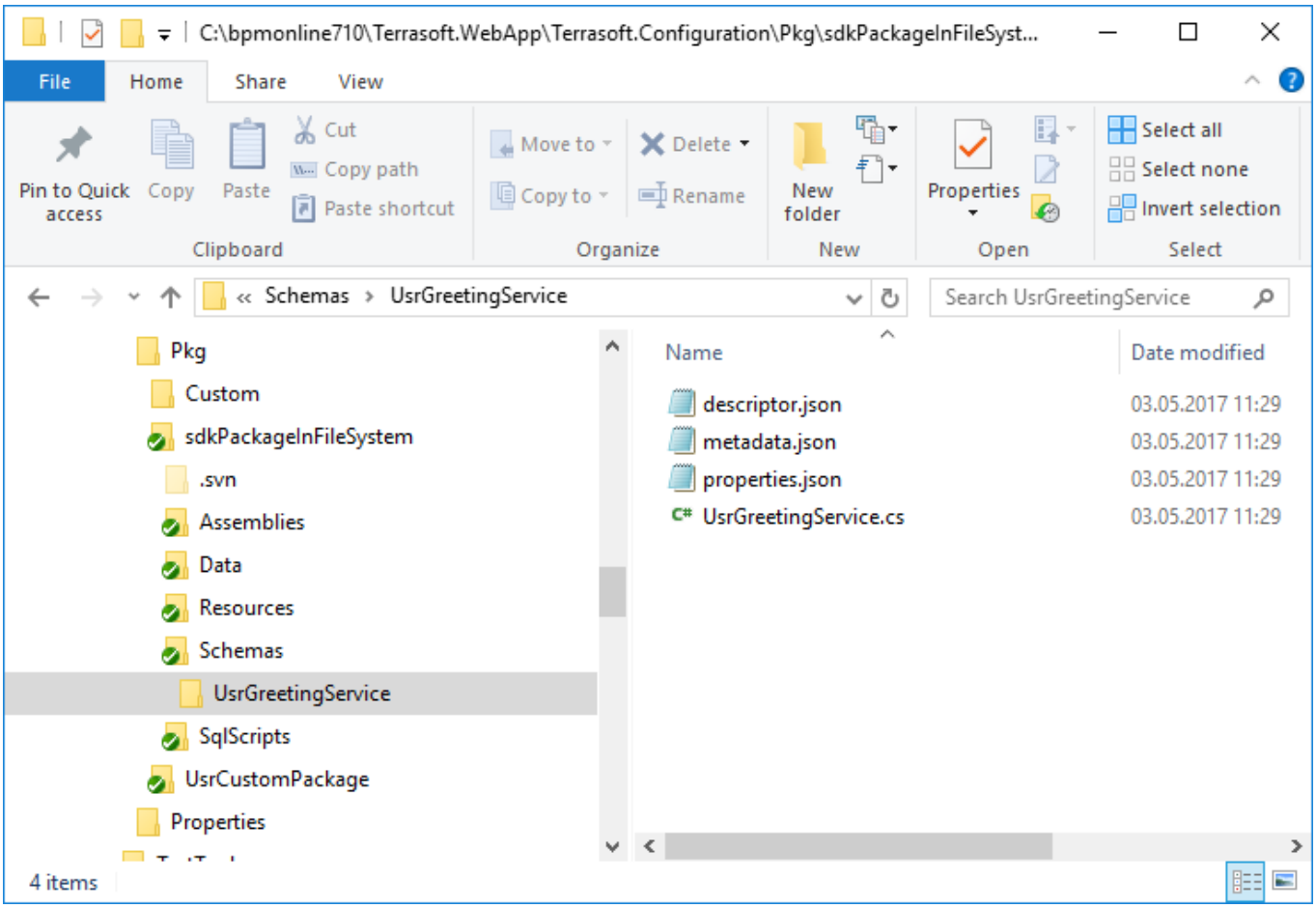
Before you start development in Visual Studio, make sure that you export existing schemas from the database to the file system.

Fig. 1 The [Download packages to file system] action



For example, if the [Source code] schema with the name *UsrGreetingService* was created in the *sdkPackageInFileSystem* package, the file of the source code of the *UsrGreetingService.cs* schema appears in the file system in the *Pkg\sdkPackageInFileSystem\Schemas* directory (Fig. 2). In this case, the system generated *UsrGreetingServiceSchema.sdkPackageInFileSystem_Entity.cs* file will be placed in the *Autogenerated\Src* directory.

Fig. 2 The source code schema file



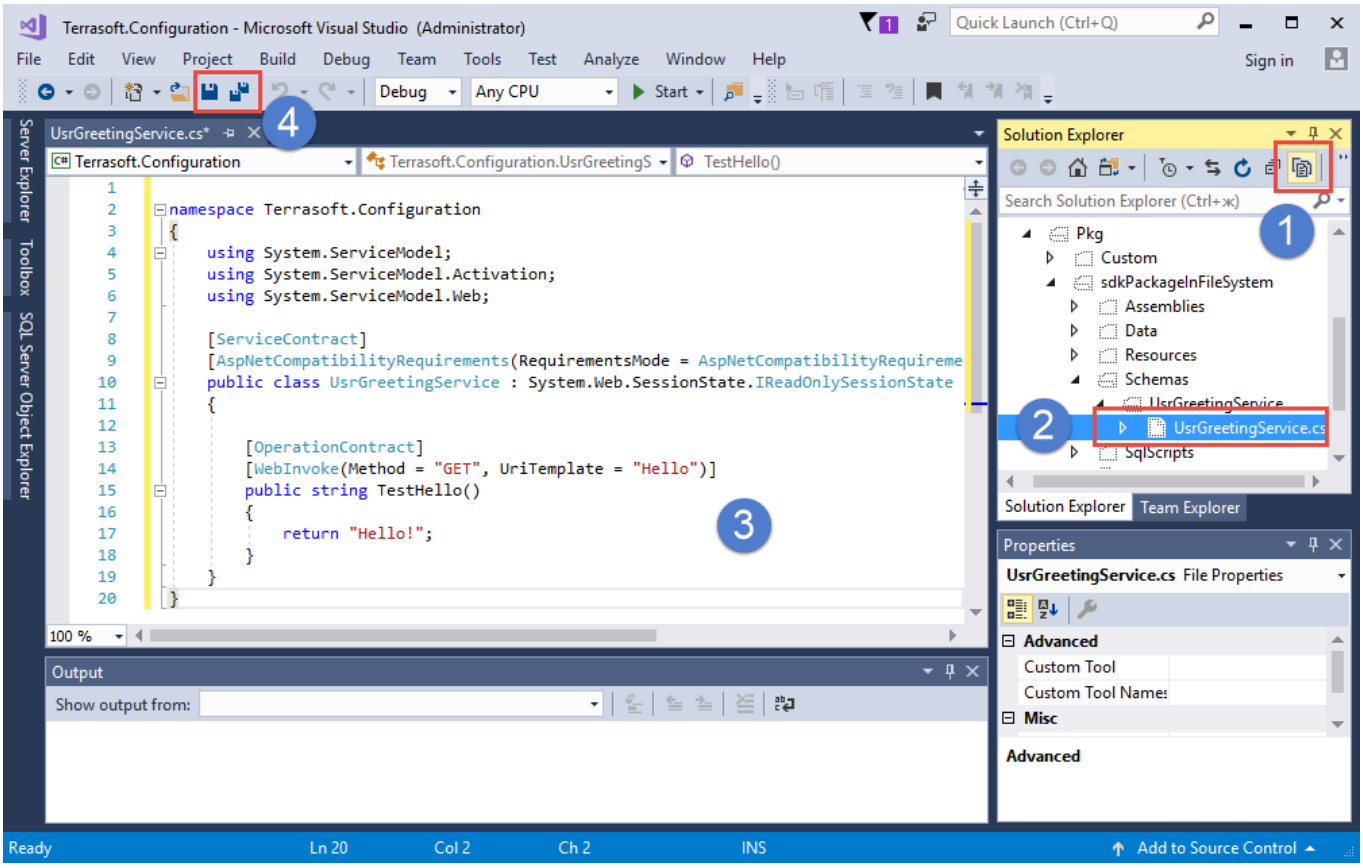
NOTE

To add a schema to SVN, you must add the entire *UsrGreetingService* directory, including the JSON files.

Open the *Terrasoft.Configuration.sln* solution in Visual Studio to start the development (see “**Development in the file system**”). In Visual Studio Solutions Explorer, enable the display of all file types (Fig. 3, 1), open the

UsrGreetingService.cs file (Fig. 3, 2) and add the required source code (Fig. 3, 3).

Fig. 3 Working with the schema file in Visual Studio



Below is an example of source code implementation, which must be added to the contents of the *UsrServiceGreeting.cs* file, using Visual Studio:

```
namespace Terrasoft.Configuration
{
    using System.ServiceModel;
    using System.ServiceModel.Activation;
    using System.ServiceModel.Web;
    // Class that implements configuration service.
    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class UsrGreetingService : System.Web.SessionState.IReadOnlySessionState
    {
        // Service operation.
        [OperationContract]
        [WebInvoke(Method = "GET", UriTemplate = "Hello")]
        public string TestHello()
        {
            return "Hello!";
        }
    }
}
```

For more information about the purpose of the attributes of the class that implements configuration services, please refer to the "**How to create custom configuration service**".

5. Save, compile and debug source code

After modifying the source code, and before compiling and debugging it, be sure to save the code. Normally, this is

done by Visual Studio automatically, but since Visual Studio compiler is not used, the developer must save the code manually.

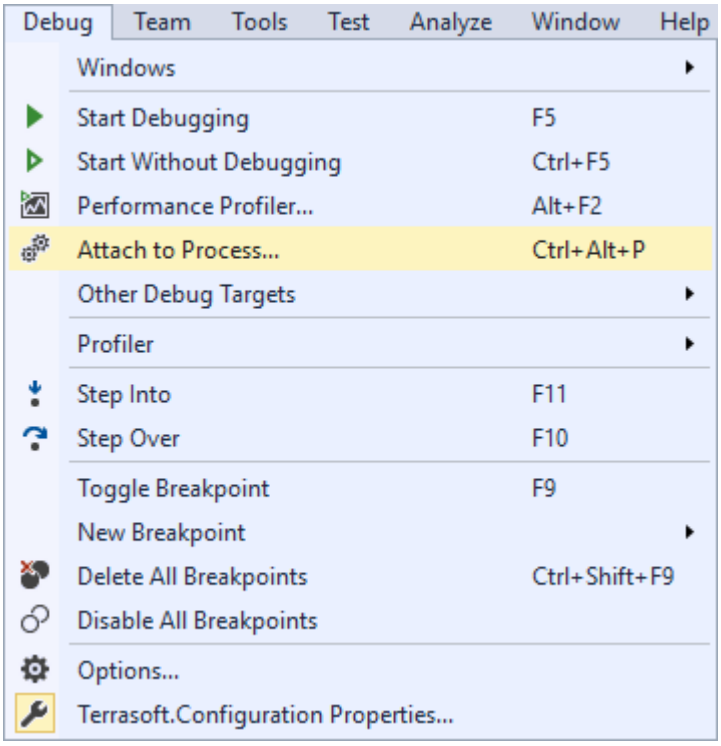
After saving, the source code must be compiled with the "Build Workspace" or "Rebuild Workspace" commands (see "**Visual Studio settings for development in the file system**"). If the compilation is successful, the code becomes available. In the example described previously, the service will become available at the following address (Fig. 4):

```
http://[Application URL]/0/rest/UsrGreetingService/Hello
```

Fig. 4 Checking service workability

To begin debugging, attach to the IIS server process, where the application runs. Execute the *Debug > Attach to process* menu command (Fig. 5).

Fig. 5 The [Attach to process] command



In the opened window, select the working IIS process in the list of processes, where the application pool is running (Fig. 6).

Fig. 6 Attaching to an IIS process

NOTE

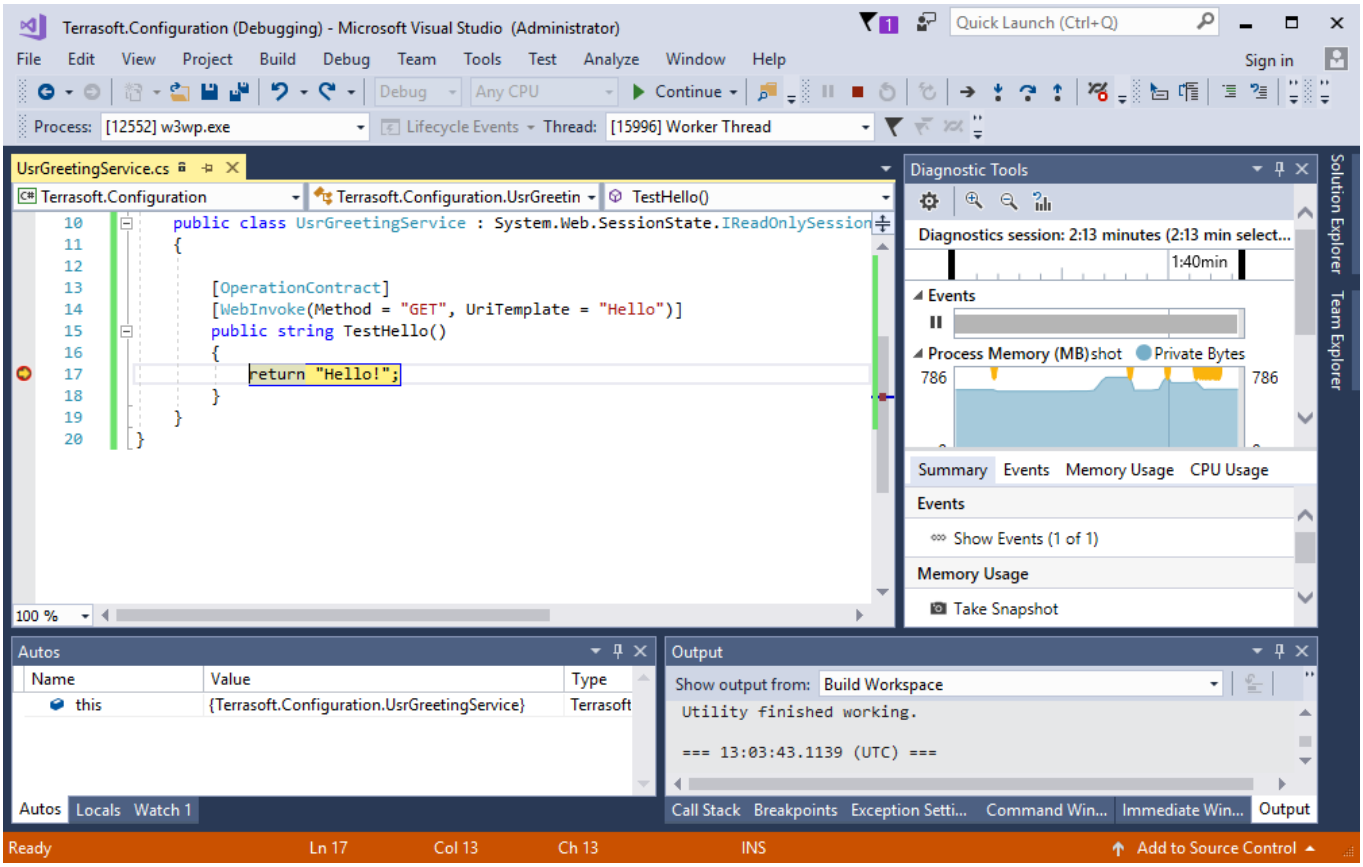
The name of the working process can be different, depending on the configuration of the IIS server being used. With a regular IIS, the process is *w3wp.exe*, but with IIS Express, the process name is *iisexpress.exe*.

By default, the IIS working process is run under the account whose name matches the name of the application pool. To display processes of all users, set *Show processes from all users* checkbox (Fig. 6).

After attaching to a working IIS process, execute compilation one more time. After that, begin the debugging process using the Visual Studio debugger. For example, you can set the stop points, view variable values, call stacks, etc. For more information on the Visual Studio debugger, please refer to the corresponding documentation.

For example, after setting up the breakpoint on the return line from the *TestHello()* method, and re-compiling the application and executing the service request, the debugger will stop the program execution on the breakpoint (Fig. 7).

Fig. 7 Stopping an application on the breakpoint



ATTENTION

The debugging feature depends on the **correct configuration of Visual Studio**.

Working with the client code in the file system

Difficulty level

Beginner Easy Medium Advanced

Introduction

The updated method of working with the client code in the file system enables better development flexibility. Download the client schema source code from the database to *.js files and LESS module styles into *.less files for working with them in the Integrated Development Environment (IDE) (e.g. WebStorm, Visual Studio Code, Sublime

Text, etc.).

General outline:

1. Pre-configure bpm'online.
2. Create, obtain or update a package from the SVN repository.
3. Create a [Source code] schema.
4. Save the database content to the file system.
5. Carry out the development of the schema source code in IDE.
6. Save, compile and debug the source code.

General outline:

1. Pre-configure bpm'online

Setting up bpm’online for development in the file system is described in the “**Development in the file system**” article.

ATTENTION

The *UseFileContent* attribute of the *clientUnits* element in the Web.config file (located in the application folder) was used to upload the client module source code to the file system up until bpm’online version 7.10.0. Since version 7.10.0, the *UseFileContent* attribute has been removed from the Web.Config file. Use the method described in the “**Development in the file system**” to upload the client module source code from pre-installed packages (the *Autogenerated\Src* folder description).

2. Create, obtain or update a package from the SVN repository

Creating a custom package with or without SVN is described in the “**Creating and installing a package for development**" and “**Creating a package in the file system development mode**” articles. Installing and updating packages is described in the “**Installing packages from repository**” and “**Updating package from repository**” articles.

NOTE

We recommend using Tortoise SVN or Git to work with version control repositories.

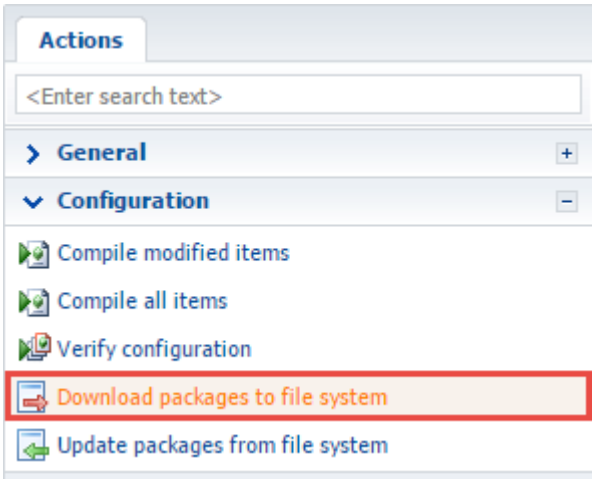
3. Create a custom schema for development

Learn more about custom schemas in the “**Creating a custom client module schema**” article.

4. Upload the schema from the database to the file system

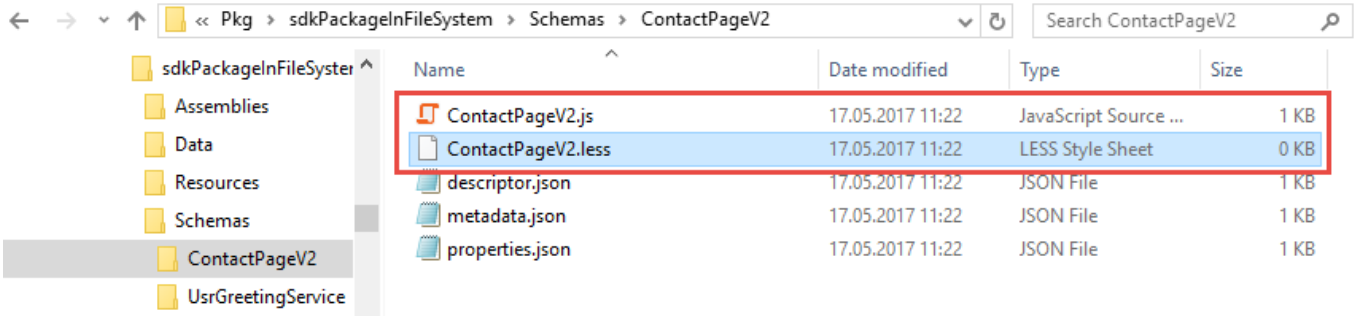
To do this, use the [Download packages to file system] command (Fig. 1) in the [Configuration] section.

Fig. 1. The [Download packages to file system] command



For example, if you created a replacing *ContactPageV2* schema ([Display schema - Contact card]) in a custom *sdkPackageInFileSystem* package, the files in the *Pkg\sdkPackageInFileSystem\Schemas*ContactPageV2 folder will contain the source code files of the *ContactPageV2.js* schema and *ContactPageV2.less* styles (Fig. 2).

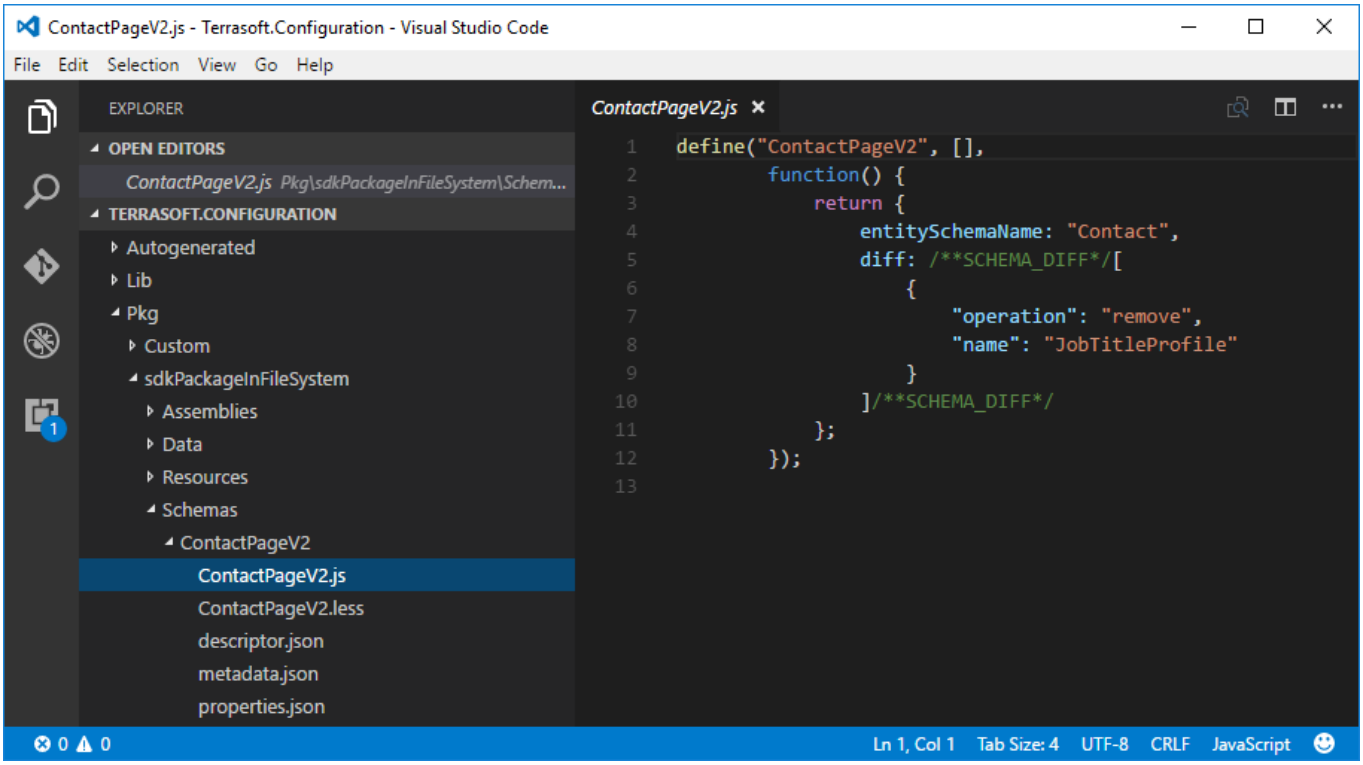
Fig. 2. The source code schema file



5. Carry out the development of the schema source code in IDE

To perform the development, open the file with the schema source code in the preferred IDE (or any text editor) and add the necessary source code (Fig. 3).

Fig. 3. Editing a schema file in Visual Studio Code



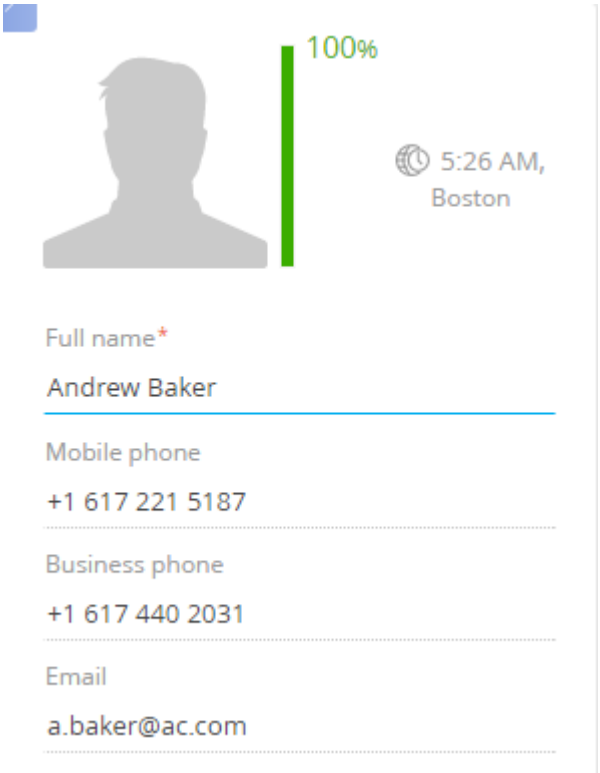
For example, add the following source code to the *ContactPageV2.js* file to hide the [Full job title] field from the contact edit page:

```
define("ContactPageV2", [],
        function() {
            return {
                entitySchemaName: "Contact",
                diff: /**SCHEMA_DIFF*/[
                    {
                        "operation": "remove",
                        "name": "JobTitleProfile"
                    }
                ]/**SCHEMA_DIFF*/
            };
        });
```

6. Save, compile and debug the source code

The [Full job title] will be removed from the contact edit page upon saving the *ContactPageV2.js* file and refreshing the page (Fig. 4).

Fig. 4. Contact page without the [Full job title] field



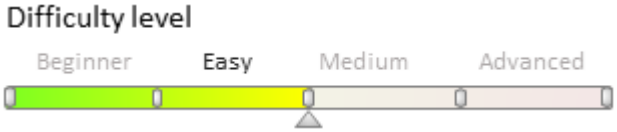
Debug the code if you encounter any errors (see: “**Client code debugging**”).

ATTENTION

To return to built-in bpm’online development tools, do the following:

1. Update packages from file system.
2. Disable the file system development mode by setting the *enabled="false"* attribute of the *fileDesignMode* element of the Web.config configuration file (see “**Development in the file system**”).

Working with SVN in the file system



Introduction

Subversion (SVN) is a centralized system designed for collaborative work. It is based on a repository that contains data in form of a “tree” hierarchy of files and folders. Users can connect to the repository to browse, view or modify files. Their modifications are available to all other users, and vice versa.

All modifications are documented in SVN, including the information about added, deleted, and moved files and folders. Users can access SVN files and folders at any given moment. Additionally, they can view all other versions of all files and folders.

Learn more about configuring and using Subversion in this article.

General outline:

Repository – a central database, usually located on a file server that contains versioned files with their full history. A repository can be accessed through various network protocols or from a local disk.

Working copy – a folder located on the developer’s computer. A developer can get the latest version of the files from the repository, work with them locally, and commit these files back to the repository when they are done. A working copy does not include the project’s history, but does contain the copies of files that were located in the repository prior to any changes made by the developer. This enables complete visibility over any changes made to the files.

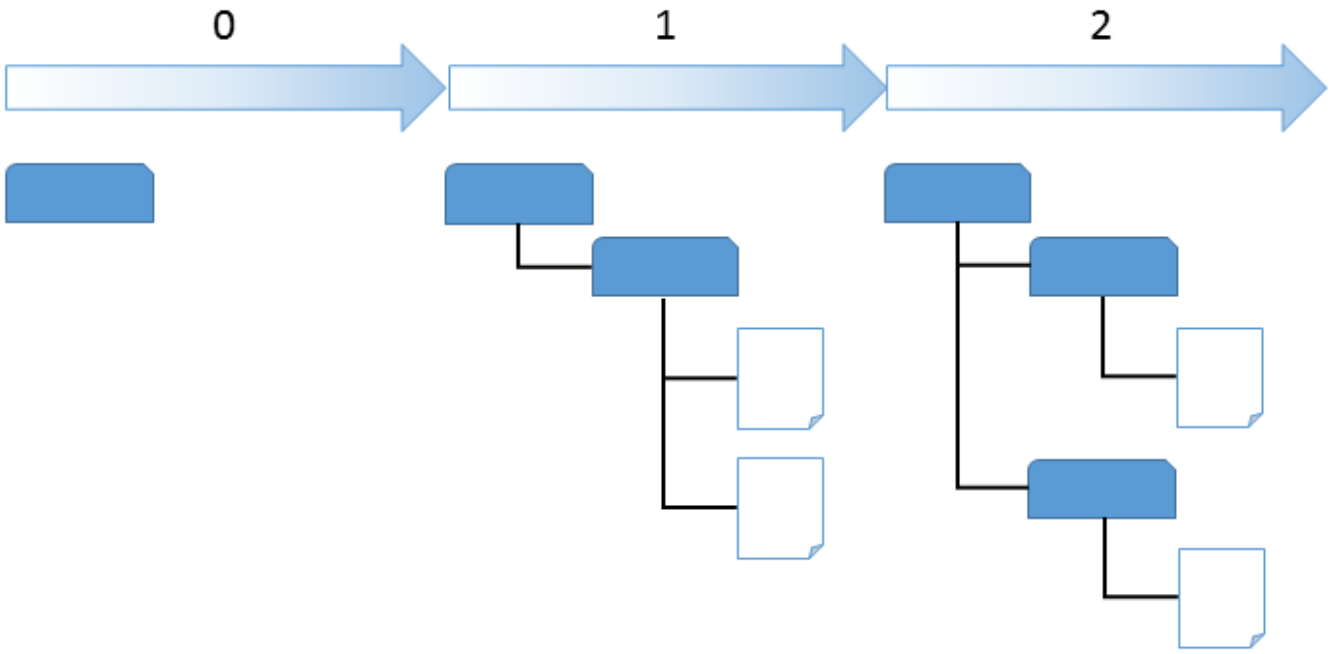
ATTENTION

Detailed changes are only documented for text files. SVN documents only the general information about changes made to binary files.

Revision – a documented state of the file hierarchy. In the repository, each commit is treated as an atomic transaction. Developers can modify several files, create, delete, rename and copy files and folders, and commit the entire set of changes as a single “revision”.

In SVN, all revisions are stored in form of file system “trees” - an array of revision numbers starting with 0 and “growing” from left to right (Fig. 1). Each number corresponds to a file system tree. And each tree is a "snapshot" of the storage state after each commit.

Fig. 1. Revisions in the repository



NOTE

Unlike other version control systems, revision numbers in Subversion refer to whole trees instead of individual files.

Versioning models

File-sharing problems

While working in SVN, there may be a problem where two (or more) developers are using the same file to implement different functionality. In this case, if one of the developers commits their changes a few seconds before the other, their changes may be overwritten. And while the system documents all changes, their work may still be lost in the latest version of the file, if the other developer accidentally overwrites the same file.

The following versioning models are used to avoid such problems:

- The “Lock-Modify-Unlock” solution
- The “Copy-Modify-Merge” solution

The “Lock-Modify-Unlock” solution

This versioning model only enables a single user to modify a specified file. A user may “block” the file for all other users, and they will not be able to commit their changes until the lock is released.

Disadvantages:

- Locking may cause administrative problems, e.g. when the first developer locks a file and forgets to release the lock.
- Locking may cause unnecessary serialization. If the developers are working on two separate parts of the same document which do not overlap (e.g., the beginning and the end), the changes can be properly merged together if the file is not locked.
- Locking may create a false sense of security. Two separate files that depend on each other may be locked by two different developers, which means the changes made to each locked file are semantically incompatible, but the two developers may think they are beginning a safe, insulated task. Thus, this model inhibits the two developers from discussing their incompatible changes early on.

This model is more appropriate when the two separate files can not be merged. For example, if two users are editing an image at the same time, they will not be able to merge their changes.

The “Copy-Modify-Merge” solution

In this model, each user's client reads the repository and creates a personal working copy of the file or project. Users then work in parallel, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but the user is responsible for making it happen correctly.

A *conflict* will happen if the changes overlap when two users work on the same file simultaneously. In this case, the user who commits the changes should select the necessary revision from the list of files in a conflict state. Upon resolving the conflict, the merged file can be committed to the repository.

The chance of semantic and syntactic conflicts increases if there is no communication between users.

Determining the state of the working copy file

Subversion records information about the following properties in the .svn service folder of a working copy for each file:

- the revision, which the file in the working copy is based on (working revision of the file)
- date and time of the latest file update from the repository

Based on this information, Subversion can determine the state of the working copy file:

1. Not modified and not outdated. Not modified in the working copy. The repository did not document any changes to this file since its working revision. The update or the commit procedures will not be executed.
2. Modified locally and not outdated. Modified in the working copy. The repository did not document any changes to this file since its base revision. The update will not be executed. The system will commit the changes successfully.
3. Not modified, outdated. Not modified in the working folder, modified in the repository. The file must be updated to match the current public revision. The update will not be executed. The system will commit the changes successfully.
4. Modified locally and outdated. Modified in the working folder and in the repository. The commit procedure will fail. The file must first be updated by attempting to merge the changes published by another developer with the local changes. The user must resolve the conflict if Subversion can not merge the files.

Working copy used in bpm’online

If the **file system development mode** is enabled, bpm’online will use a custom working copy of each user package with connected versioning. These working copies are located in the folder specified in the *defPackagesWorkingCopyPath* element of the *ConnectionStrings.config* configuration file (see “**How to deploy bpm'online on-site**”).

If the file system development mode is enabled, the working copy can be created manually in the *[Installed application path]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\[Package name]* folder (see: “**Creating a package in the file system development mode**”).

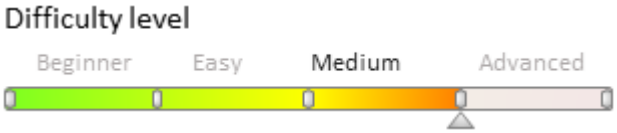
Using an application to work in SVN

We recommend using TortoiseSVN (version 1.9 and higher) to work with Subversion (SVN) in the file system. Once installed, the application will be built in in the Windows UI. Learn more about TortoiseSVN in this article.

See also

- **Creating a package in the file system development mode**
- **How to install an SVN package in the file system development mode**
- **How to bind existing package to SVN**
- **Updating and committing changes to the SVN from the file system**
- **Creation of the package and switching to the file system development mode**

Creating a package in the file system development mode



Introduction

If you do not intend to use SVN in the development process, then the process of creating a package is the file system development mode is the same as that in the normal mode. For more information on creating packages please refer to the “**Creating and installing a package for development**” article.

> Attention!
>
> The working with SVN mode is enabled in the bpm'online by default. If the [Version control system repository] field is empty when a package is created, then the package will not be bound to the repository. The versioned development of this package can be performed only after you manually bind it to the repository from the file system.

When the **file system development mode** is enabled, the SVN integration mechanism is turned off. The packages from the SVN repository can be only **installed** and **updated** with built-in tools. It is recommended to create a package with built-in tools and bind it to the repository with the external utilities like TortoiseSVN.

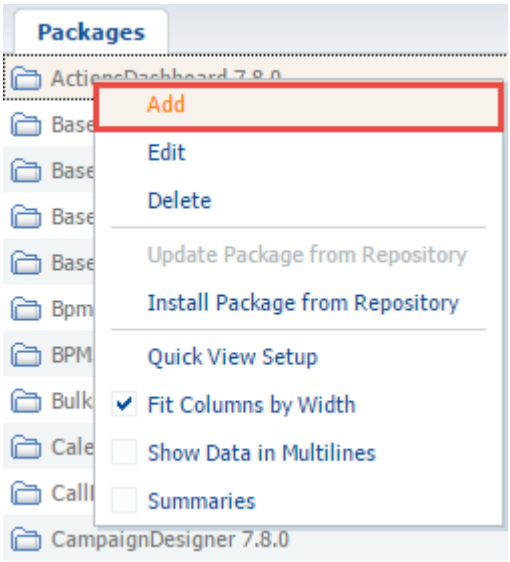
> Attention!
>
> Ensure that **application is configured to access the SVN repository** before binding the package to the SVN repository when development mode in the file system is enabled.

Package creation process

1. Create a package in the application

Select the [Add] action in the context menu of the [Packages] tab in the [Configuration] section (Fig. 1).

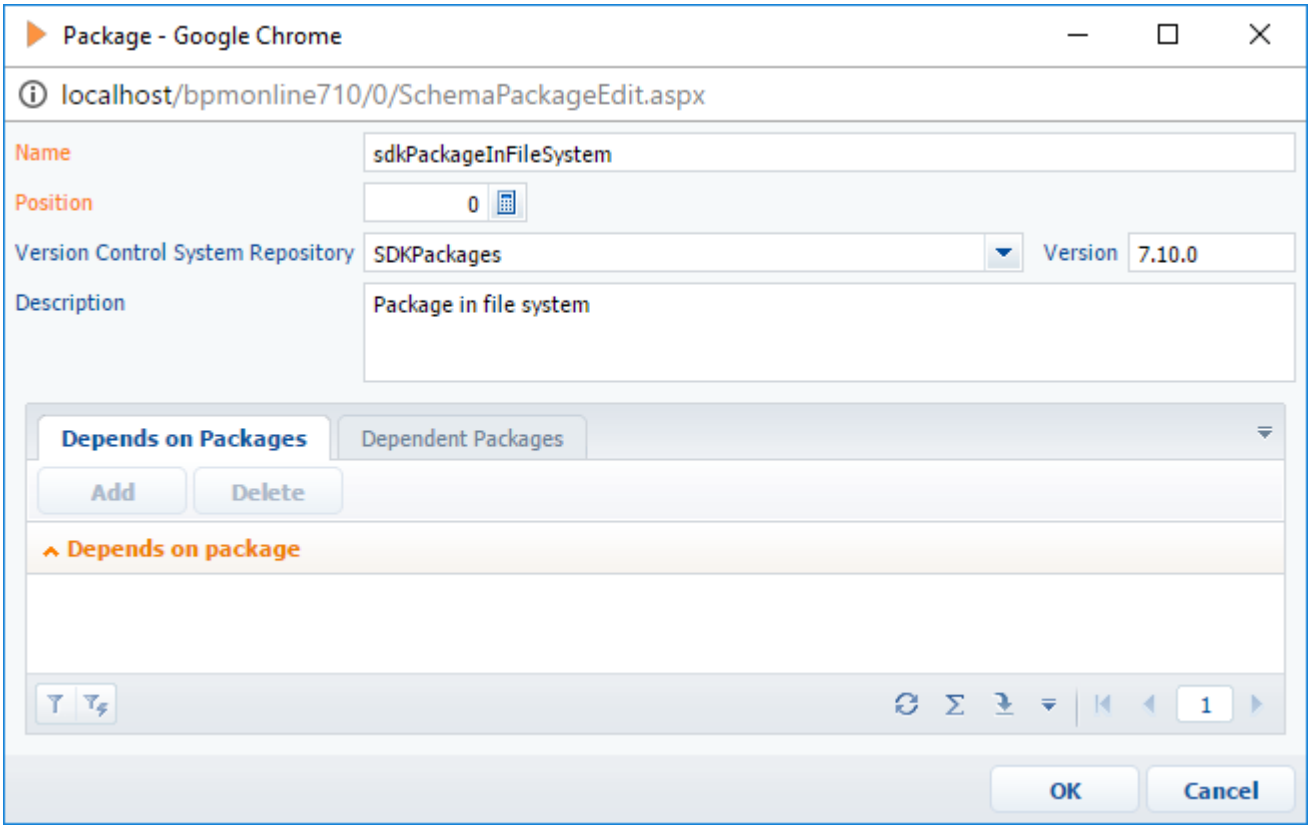
Fig. 1 Adding a package in the [Configuration] section



Fill out the main package properties fields in the package edit page (Fig. 2). Please see the "**Creating and installing a package for development**" article for details. Specify the repository name to which the package will be bound.

> Attention!
>
> The repository name in the package edit page indicates that the package will be created by third-party tools in this repository. This allows updating the package from the [Configuration] section in future.

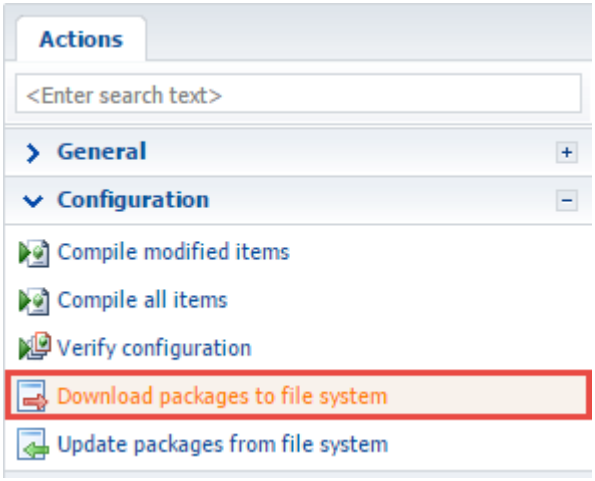
Fig. 2 Package summary



2. Download created package to file system

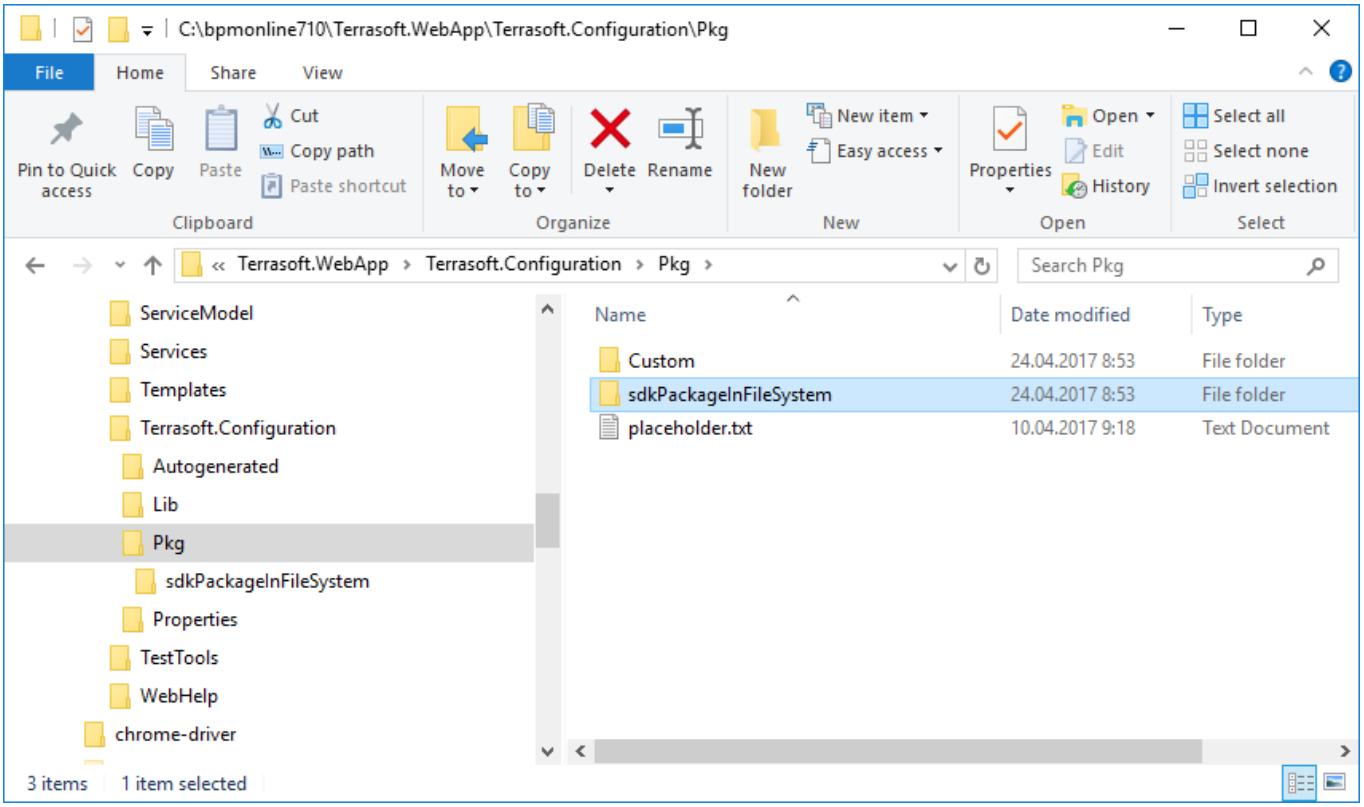
Click [Download packages to file system] (Fig. 3).

Fig. 3 [Download packages to file system] action



As a result, the empty package will be downloaded to the *[Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageInFileSystem* folder (Fig. 4).

Fig. 4 Package in the file system



NOTE

When the file system development mode is enabled, the package must be manually added to the repository.

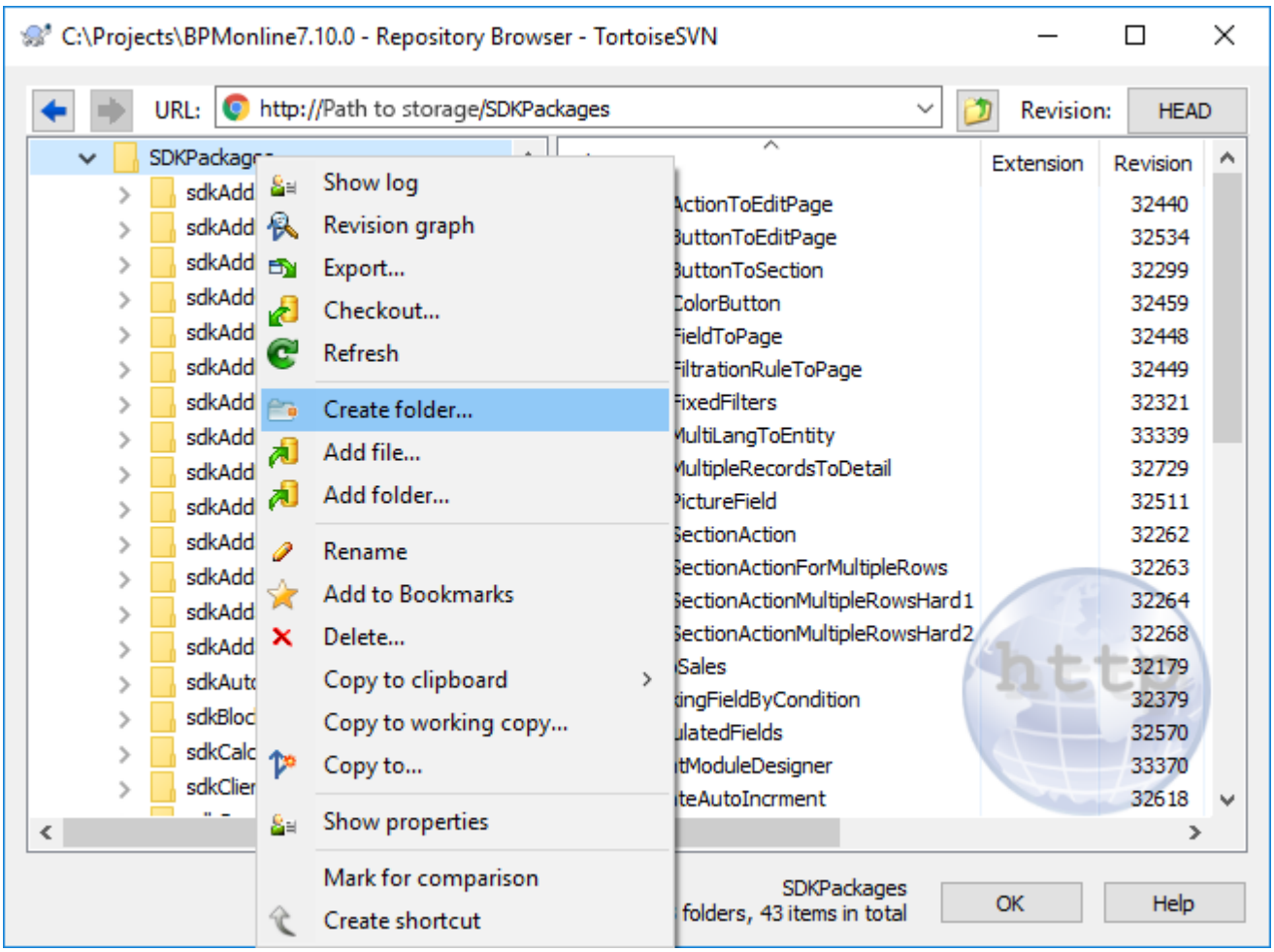
3. Create necessary folders for the packages in the SVN repository

Go to the repository specified in the package edit page to create folders for the package via SVN client (such as TortoiseSvn). Create a folder in the repository with the name that matches the name of the package created in the application.

Attention!

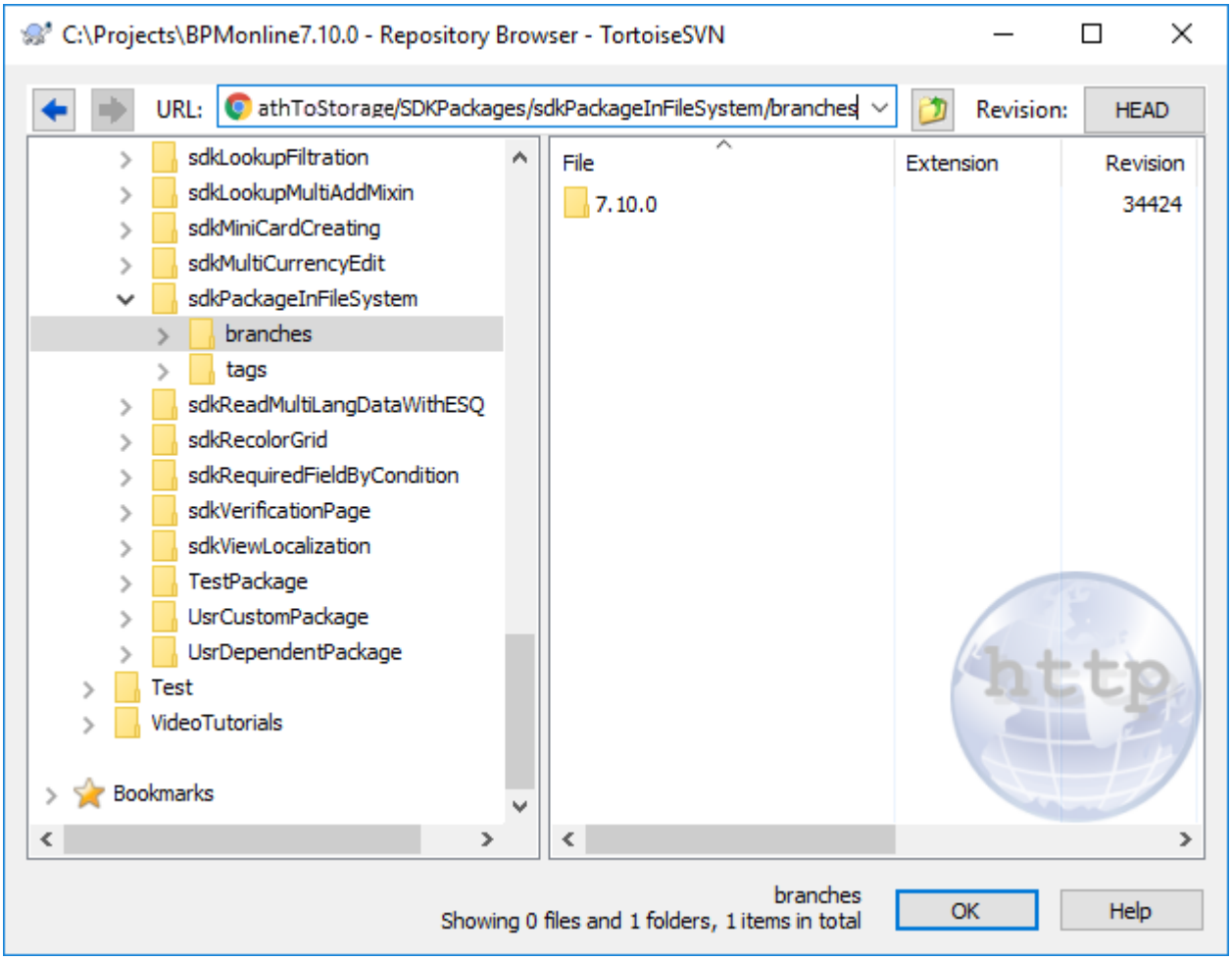
This is a brief example of working with SVN via TortoiseSvn. More information about working with SVN repository via TortoiseSvn can be found in the documentation.

Fig. 5 Creating a folder in the SVN repository



Create *branches* and *tags* sub-folders in the created folder to replicate the bpm’online **flat package structure**. Finally, create a folder with the name that matches the package version number (*7.10.0*) in the *branches* folder (Fig.6).

Fig. 6 Flat package structure in the repository



4. Create a working copy of the package version branch

To create a working copy of the package version branch, execute SVN checkout from the repository folder with the name that matches the package version number, to the package folder in the file system (Fig. 7) and confirm the download to the existing folder (Fig. 8).

Fig. 7 Obtaining the working copy of the package version branch from the repository

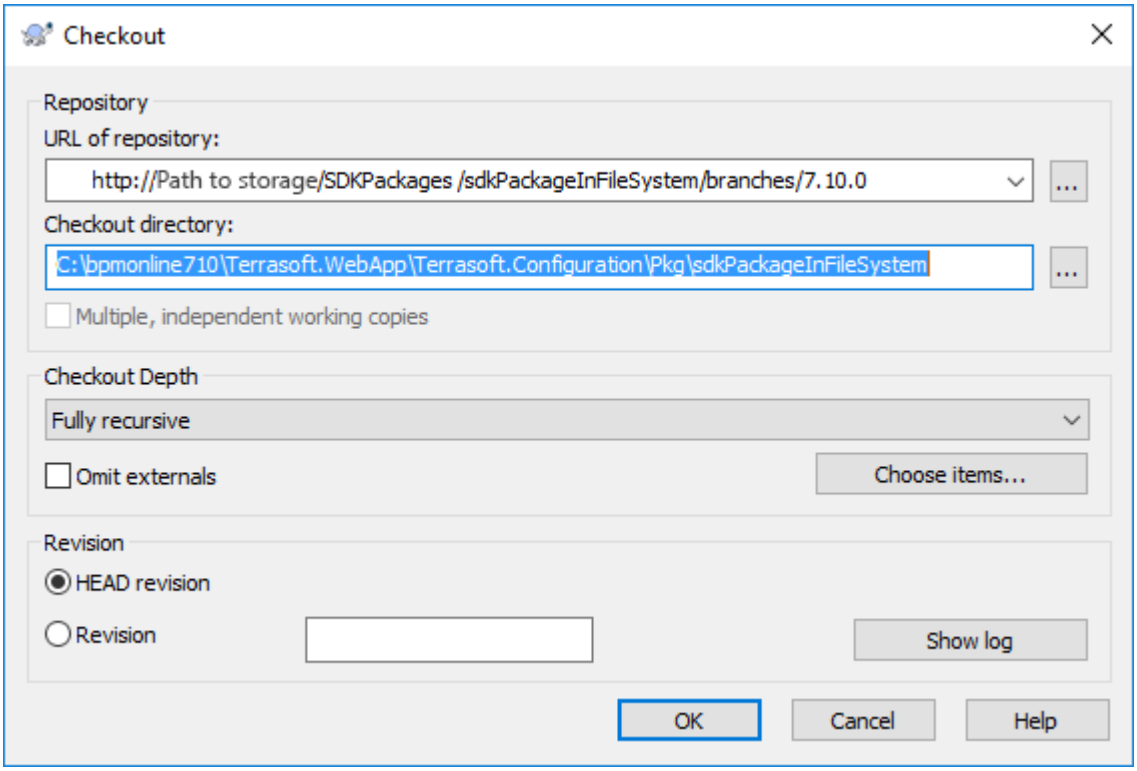
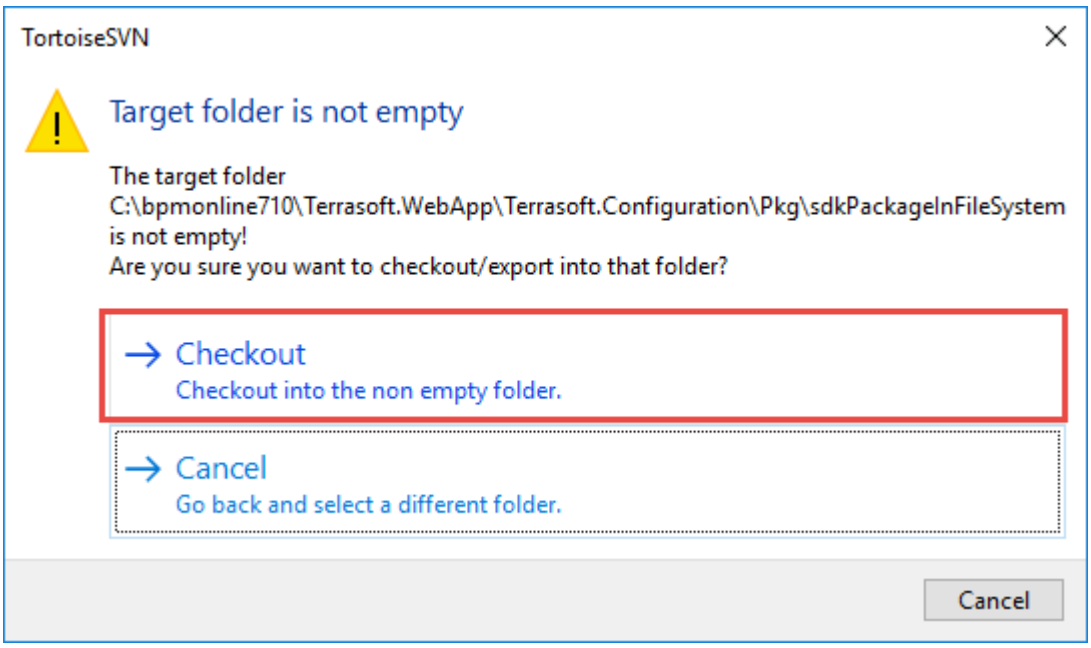
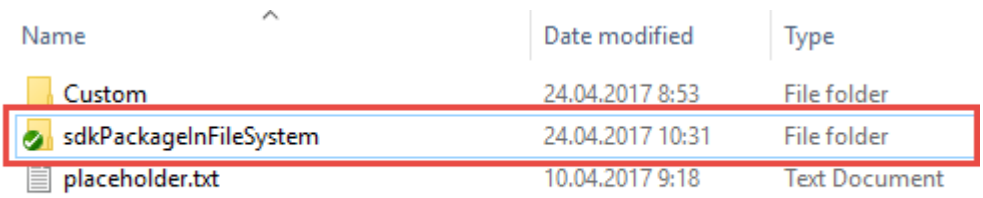


Fig. 8 Confirmation of the SVN checkout operation to the existing folder.



As a result the *[Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageInFileSystem* package folder will be bound to the branch of the 7.10.0 version of the package in the repository (Fig. 9).

Fig. 9 Visual mapping of the bound of the folder with the SVN repository



5. Commit the package folder in the repository

To commit the package folder, add all the contents of the following folder to the repository: *[Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageInFileSystem*. After adding the folder to the repository, execute the “Commit” command (Fig. 11).

Fig. 10 Adding a folder to the repository

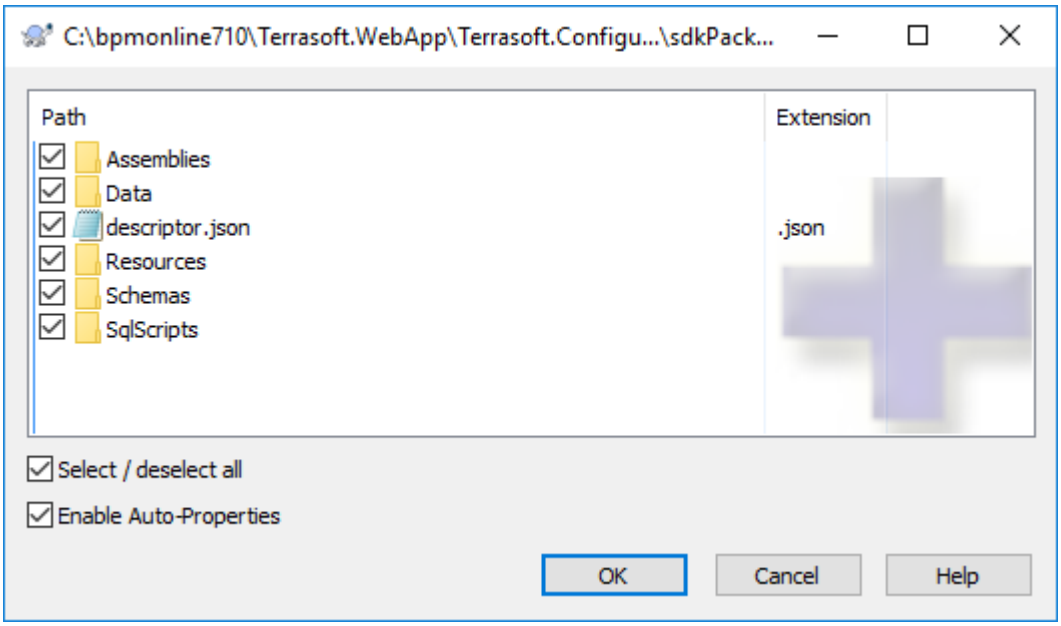
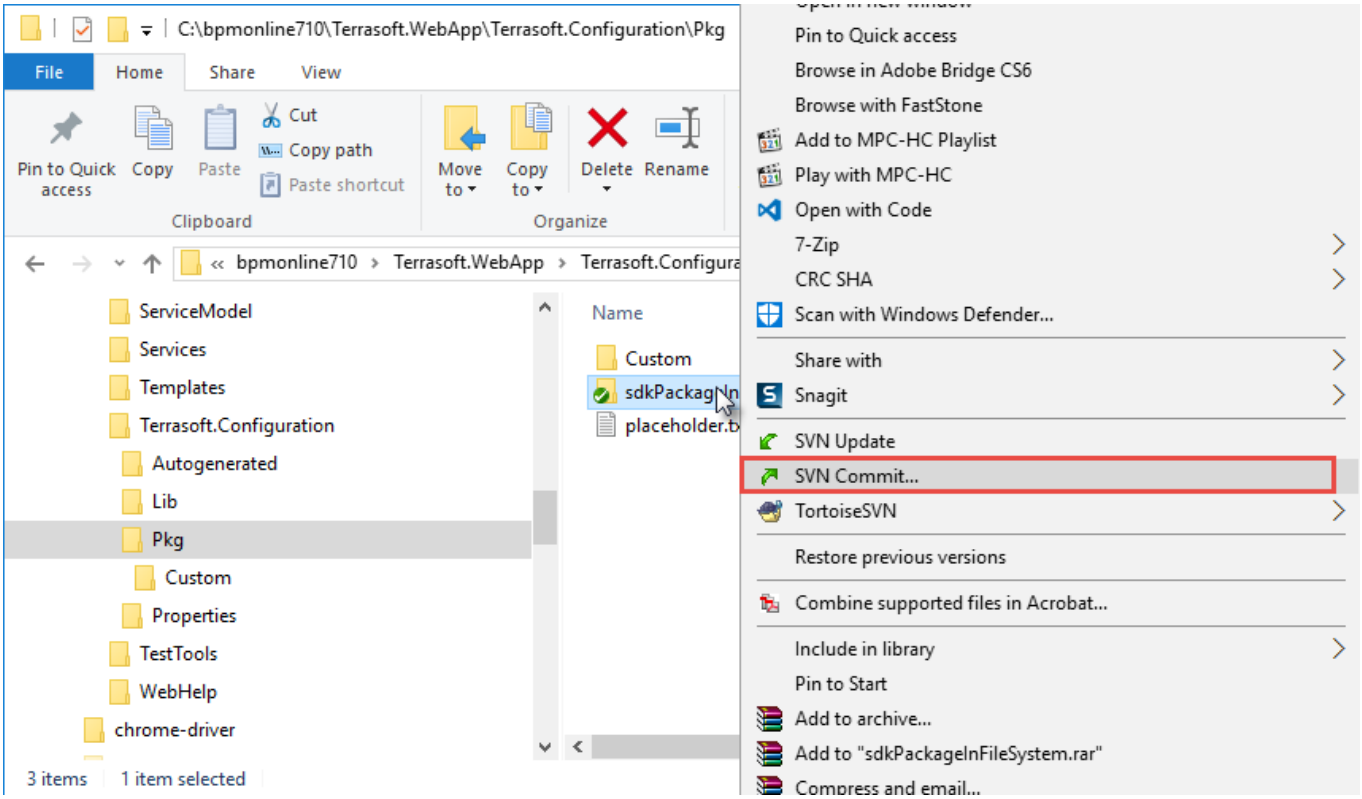


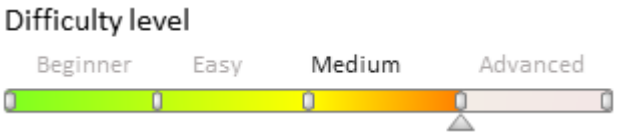
Fig. 11 Committing changes to the repository



See also

- **Working with SVN in the file system**
- **How to install an SVN package in the file system development mode**
- **How to bind existing package to SVN**
- **Updating and committing changes to the SVN from the file system**
- **Creation of the package and switching to the file system development mode**

How to install an SVN package in the file system development mode



Introduction

Package installation from the SVN repository in the **file system development mode** is considerably different from package setup in the [Configuration] section (see “ **Installing packages from repository**”). The main difference is that the interaction with the SVN repository (installing, committing, updating packages) is performed only from the file system.

The procedure for installing an SVN package in the file system development mode is as follows:

1. Install the package in the file system.

2. Install the package in the application.

3. Generate source codes.

4. Compile the changes.

5. Update the database structure.

6. Install SQL scripts and bound data, if necessary.

ATTENTION

To apply all necessary changes automatically, after the package is installed, enable the automatic mechanisms that apply changes. To do this, edit the ..\Terrasoft.WebApp\Web.config file and set the following *appSettings* element keys to *true*:

```
<add key="AutoUpdateOnCommit" value="true" />
<add key="AutoUpdateDBStructure" value="true" />
<add key="AutoInstallSqlScript" value="true" />
<add key="AutoInstallPackageData" value="true" />
```

The *AutoUpdateOnCommit* key is responsible for automatic updating of packages from the SVN before committing. If this key is set to *false*, then the application will notify the user that an update is required if the package schemas have been modified. The *AutoUpdateDBStructure*, *AutoInstallSqlScript*, *AutoInstallPackageData* keys are responsible for automatic database structure update, SQL script installation and installation of bound data respectively.

If the mechanism for auto-applying the changes is enabled, then steps 3—6 can be skipped. The mode must be enabled before the package is installed.

ATTENTION

If you need to interact with the SVN repository both from the [Configuration] section and the file system, then:

1. Install the package from the repository in the [Configuration] section (see " **Installing packages from repository**").

2. Export the package to the file system using the [Download packages to file system] action.

Repeat steps 3—6 from the package installation sequence.

Case description

In the file system development mode, install the package from the SVN repository under the following URL:

```
http://svn-server:8050/SDKPackages/sdkCreateDetailWithEditableGrid/branches/7.8.0
```

NOTE

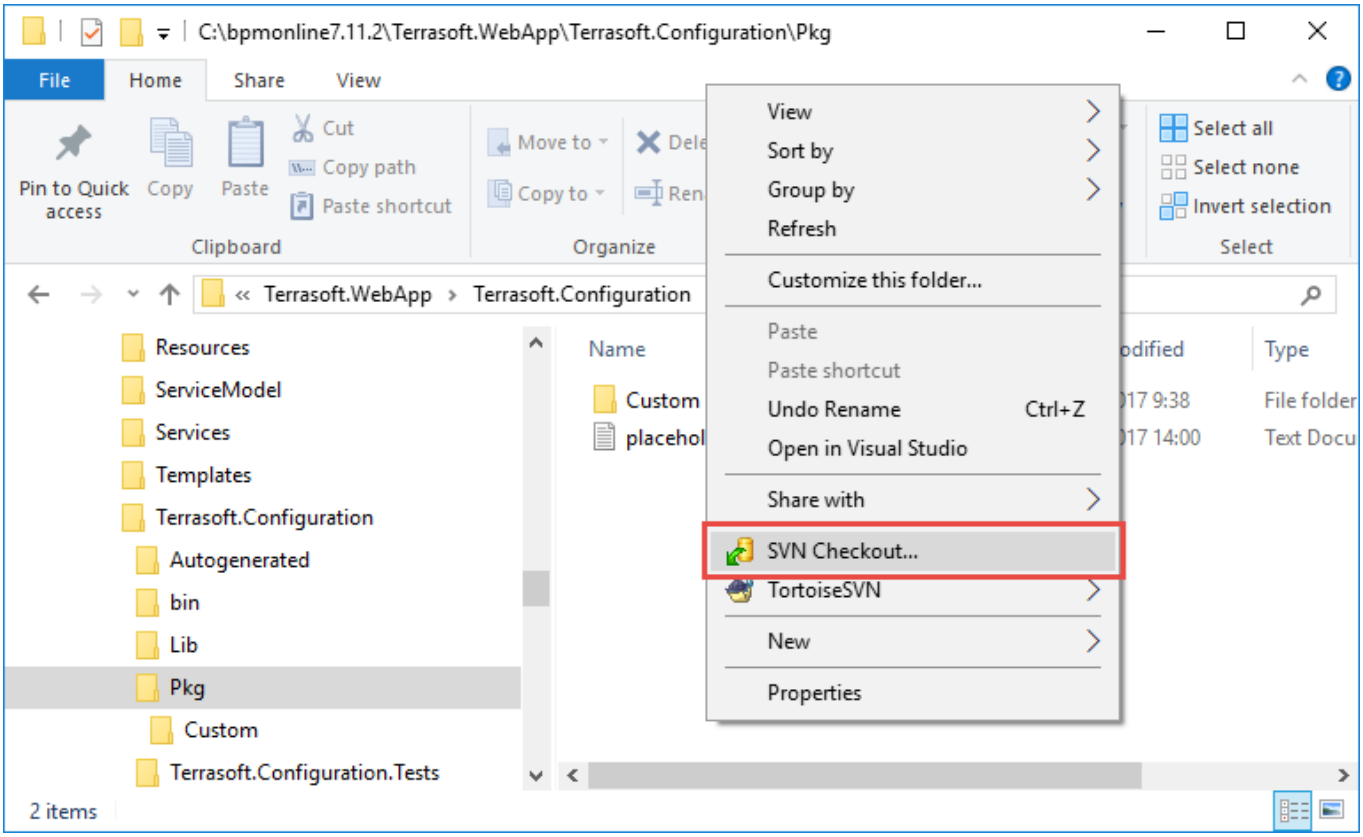
The package in this particular example contains a detail with editable list, which was created according to this article: "**Adding a detail with an editable list**".

Case implementation algorithm

1. Installing the package in the file system

Open the application catalog ...\Terrasoft.WebApp\Terrasoft.Configuration\Pkg in Windows Explorer and execute [SCN Checkout] (Fig. 1).

Fig. 1. Running the [SVN Checkout] action

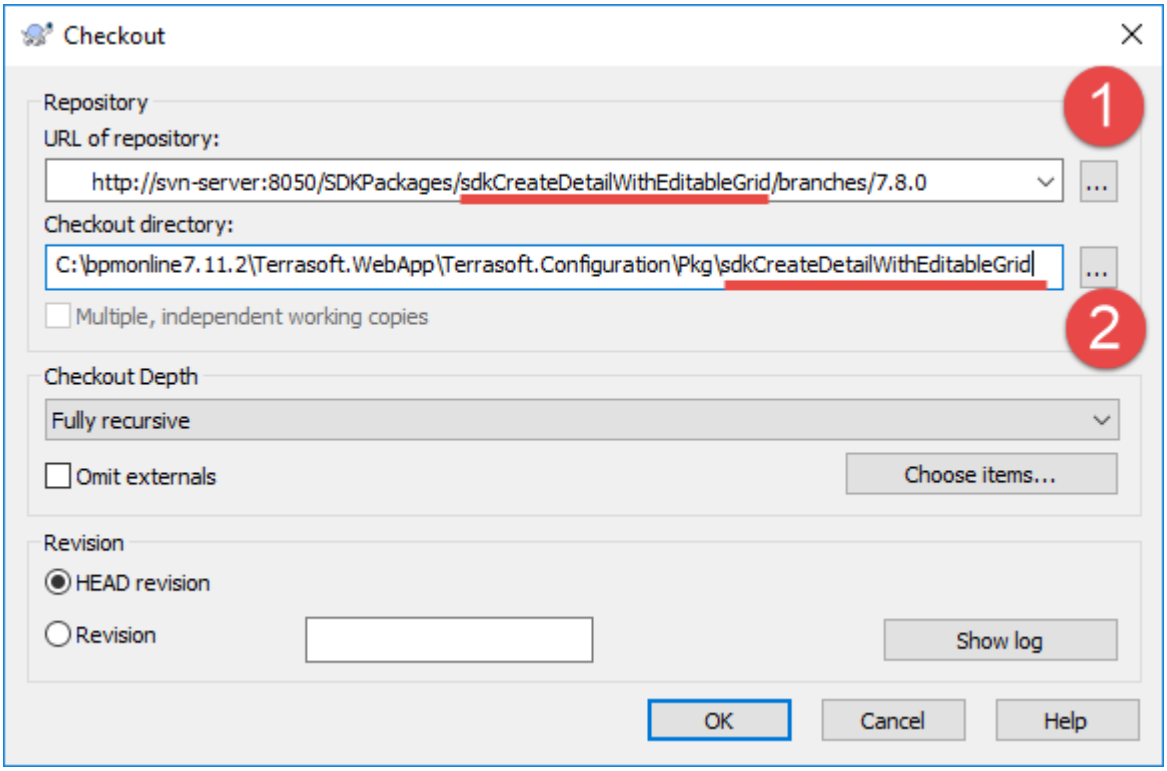


NOTE

We recommend using the TortoiseSVN client (v.1.8 and up) for working with Subversion. This client is a Windows shell extension and is built into the Explorer context menu. For more information see TortoiseSVN documentation.

In the checkout window (Fig. 2), specify the package repository address (1) and the export catalog for the package contents (2).

Fig. 2. TortoiseSVN Checkout window



ATTENTION

The checkout catalog name must be the same as the package name (Fig. 2).

After the checkout is complete (Fig. 3), the ..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg catalog will contain a local working copy of the package (Fig. 4).

Fig. 3. Checkout operation log

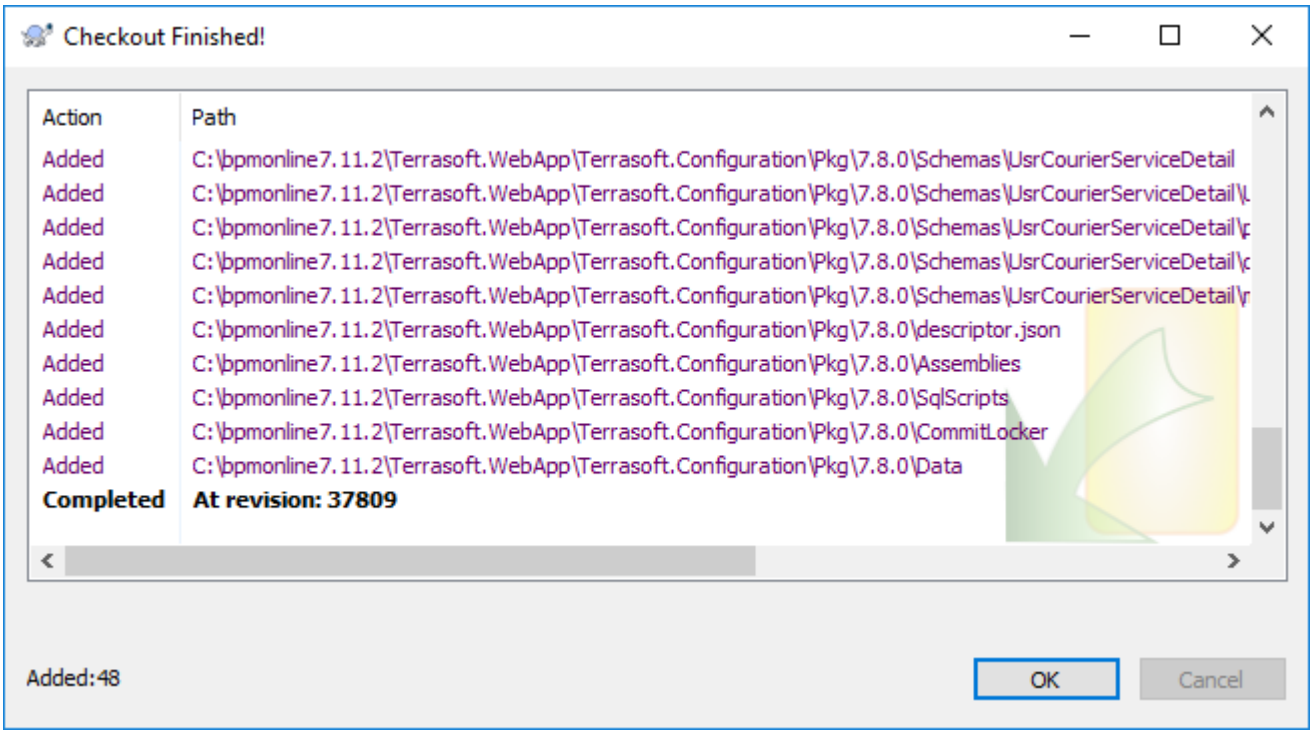
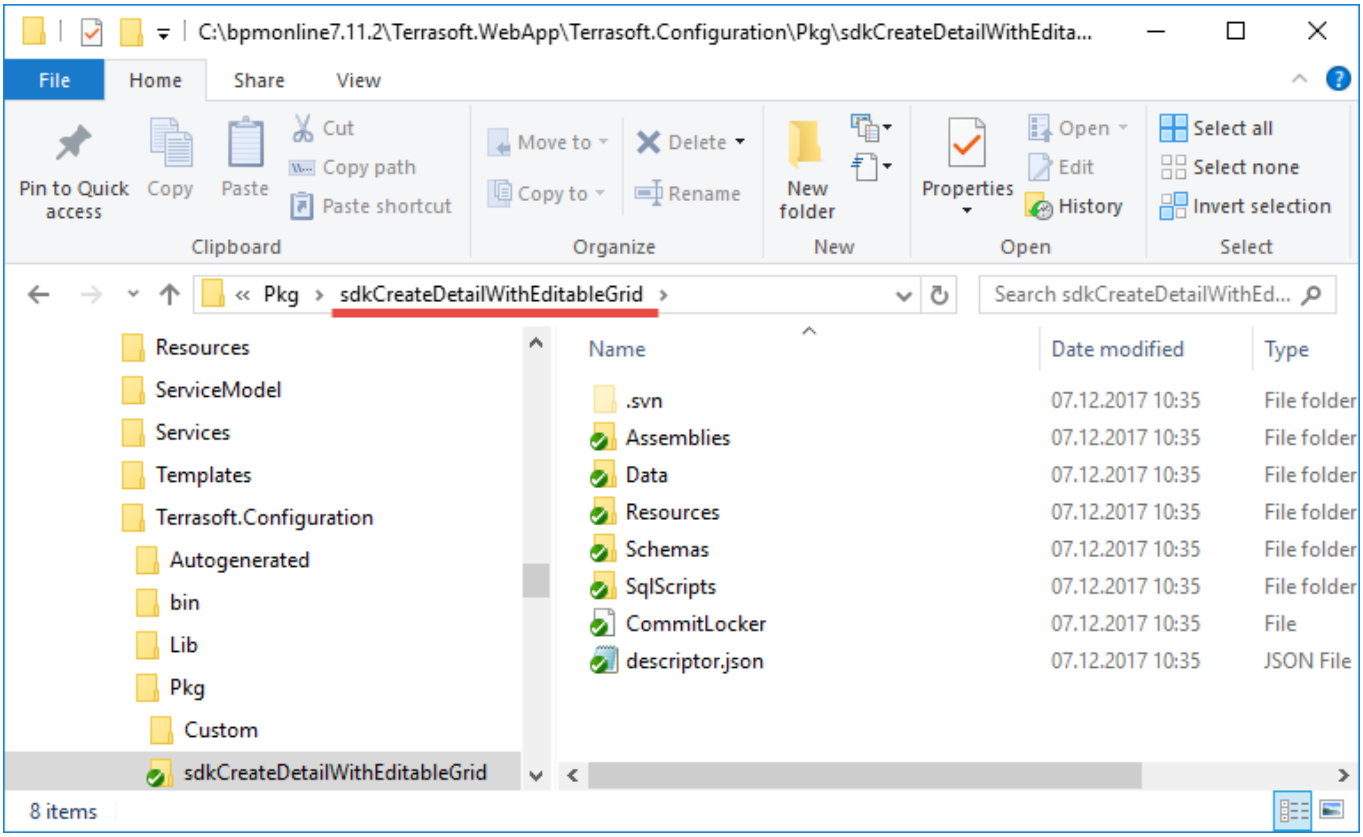


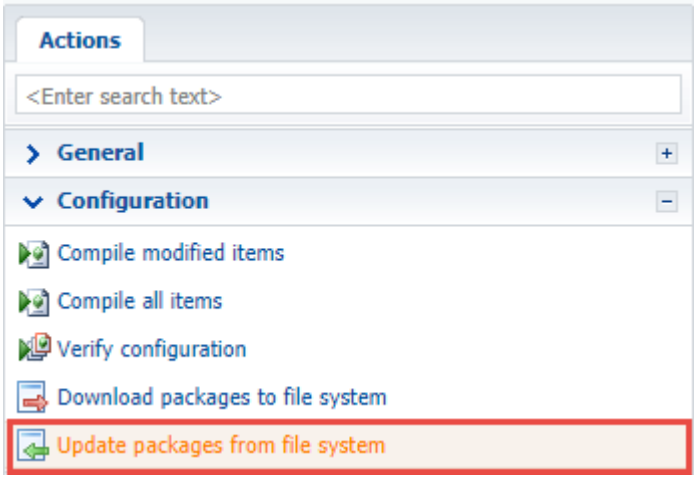
Fig. 4. Package working copy



2. Installing the package in the application

To install a package from the file system, go to the [Configuration] section and execute the [Update packages from file system] action (Fig. 5).

Fig. 5. The [Update packages from file system] action



As a result, the package will be added to the application (Fig. 6, Fig. 7).

Fig. 6. Message with the package importing status

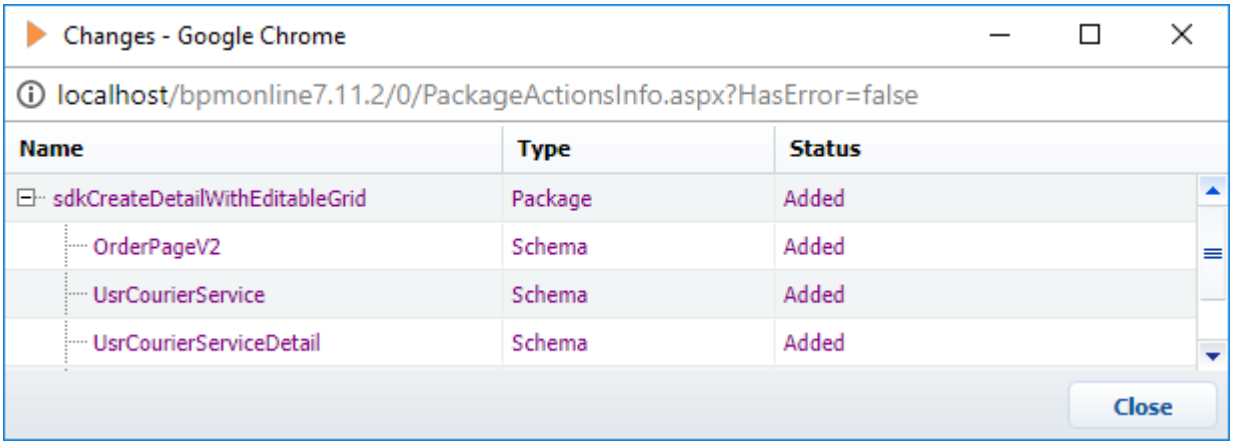
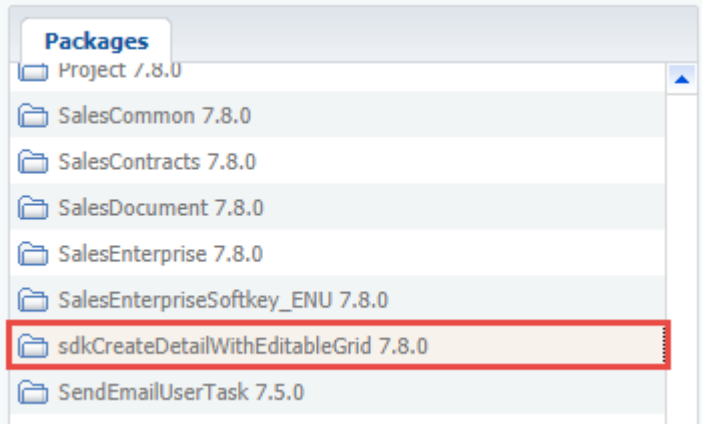


Fig. 7. The package on the [Packages] tab



ATTENTION

If the repository name is missing from the package name, then all changes can be committed to the repository from the file system only.

3. Generating source codes

Execute the [Generate where it is needed] action in the [Configuration] section to generate source core. For more on source code actions, see "**The [Configuration] section**".

4. Compiling the changes

To compile the changes, execute the [Compile modified items] action. For more on configuration actions, see "**The [Configuration] section**".

NOTE

The requirement for updating the database structure and installing SQL scripts is indicated by the [Database update required] and [Needs to be installed in the database] columns. Please refer to the [Last error message text] column in case of errors during the database structure update and SQL script installation.

Not all these columns display in the list of the [Schemas], [SQL scripts] and [Data] tabs of **the [Configuration] section** by default. Right-click the list and select the [Set up columns] command to add these columns to view.

5. Updating the database structure

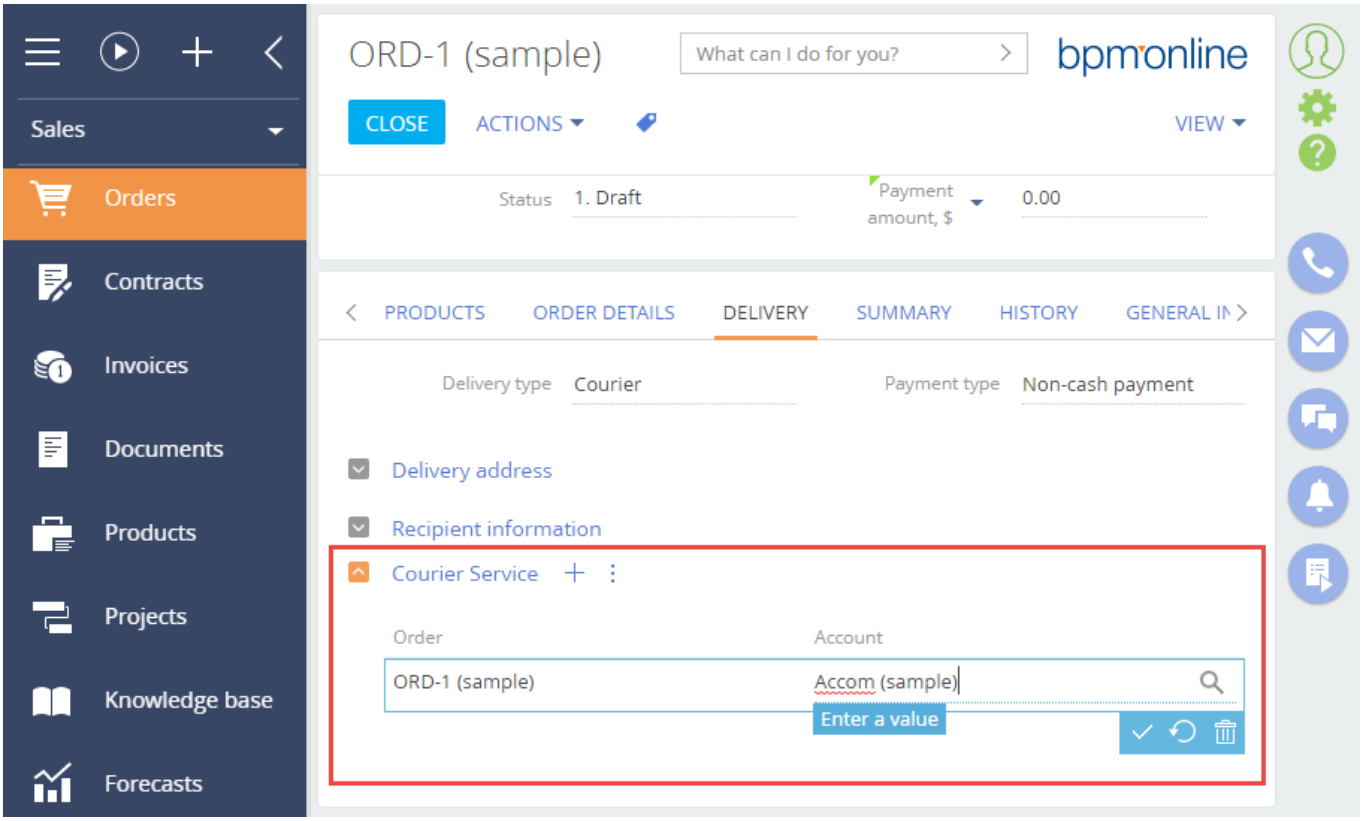
After the compilation is complete, run the [Update where it is needed] action. For more on database structure actions, see "**The [Configuration] section**".

6. Installing SQL scripts and bound data

If the package contains bound SQL scripts and data, you will need to run additional actions. For more on the SQL script and data actions, see “**The [Configuration] section**”.

After all the setup actions have been completed, the package functions will become available. In this case, it is a custom detail with editable list (Fig. 8).

Fig. 8. Custom detail with editable list

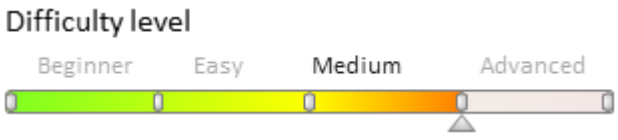


> **NOTE**
>
> Users may need to update the page and clear browser cache to access the new functions.

See also

- **Working with SVN in the file system**
- **Creating a package in the file system development mode**
- **How to bind existing package to SVN**
- **Updating and committing changes to the SVN from the file system**
- **Creation of the package and switching to the file system development mode**

How to bind existing package to SVN



> **ATTENTION**
>
> You can bind an existing package to SVN only in an on-site application. Working with such a package is

possible only in the file system development mode.

NOTE

After binding a package to SVN in the file system, it can be installed in a different application using bpm’online built-in tools (“**Installing packages from repository**”).

Introduction

Simple custom functions can be developed by a single developer. The package in which the development is performed often is not connected to the version control repository (SVN).

As the complexity grows, more developers are required to work with the package, which makes it necessary to bind the package to SVN.

The general sequence for binding a package to the repository is as follows:

1. Switch to the file system development mode

2. Export the package to the file system.

3. Create the catalogs for the package in the SVN repository.

4. Create a working copy of the package branch.

5. Commit the package catalog in the repository.

As an alternative, you can use direct SQL queries to the database for binding packages to the repository. To do this:

1. Add information about the SVN repository in the **[Configuration] section**.

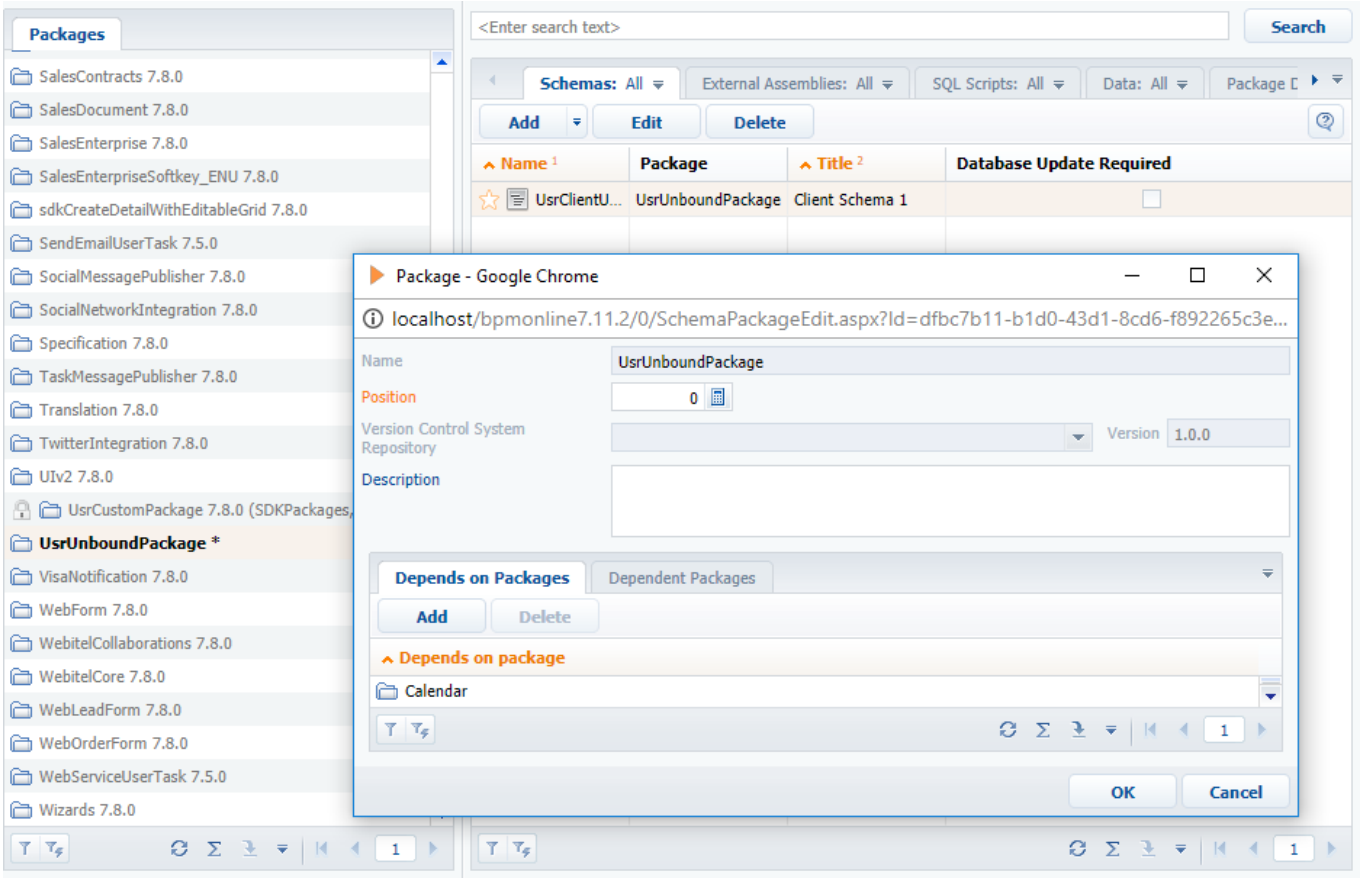
2. Bind the repository to the package. To do this:

- In the SysRepository table, read the record ID, which contains the SVN repository address.
- In the SysPackage table, add the obtained Id to the SysRepositoryId for the unbound package.

Case description

Bind the custom *UsrUnboundPackage* package (Fig. 1) to the repository.

Fig. 1. Custom package properties



You can obtain SVN access via the following URL:

```
http://svn-server:8050/SDKPackages
```

Case implementation algorithm

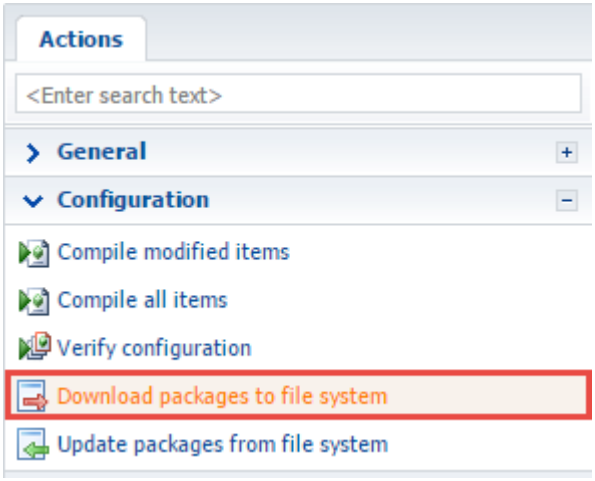
1. Switch to the file system development mode.

For more information about the file system development mode, see the “**Development in the file system**” article.

2. Export the package to the file system.

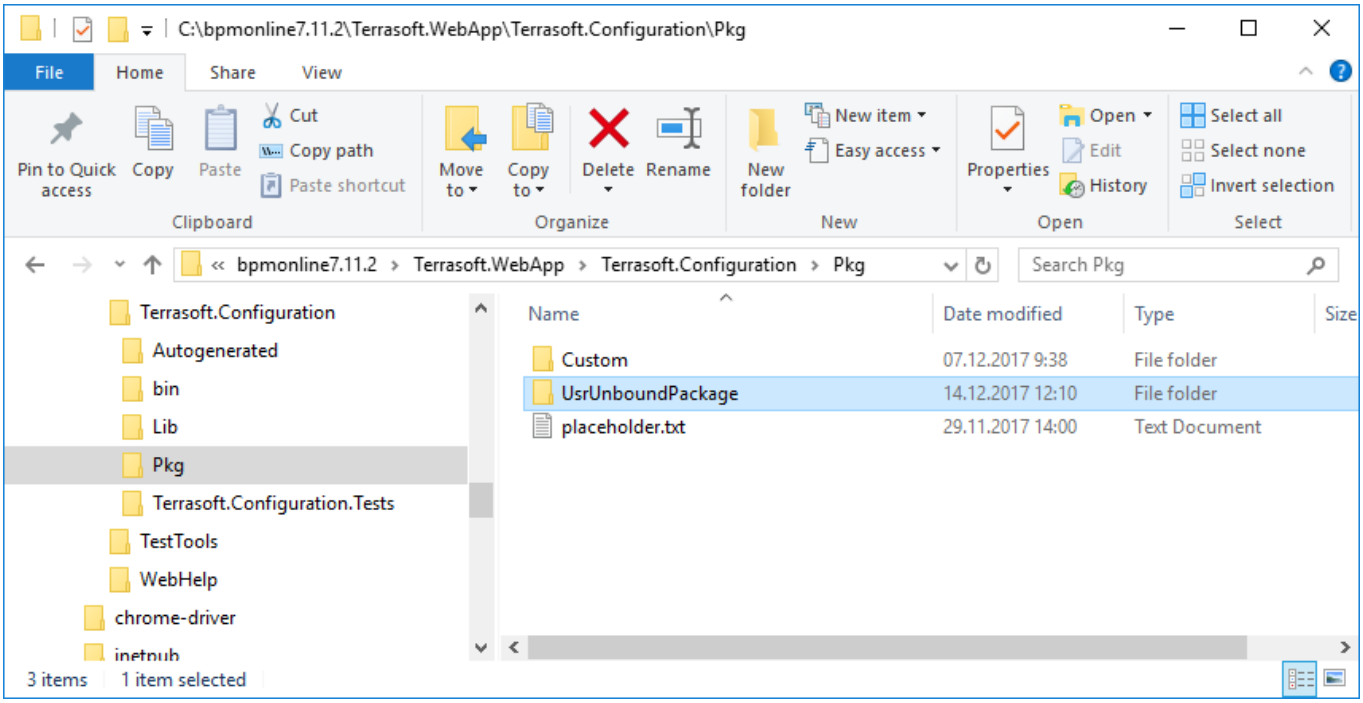
In the [Configuration] section, run the [Download packages to file system] command (Fig. 2).

Fig. 2. The [Download packages to file system] command



As a result, the package will be exported to the following catalog:
...\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\UsrUnboundPackage (Fig. 3).

Fig. 3. The package in the file system

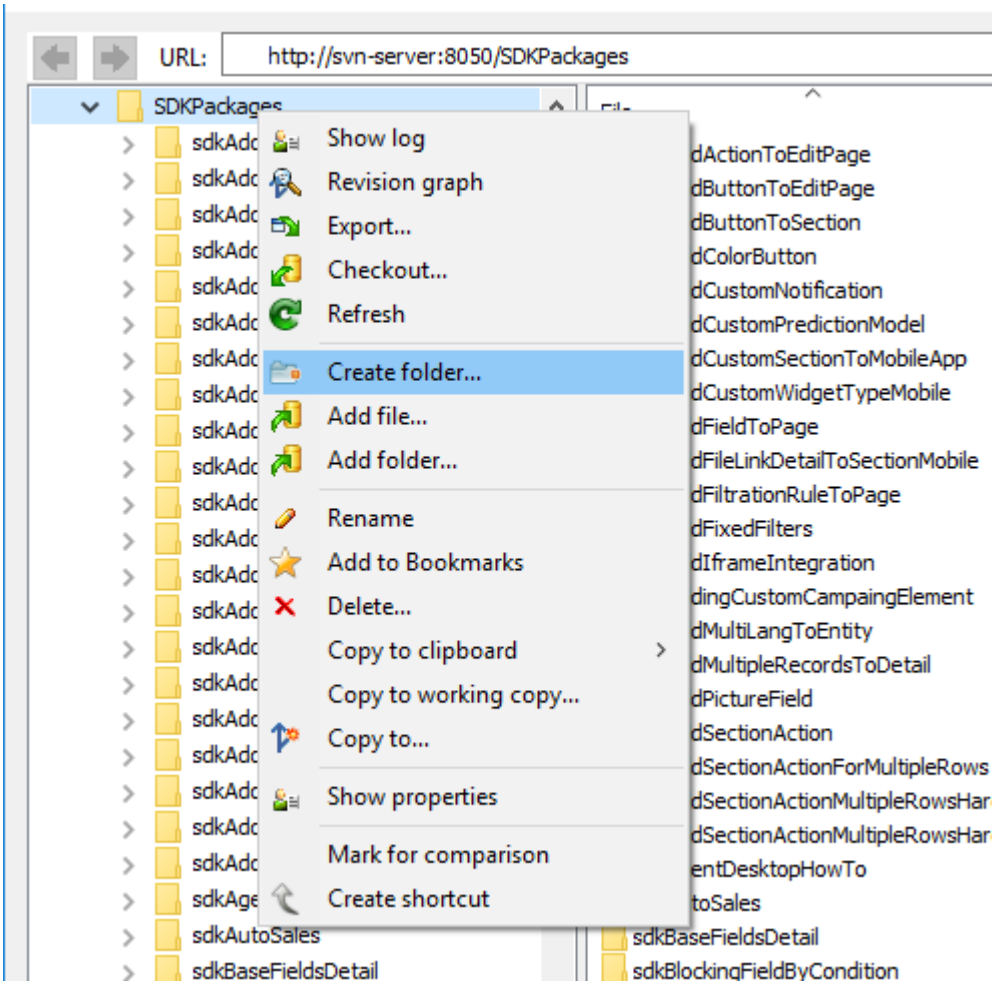


3. Create the catalogs for the package in the SVN repository.

To create the package catalogs using an SVN client (such as TortoiseSvn), go to the repository and add a catalog (Fig. 4) with the same name as the package.

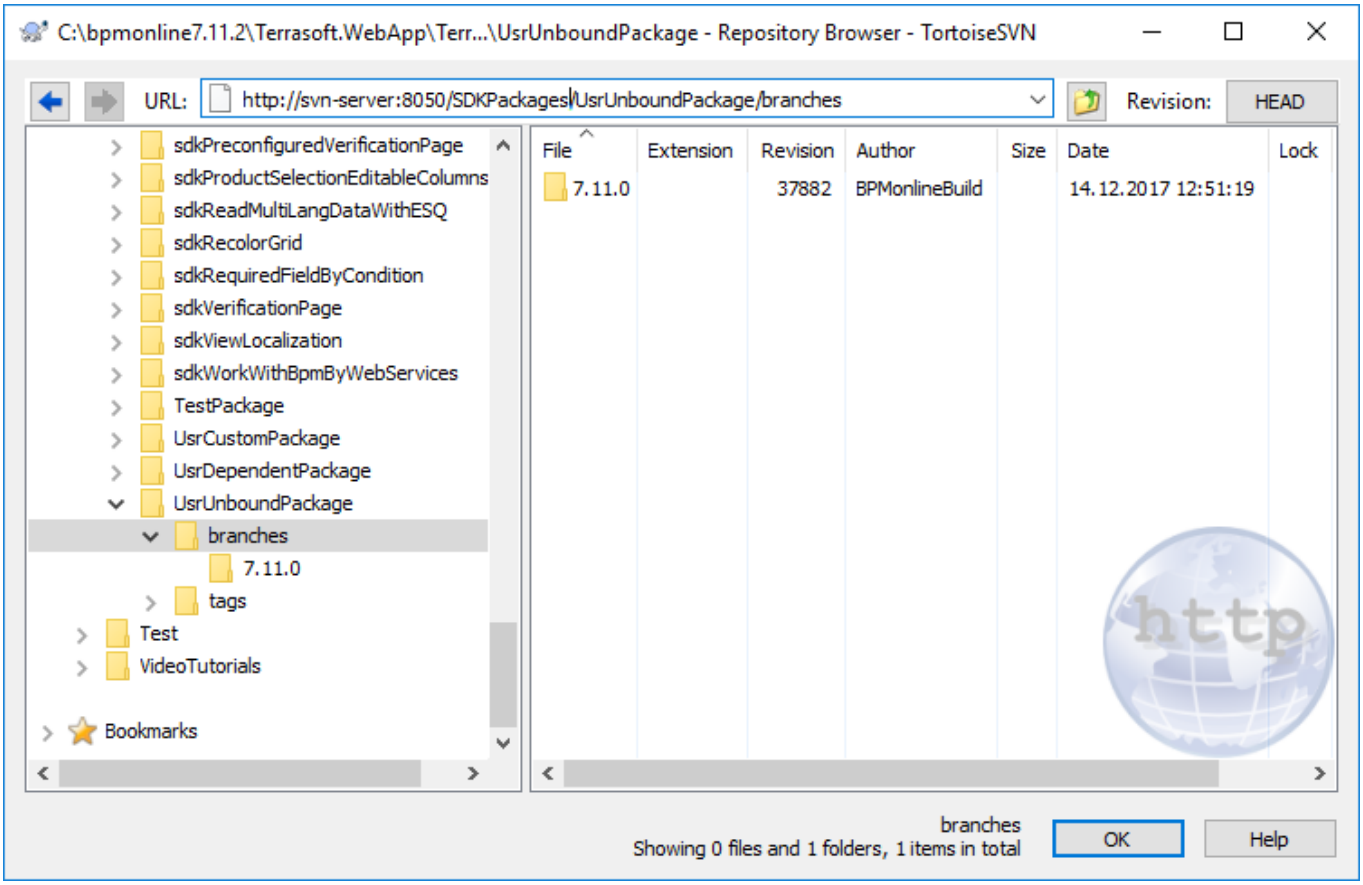
> ATTENTION
>
> This article contains only general instructions for working with the SVN client. For in-depth instructions on working with the repository via TortoiseSvn are available in the official TortoiseSvn documentation.

Fig. 4. Creating a catalog in the SVN repository



Create the *branches* and *tags* sub-folders in the package catalog, i.e., reproduce bpm’online **flat package structure**. In the *branches* catalog, create a package version catalog, i.e., *7.11.0* (Fig. 5).

Fig. 5. Flat package structure in the repository



> **NOTE**
>
> Reproducing the flat package structure is required only if you plan on using bpm’online built-in tools for working with SVN.

4. Create a working copy of the package branch.

To create a working copy if the version-controlled package branch, export the catalog whose name matches the package version number (the [SVN Checkout...] command, Fig. 6) to the package's catalog in the file system.

Fig. 6. Running the [SVN Checkout] action

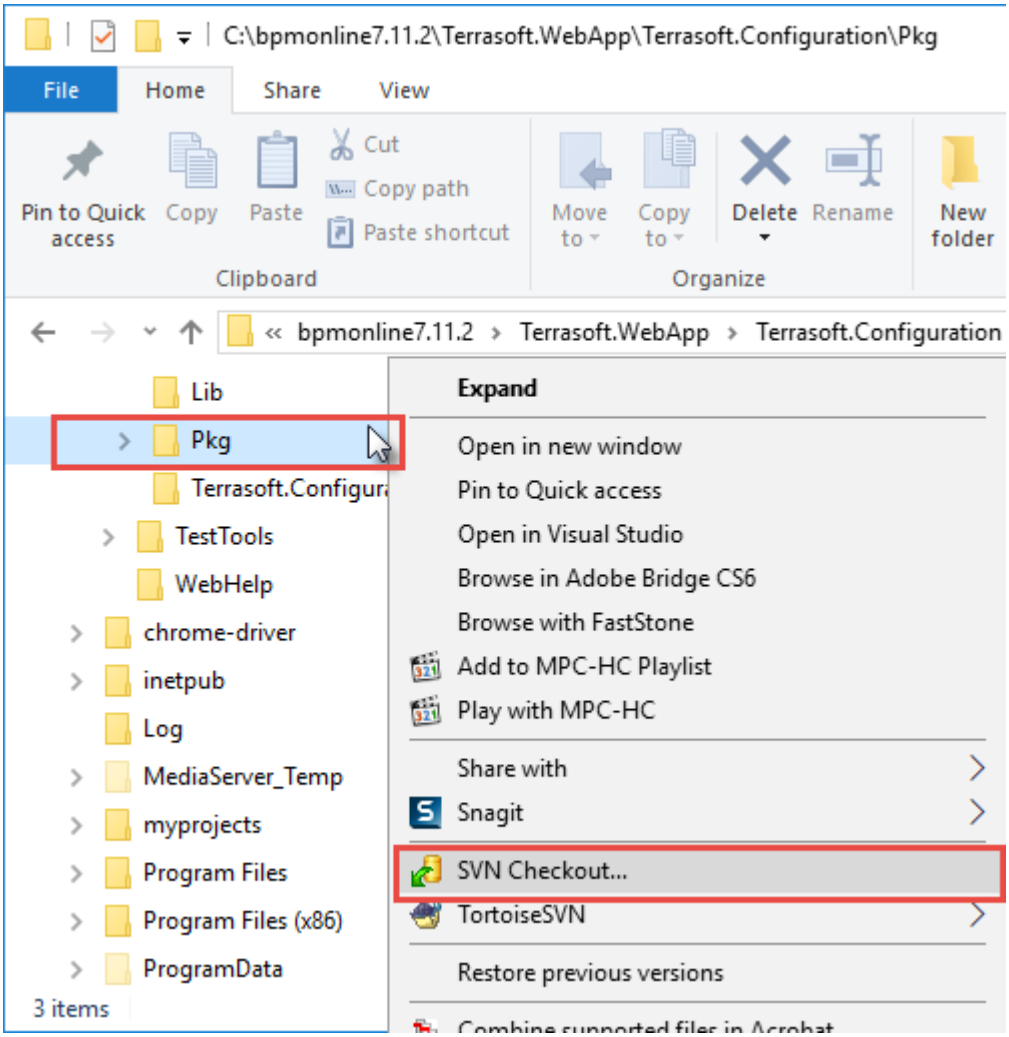


Fig. 7. Exporting working copy of a version-controlled package branch

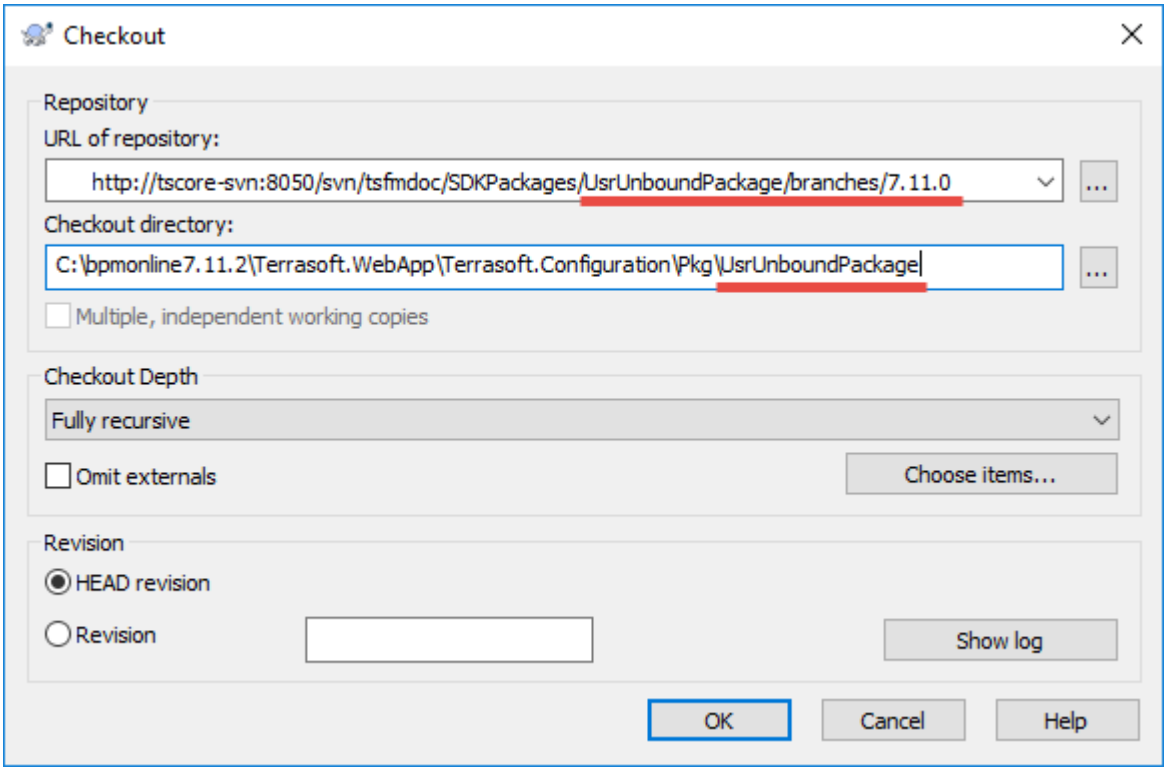
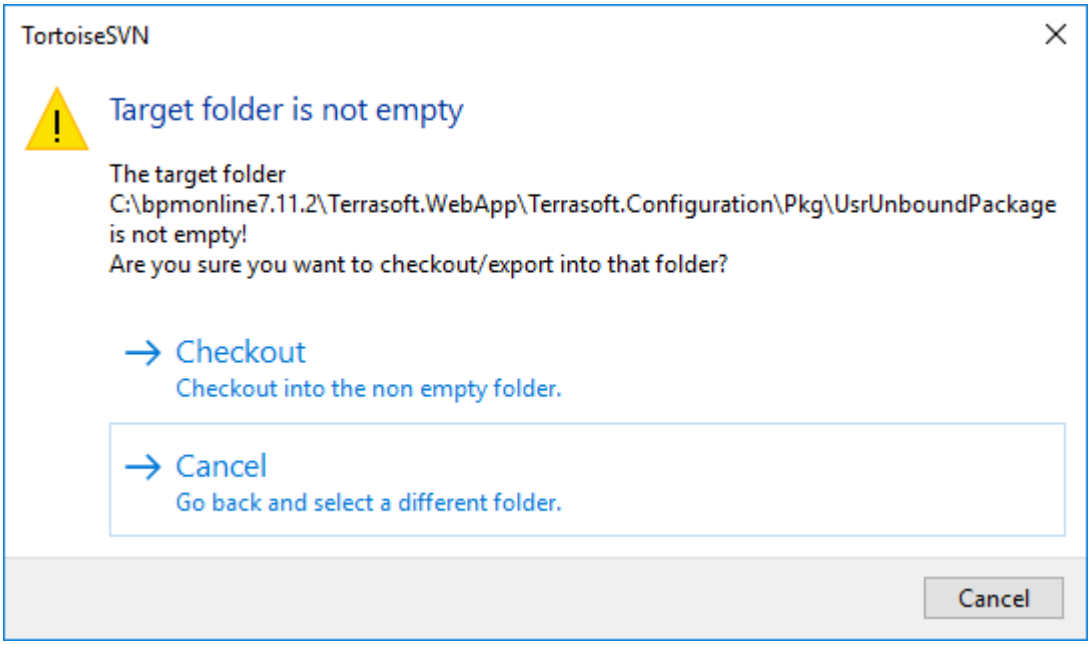
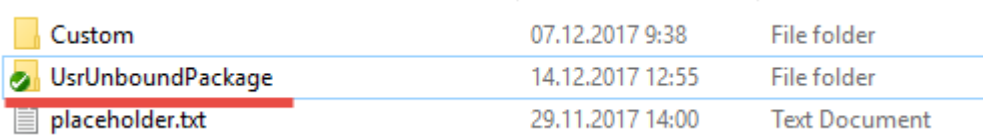


Fig. 8. Confirming the export to an existing catalog



As a result, the package catalog in the file system (*...\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\UsrUnboundPackage*) will become a working copy of a 7.11.0 package in the repository (Fig. 9).

Fig. 9. A catalog, connected to the SVN repository



5. Commit the package catalog in the repository.

To commit the contents of a package catalog to the repository, run the TortoiseSVN [Add...] command (Fig. 10 and 11), then run the [SVN Commit...] command (Fig. 12).

Fig. 10. The [Add] command

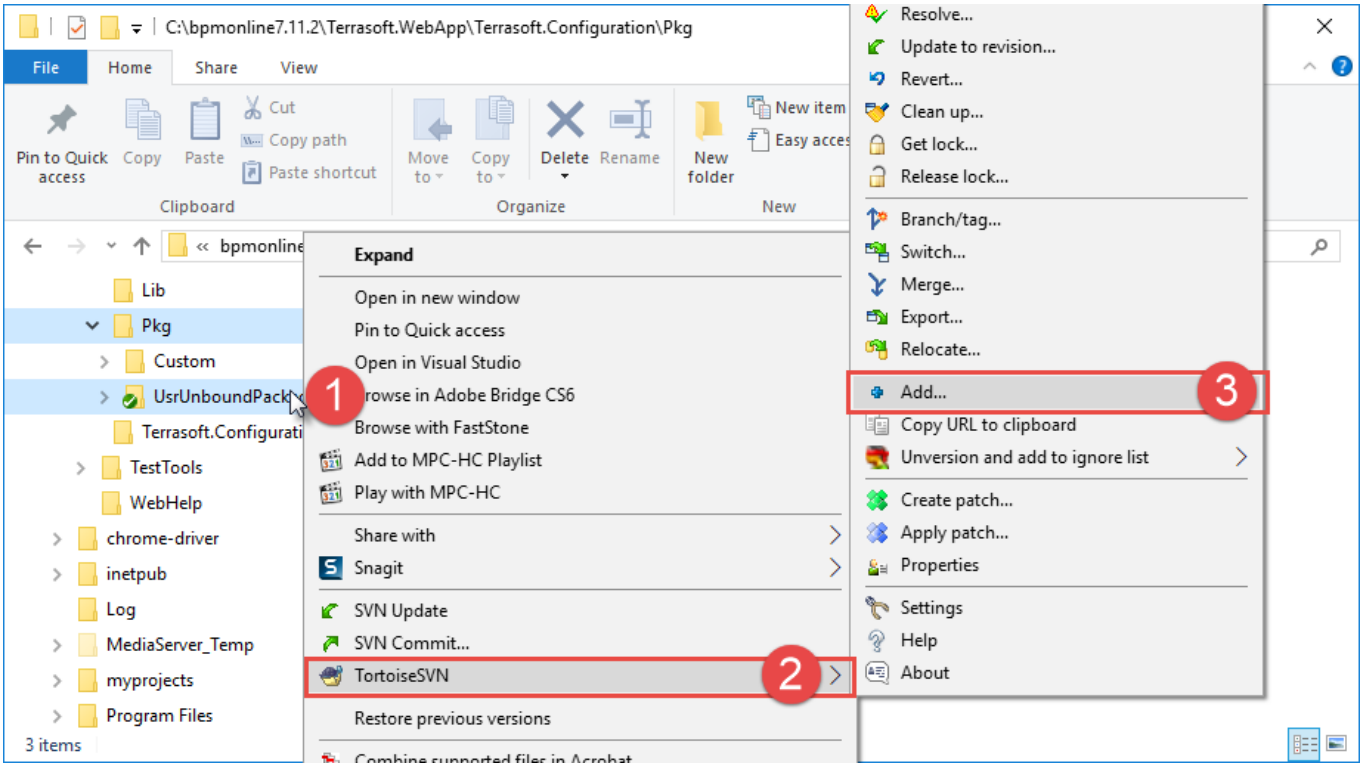


Fig. 11. Dialog for selecting items to add to the repository

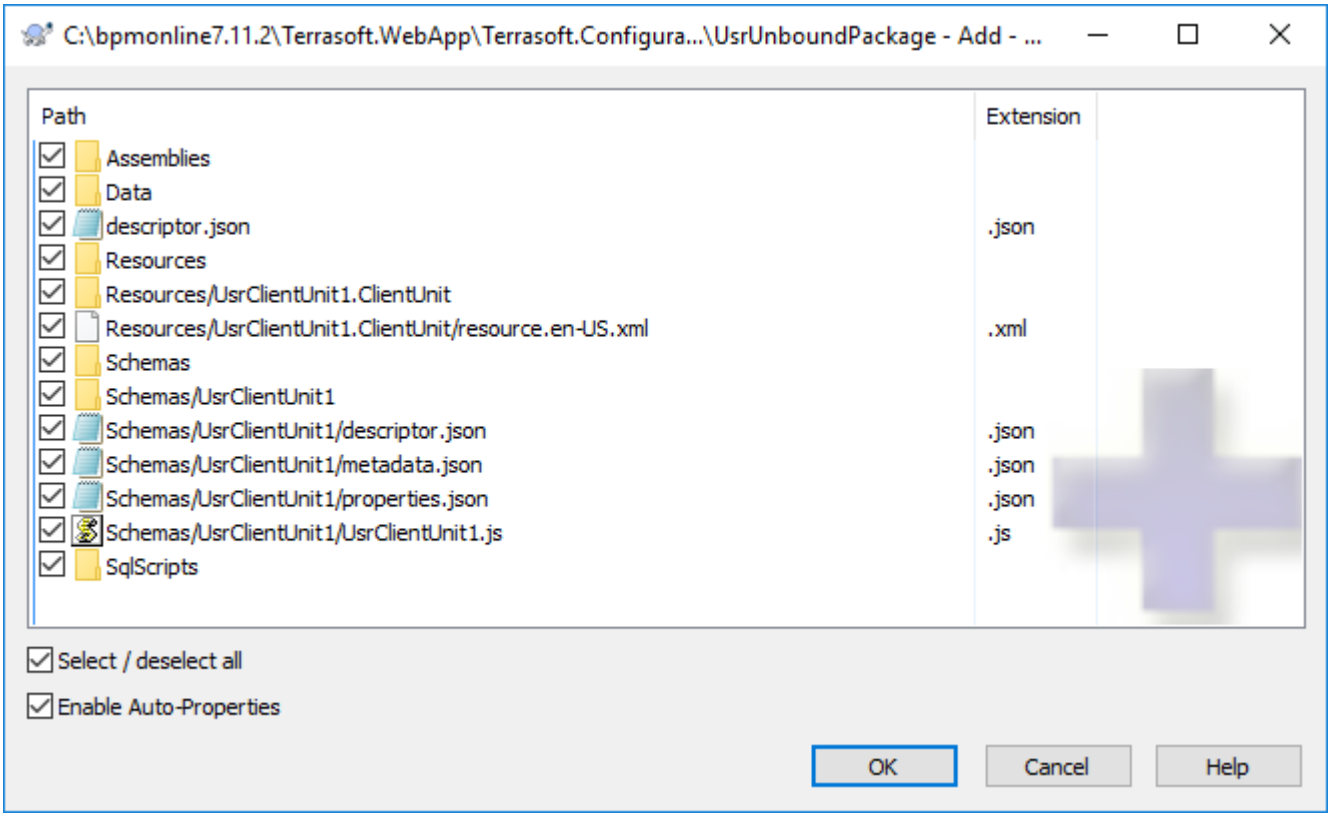
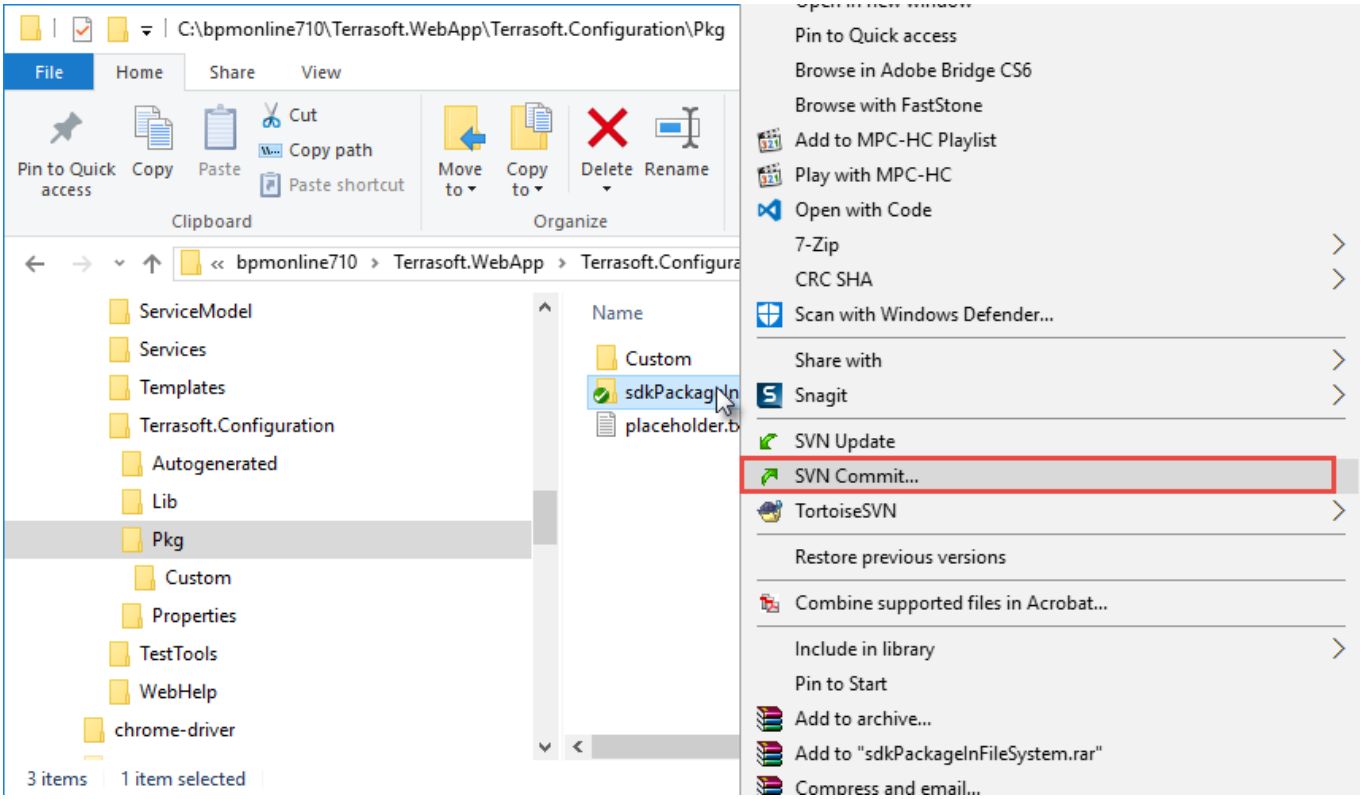
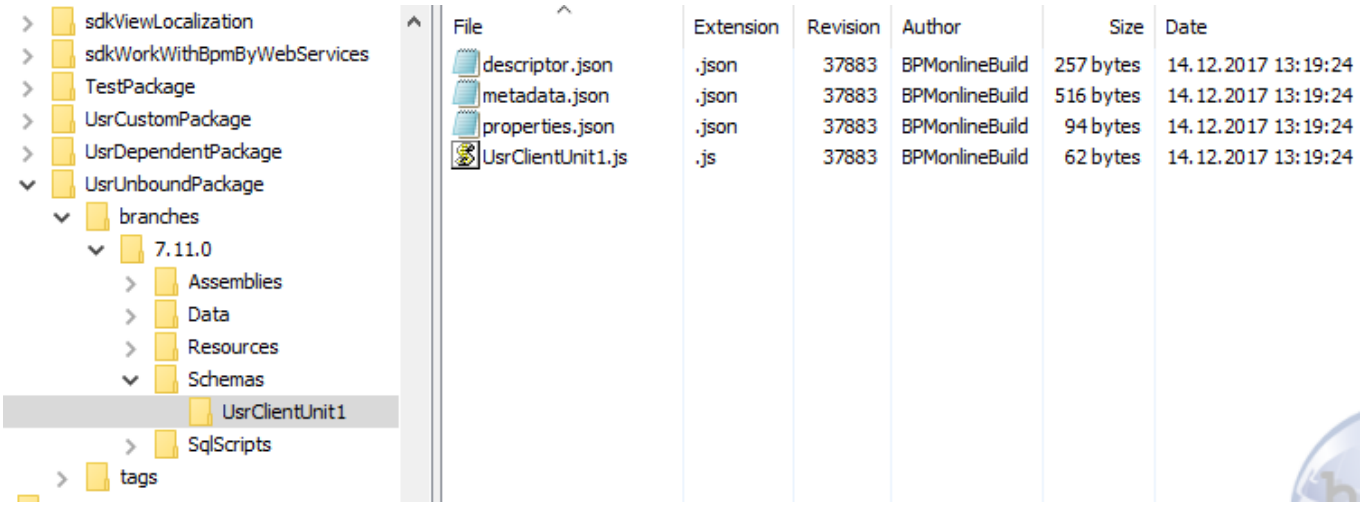


Fig. 12. Committing to the repository



As a result, all package contents will be bound to the SVN repository (Fig. 13).

Fig. 13. A package in SVN



> **NOTE**
>
> Repeat step 5 to commit new package schemas in the SVN repository.

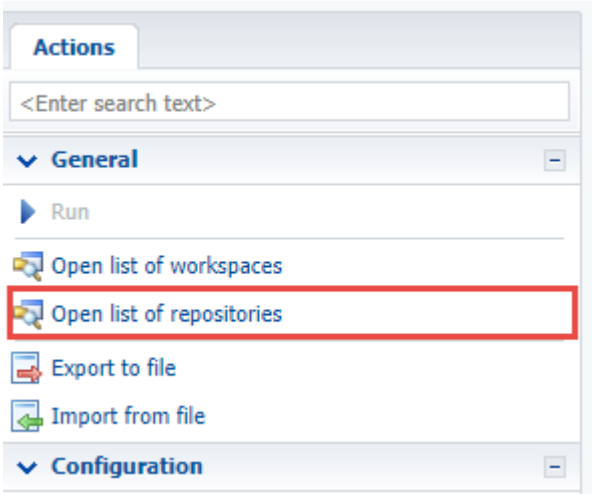
Alternative implementation

1. Add information about the SVN repository in the [Configuration] section.

If the information about the needed repository has not been added in the [Configuration] section:

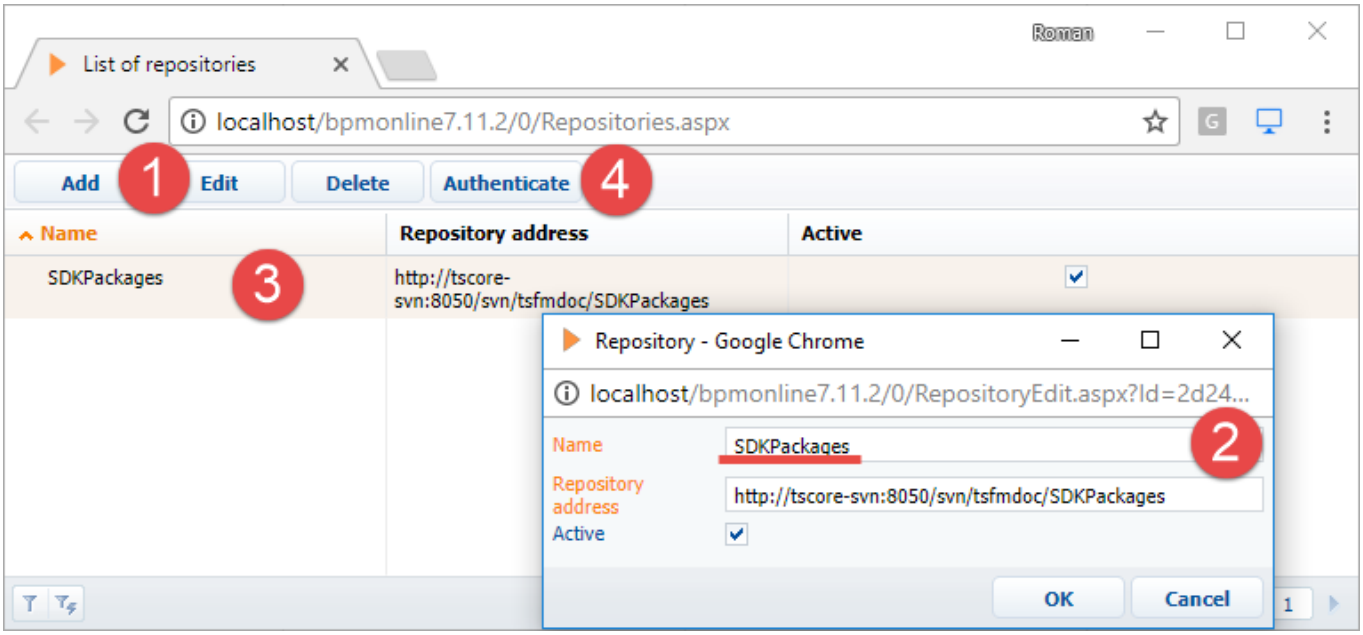
1. Run the [Open list of repositories] command (Fig. 14).

Fig. 14. The [Open list of repositories] command



2. In the opened [List of repositories], use the [Add] command (Fig. 15, 1) to add necessary repository (Fig. 15, 2). After this, repository information will display in the [List of repositories] window (Fig. 15, 3). Select the string with information on the repository anf perform authentication (Fig. 15, 4).

Fig. 15. The actions of the [Open list of repositories] window



2. Bind the repository to the package.

Execute repository binding SQL query. Example of the SQL query to binding repository to a package is as follows:

```
UPDATE SysPackage
SET
    [SysRepositoryId] =
    (
        select top 1 Id from SysRepository
        where Name = 'SDKPackages'-- Repository name.
    )
where [Name]='UsrUnboundPackage'-- Name of the custom package.
```

In this query, "SDKPackages" is the repository name (Fig. 15, 2), and "UsrUnboundPackage" is the name of the custom package.

> ⚠ ATTENTION
>
> To apply the changes in the repository, log out from the application and then log back in.

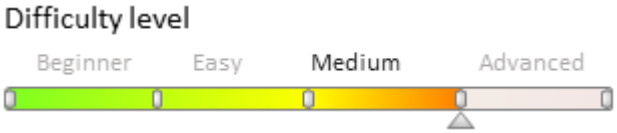
3. Export the package to the file system.

In the [Configuration] section, run the [Download packages to file system] command (Fig. 2). As a result, the bound package will be exported to the following catalog:
...\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\UsrUnboundPackage (Fig. 9).

See also

- **Working with SVN in the file system**
- **Creating a package in the file system development mode**
- **How to install an SVN package in the file system development mode**
- **Updating and committing changes to the SVN from the file system**
- **Creation of the package and switching to the file system development mode**

Updating and committing changes to the SVN from the file system



Introduction

Different developers can use the SVN to develop the functionality in the same file. The possible situation when one of the developers modifies the file first and the next developer could overwrite file with a new version and the modifications of the first developer will be lost. The modifications of the first developer are saved by the system, but these modifications will be lost in the last revision of the file. To avoid this, use one of the following versioning models: “Lock-Modify-Unlock” or “Copy-Modify-Merge” (see ”**Working with SVN in the file system**”).

Regardless of the chosen solution, the actions for updating and committing changes to the SVN are almost the same.

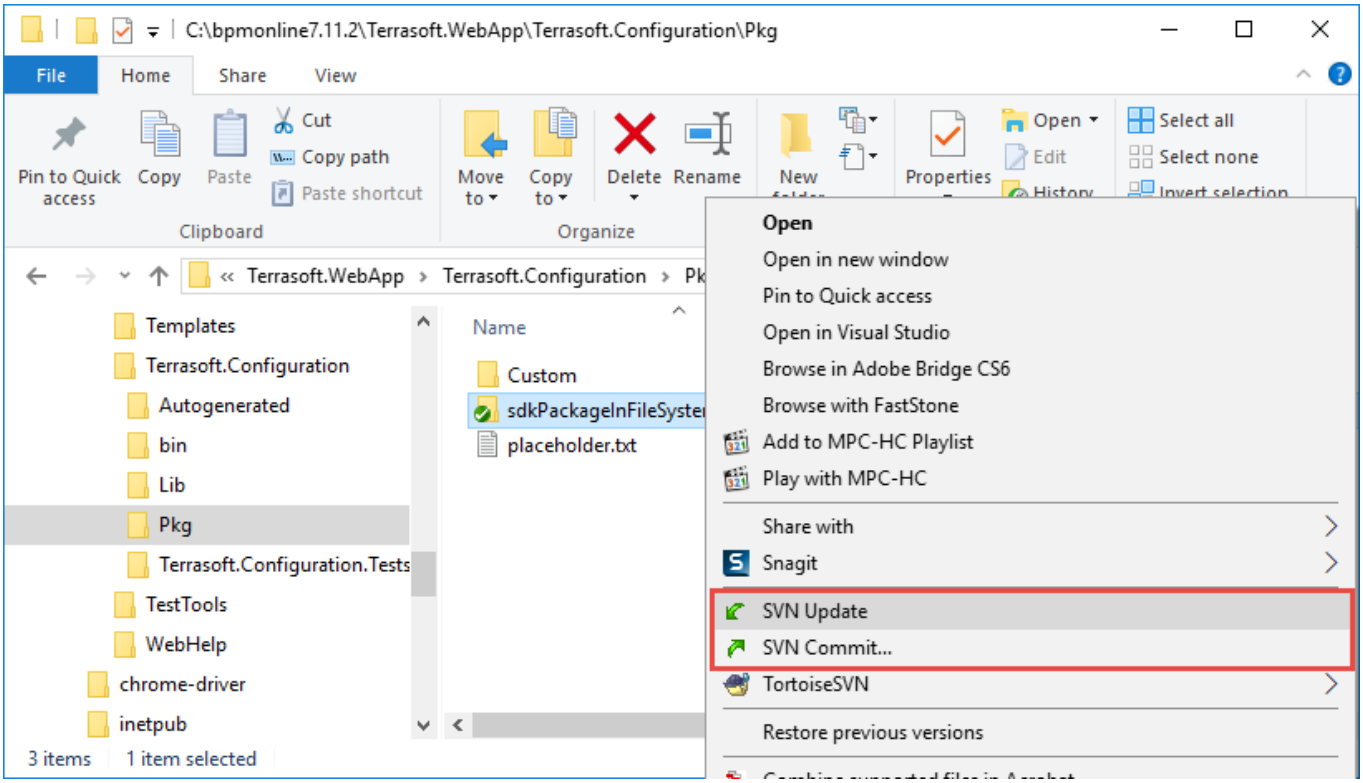
Case description

The *sdkPackageInFileSystem* custom package is implemented in the bpm'online configuration (see “**Creating a package in the file system development mode**”). Update the package in the file system development mode and after that commit it to the SVN.

Updating the package

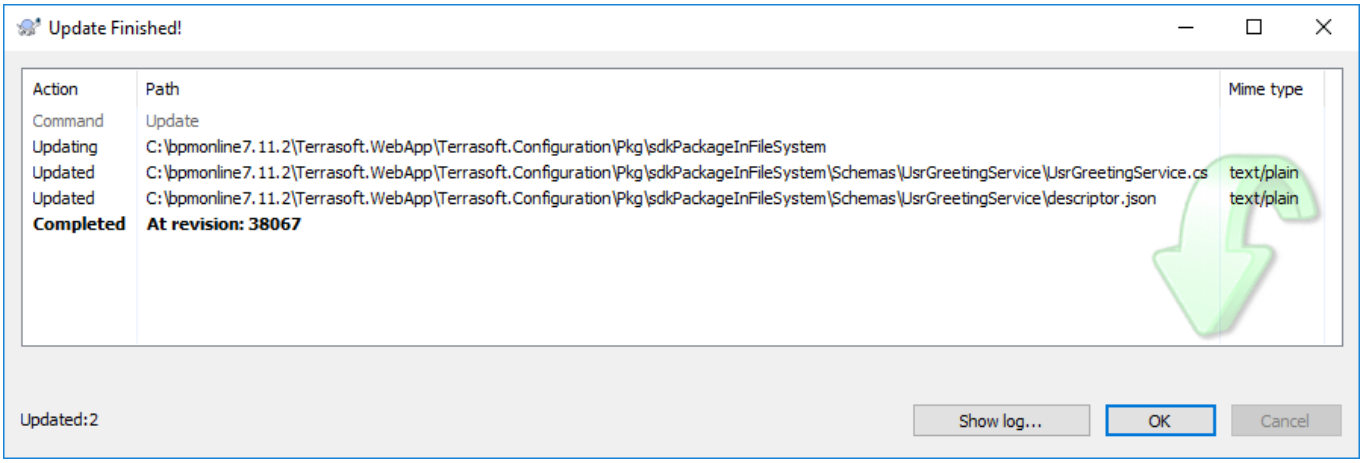
To get the latest package revision select the corresponding package in the *..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg* folder and perform the [SVN Update] action (Fig.1).

Fig. 1. Running the [SVN Update] action



After the action is executed, the window with update results will be displayed (Fig. 2).

Fig. 2. Update results



As a result the descriptor.json file (the package modification date has changed) and the UsrGreetingService schema were modified. The source code of the schema is available below:

```
namespace Terrasoft.Configuration
{
    using System.ServiceModel;
    using System.ServiceModel.Activation;
    using System.ServiceModel.Web;
    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class UsrGreetingService : System.Web.SessionState.IReadOnlySessionState
    {
        [OperationContract]
        [WebInvoke(Method = "GET", UriTemplate = "Hello")]
        public string TestHello()
        {
```

```
            return "Hello!";
        }
    }
}
```

ATTENTION

To implement changes to the application database, execute the [Update packages from file system] action in the **[Configuration] section**. If the schemas of an object or a source code were modified, execute steps 3 – 6 of the “**How to install an SVN package in the file system development mode**” article to apply changes.

Package contents modifications

After updating the package you can modify its contents. For example, add the TestHelloWorld() method to the source code of the UsrGreetingService.cs schema.

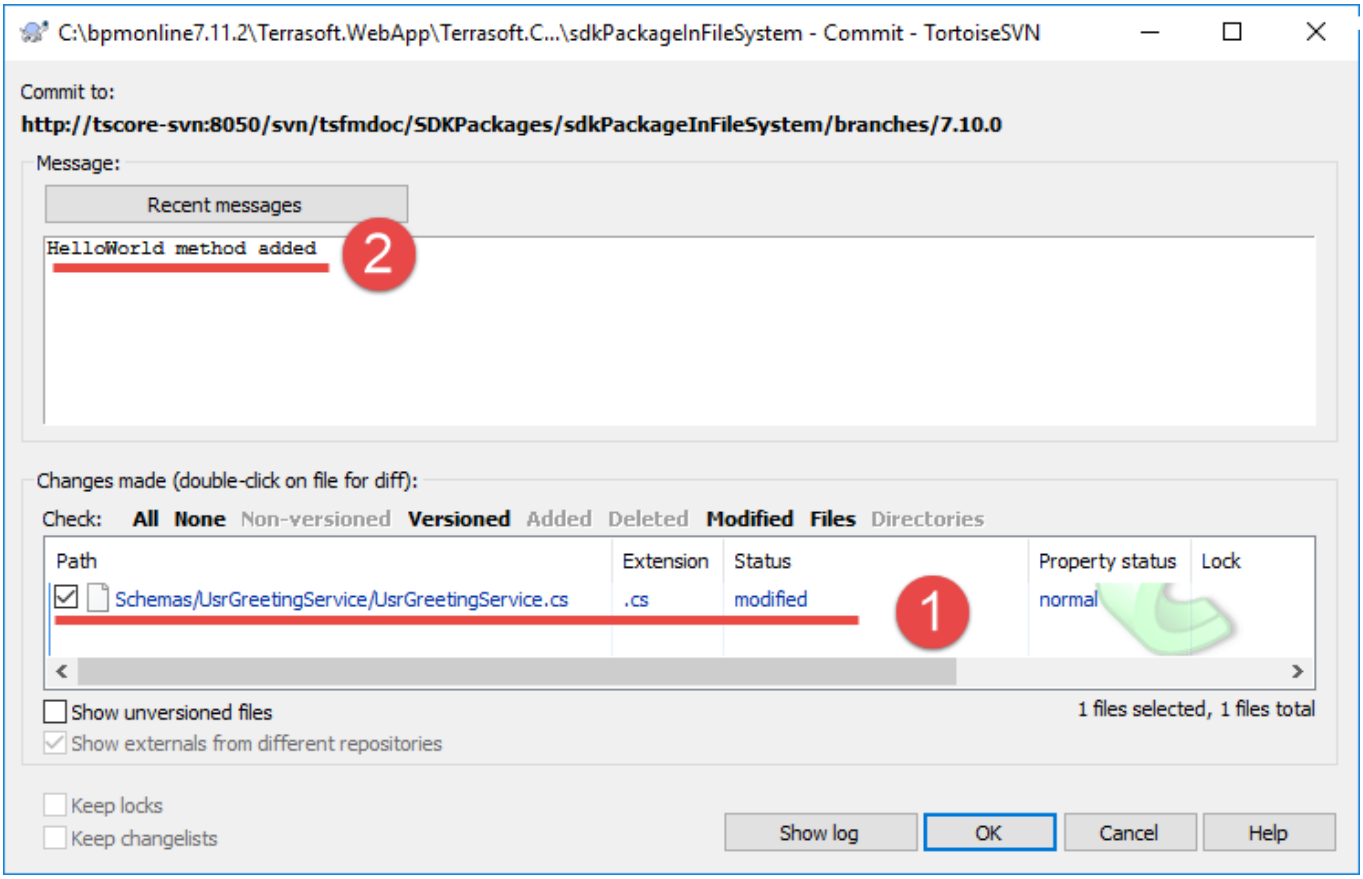
```
namespace Terrasoft.Configuration
{
    using System.ServiceModel;
    using System.ServiceModel.Activation;
    using System.ServiceModel.Web;
    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class UsrGreetingService : System.Web.SessionState.IReadOnlySessionState
    {
        [OperationContract]
        [WebInvoke(Method = "GET", UriTemplate = "Hello")]
        public string TestHello()
        {
            return "Hello!";
        }

        [OperationContract]
        [WebInvoke(Method = "GET", UriTemplate = "HelloWorld")]
        public string TestHelloWorld()
        {
            return "Hello world!";
        }
    }
}
```

Committing a package to storage

To commit the modifications in the SVN select the corresponding package in the ..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg folder and perform the [SVN Commit...] action. (Fig. 1).

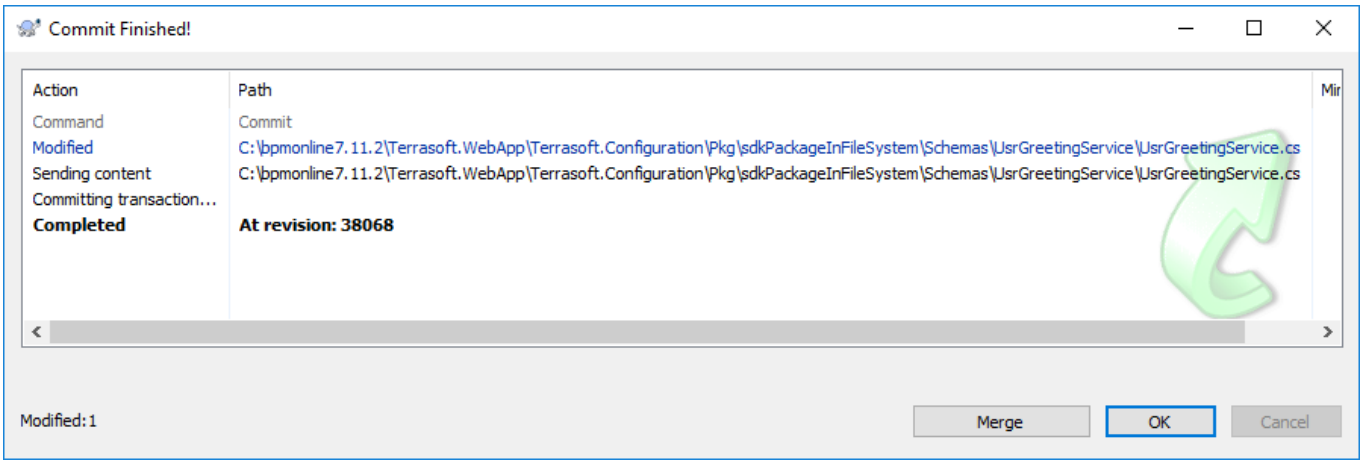
After that the revision properties window with modified files (1) will be displayed (Fig. 3). In this window you can add the log message with description of changes for the current revision (2). Click the [OK] button to start committing.

Fig. 3. Revision properties window



Committing process is displayed in the information window (Fig. 4).

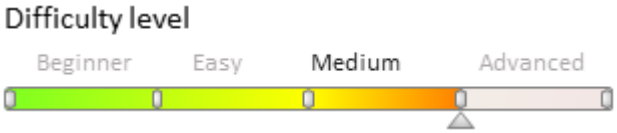
Fig. 4. Result of committing



See also

- **Working with SVN in the file system**
- **Creating a package in the file system development mode**
- **How to install an SVN package in the file system development mode**
- **How to bind existing package to SVN**
- **Creation of the package and switching to the file system development mode**

Creation of the package and switching to the file system development mode



Introduction

After performing the [Download packages to file system] action in the development in the file system mode all custom packages will be uploaded to the ..*Terrasoft.WebApp**Terrasoft.Configuration**Pkg* folder. The content of the custom package uploaded to the file system will not be bound to the SVN storage even if the package is bound to the storage in the [Configuration] section.

Fill out the [Revision control system storage] field to bind a package to the SVN storage (see “**Creating and installing a package for development**”). A working copy of the package will be created in the file system. A path to the folder where the work copies of the packages are being created is specified in the *defPackagesWorkingCopyPath* setting in the *ConnectionStrings.config* file (see “What parameters are used in ConnectionStrings.config”).

This feature can be used to create a package bound to SVN and used for development in the file system. If you specify a path to the ..*Terrasoft.WebApp**Terrasoft.Configuration**Pkg* folder in the *defPackagesWorkingCopyPath* setting, the package will be automatically bound to the SVN storage after it is uploaded to the file system.

To do this:

1. Specify a path to the ..*Terrasoft.WebApp**Terrasoft.Configuration**Pkg*. folder in the *defPackagesWorkingCopyPath* setting.

2. In the development mode, use the built-in tools to create a package in the [Configuration] section bound to the SVN storage.

3. Commit the package in the storage in the [Configuration] section.

4. Switch to the file system development mode

5. Export the package to the file system.

6. Add new elements of the package to the SVN storage.

Case description

In the development mode, use the built-in tools to create a custom package in the [Configuration] section bound to the SVN storage. Configure the bpm’online so that the content of the package in the development mode was bound to the SVN storage after the package upload.

> ⚠ ATTENTION
>
> The case requires understanding of the difference between development modes. In general: in the development mode in the file system, it is necessary to work with the SVN storage only from the file system, and in the development mode using the built-in tools it is necessary to work with SVN only via the built-in tools of the [Configuration] section.

Case implementation algorithm

1. Modify the defPackagesWorkingCopyPath setting

Specify a path to the ..*Terrasoft.WebApp**Terrasoft.Configuration**Pkg*. folder in the *defPackagesWorkingCopyPath* setting of the *ConnectionStrings.config* file. Example:

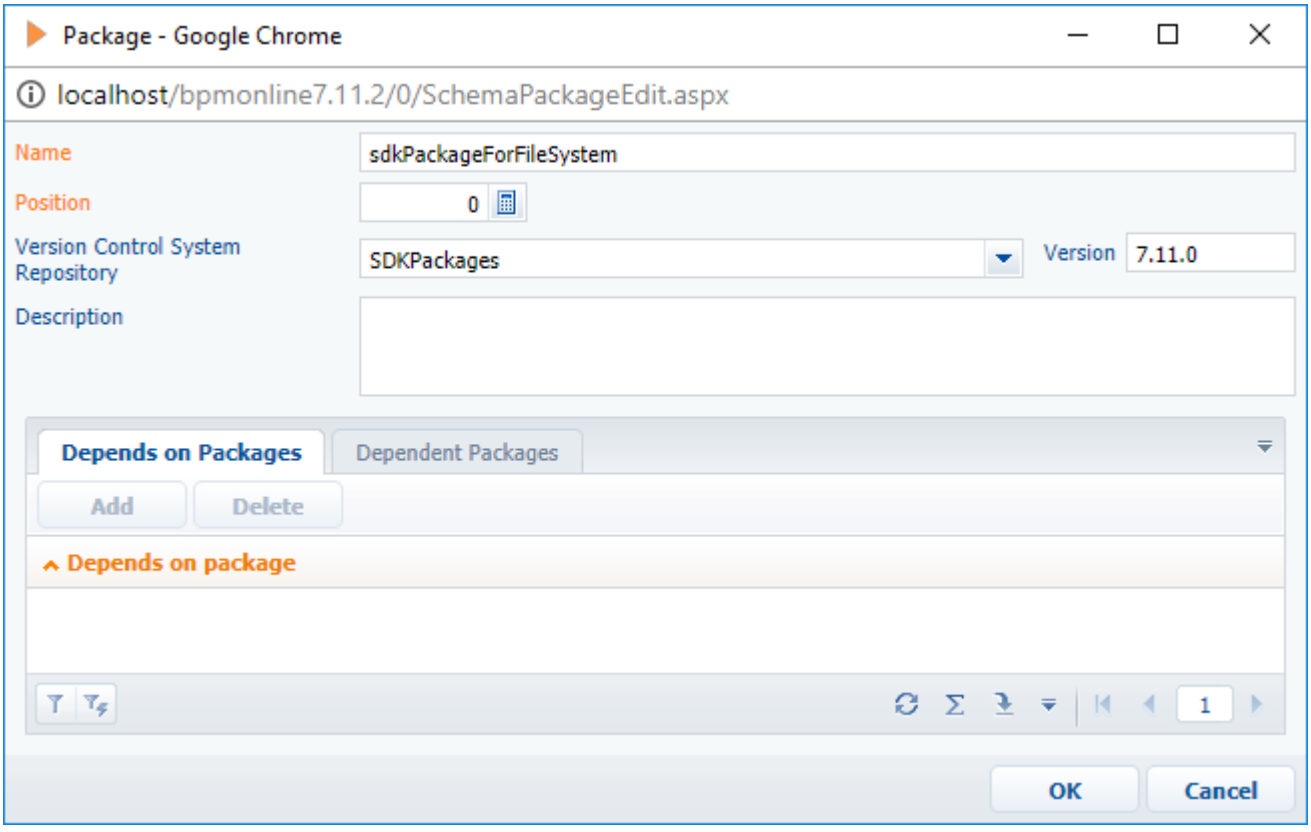
```
<?xml version="1.0" encoding="utf-8"?>
<connectionStrings>
  ...
  <add name="defPackagesWorkingCopyPath"
connectionString="C:\bpmonline7.11.2\Terrasoft.WebApp\Terrasoft.Configuration\Pkg/>
  ...
</connectionStrings>
```

This modification enables to combine the folder with working copies of custom packages with the folder in which the packages will be uploaded in the development in the file system mode.

2. Create a custom package

In the development mode, create a custom package in the [Configuration] section bound to the SVN storage via the built-in tools. Please refer to "**Creating and installing a package for development**" for any details. Specify the name, storage and version of the created package (Fig. 1).

Fig. 1. Package properties



> ATTENTION
>
> After creation of the package add necessary dependencies form the base packages (see "**Package dependencies. Basic application packages**").

3. Commit a package to storage

To commit a package to the storage perform the [Commit package to repository] action (Fig. 2). In the dialog box (Fig. 3) add the description of changes (1) and press the [OK] button. After the commit is complete, the corresponding message will appear (3).

Fig. 2. The [Commit package to storage] action

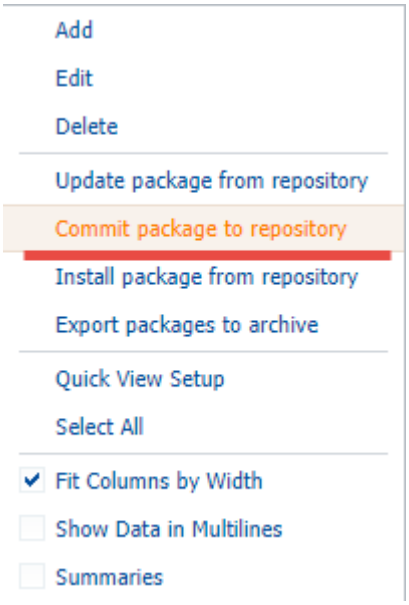
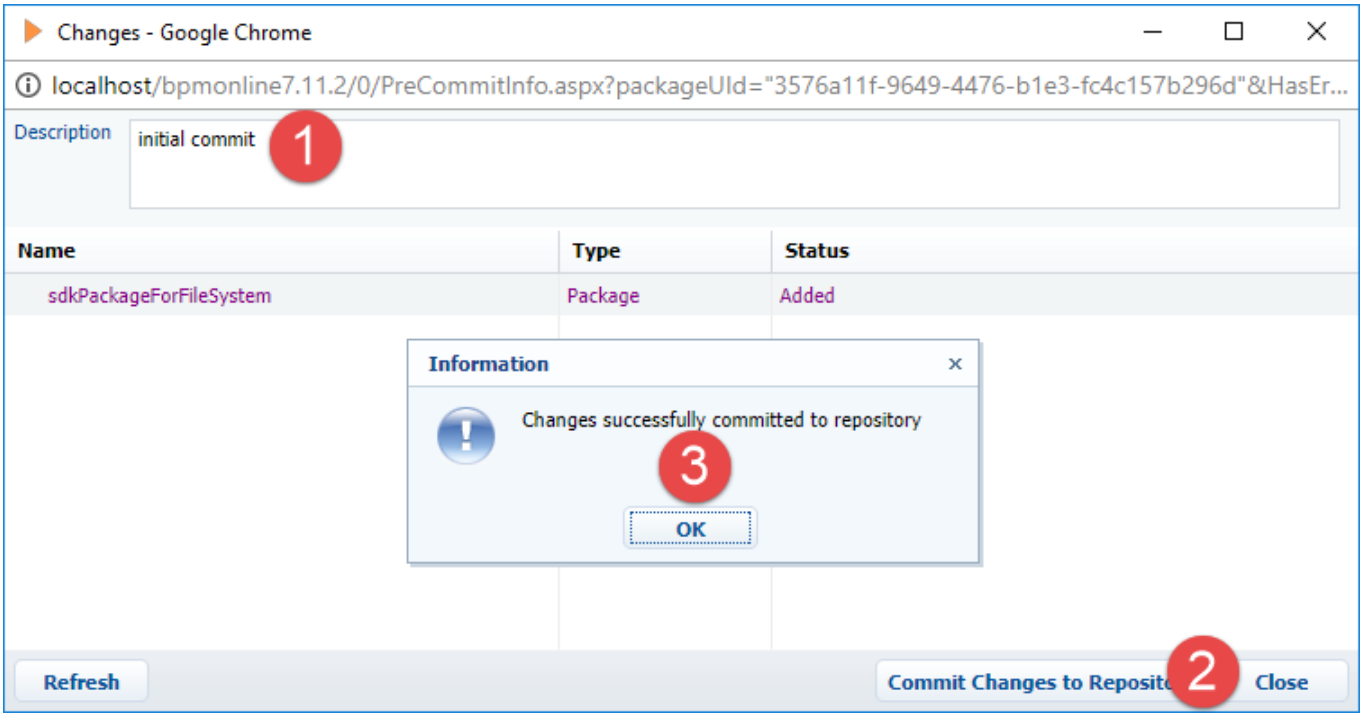
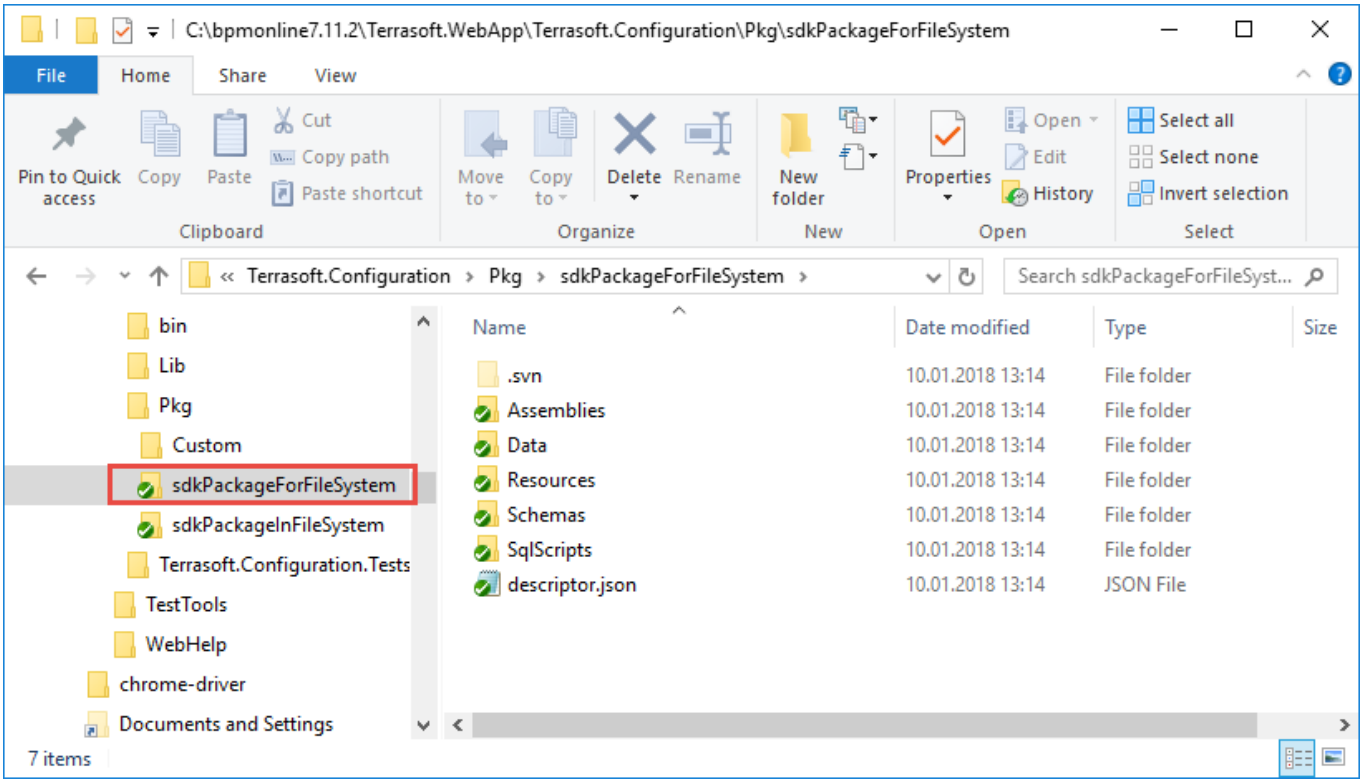


Fig. 3. Commit properties window



After the commit is complete, the ..*Terrasoft.WebApp**Terrasoft.Configuration**Pkg* catalog will contain a local working copy of the package (Fig. 4).

Fig. 4. Package working copy



4. Switch to the file system development mode.

To enable the development in the file system mode, edit the Web.config file in the application root folder and set *enabled* attribute of the *fileDesignMode* element to *true*.

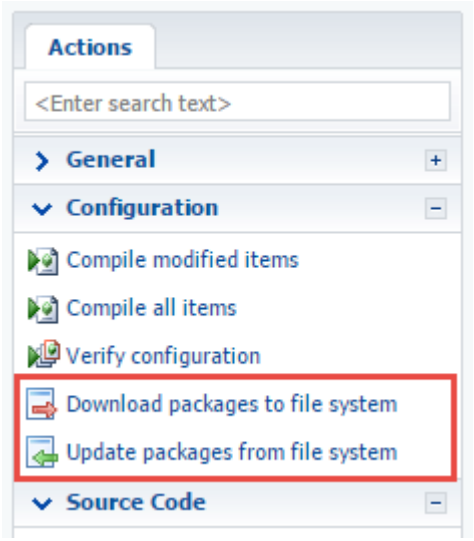
```
<fileDesignMode enabled="true"/>
```

> ⚠ ATTENTION
>
> Disable the using of the static content (see "**Client static content in the file system**").

After the development in the file system mode is enabled, two buttons will appear on the [Actions] tab in the [Configuration] section (Fig. 1):

- [Download packages to file system] – exports the packages from the application database to the following directory: ...*Terrasoft.WebApp**Terrasoft.Configuration**Pkg*.
- [Update packages from file system] – imports the packages from the following catalog: ...*Terrasoft.WebApp**Terrasoft.Configuration**Pkg* to the database.

Fig. 5. Actions in the [Configuration] section for development in the file system



5. Export the package to the file system.

> NOTE
>
> If the package content was not changed after committing to the storage, this action is optional.

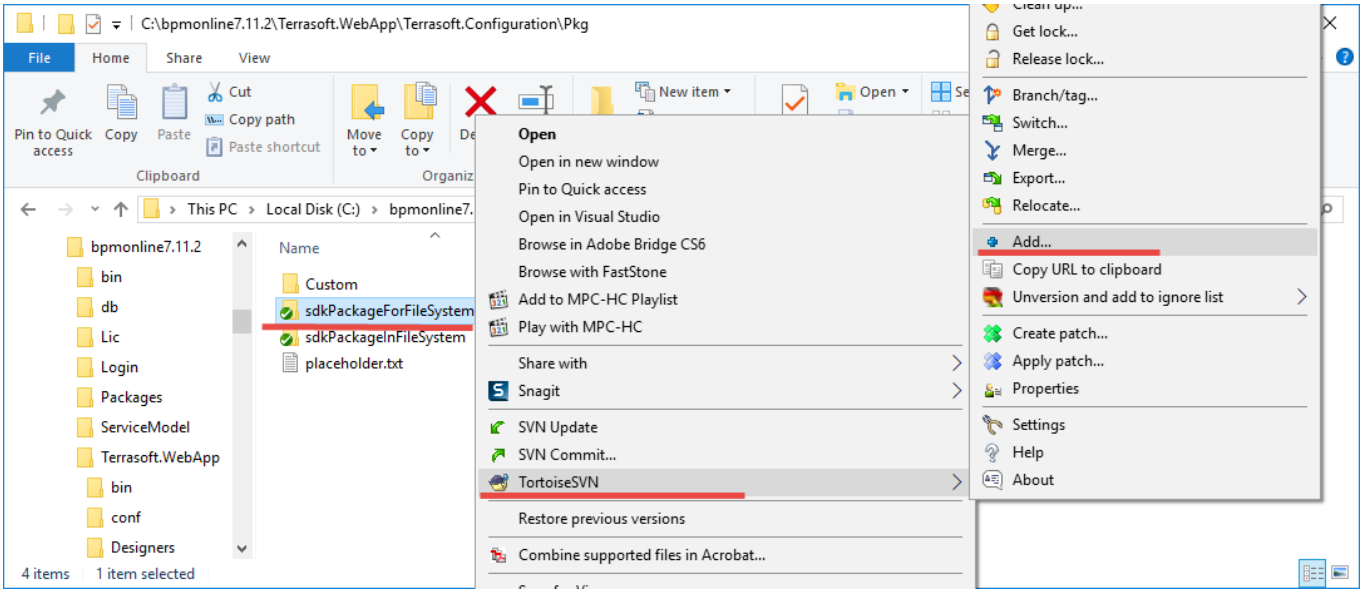
Perform the [Download packages to file system] action to download packages to the file system. As a result, all elements of the package that were created or modified via built-in tools in the [Configuration] section will be downloaded to the file system to the *..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg* folder.

> ATTENTION
>
> Since in the development in the file system mode the built-in tools for working with SVN are disabled, the new elements of the package will not be bound to the storage.

6. Add new elements of the package to the SVN storage

To add the new elements of the package to the storage, select the folder of the package working copy and perform the [Add...] command of the SVN application (for example, *TortoiseSVN*) (Fig. 6).

Fig. 6. Command of adding the elements to the storage



After this the dialog box with selection of the elements to add will be displayed (Fig. 7). Select the necessary

elements and click the [OK] button, after that the window with the results of command execution will be displayed (Fig. 8).

Fig. 7. Selection dialog box

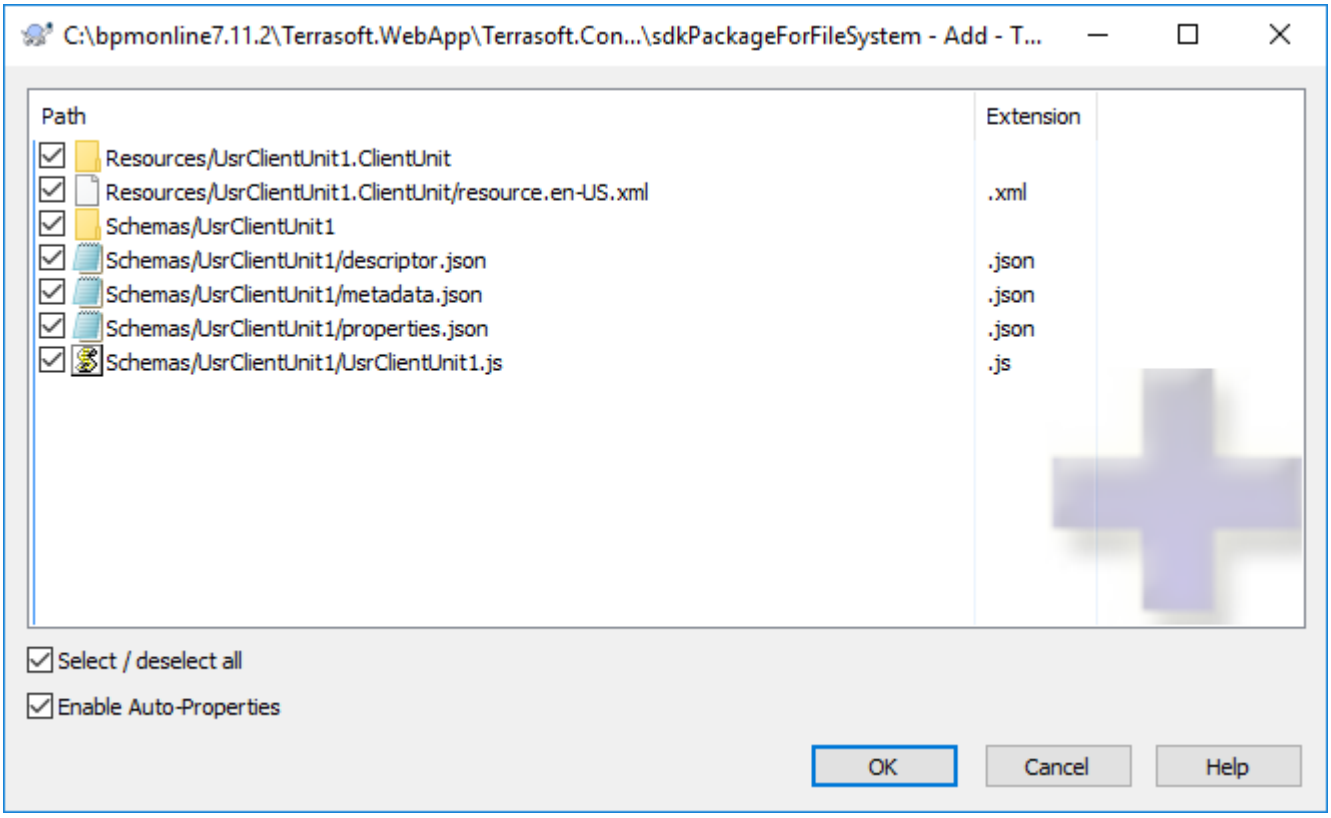
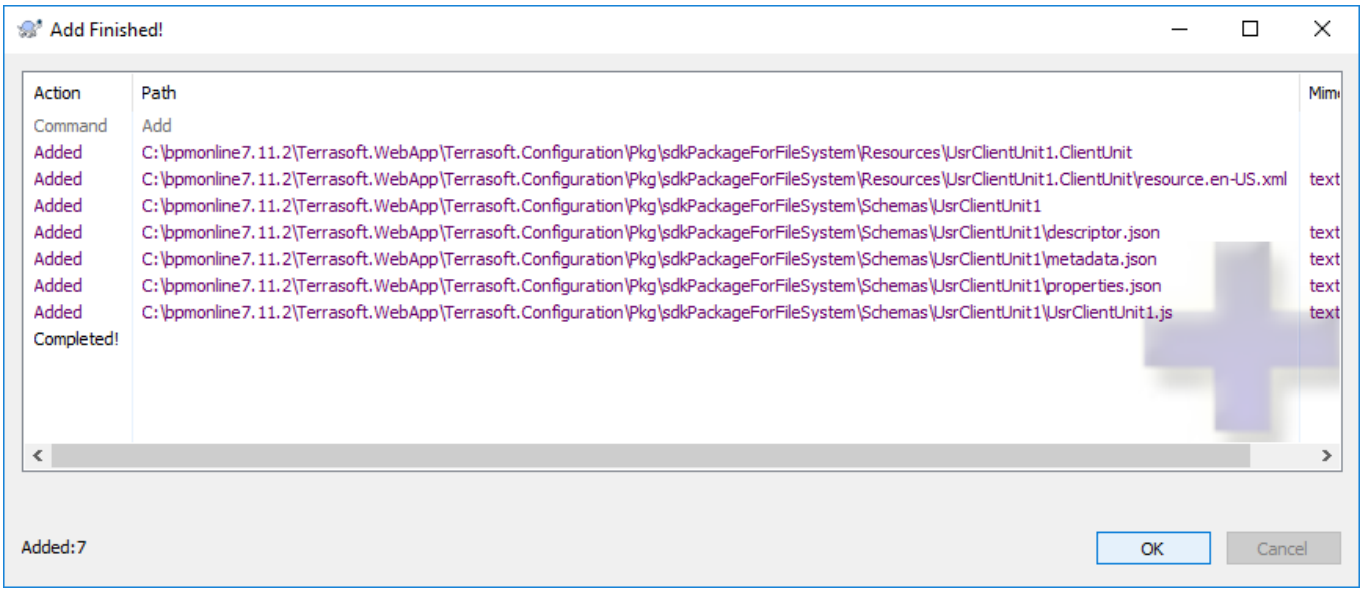
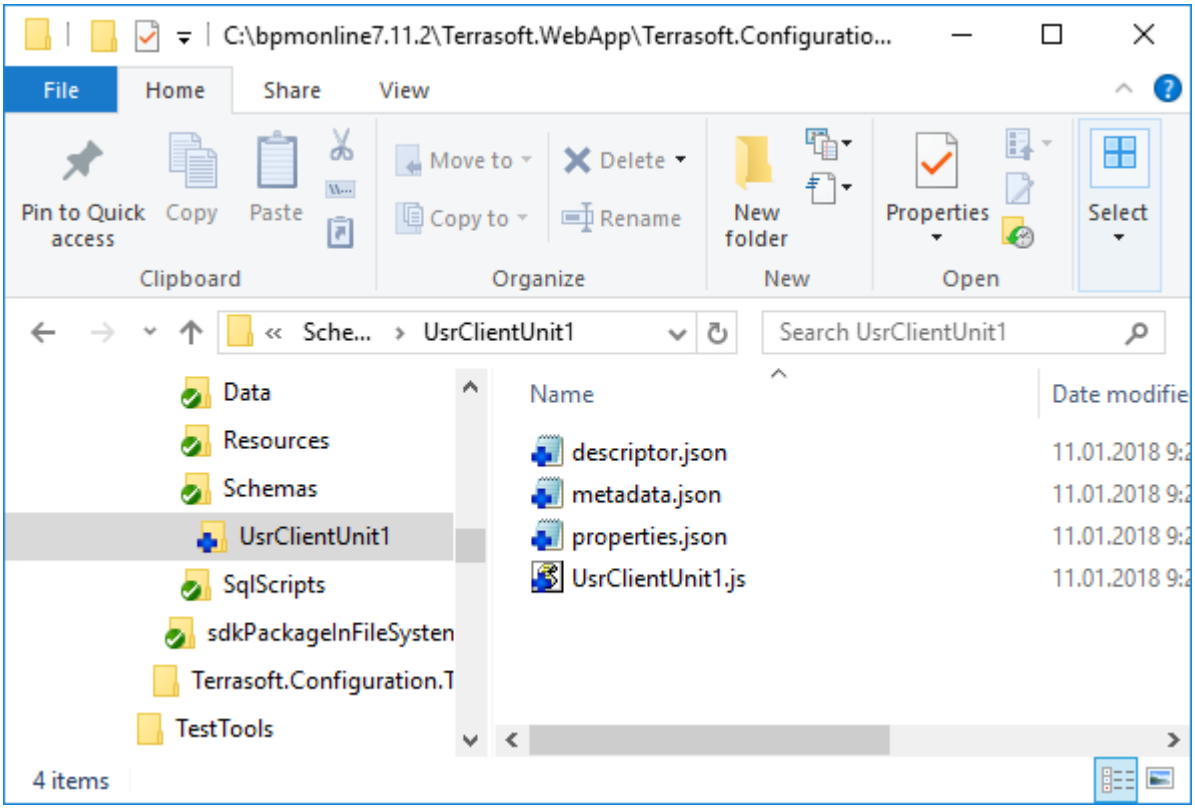


Fig. 8. Information window



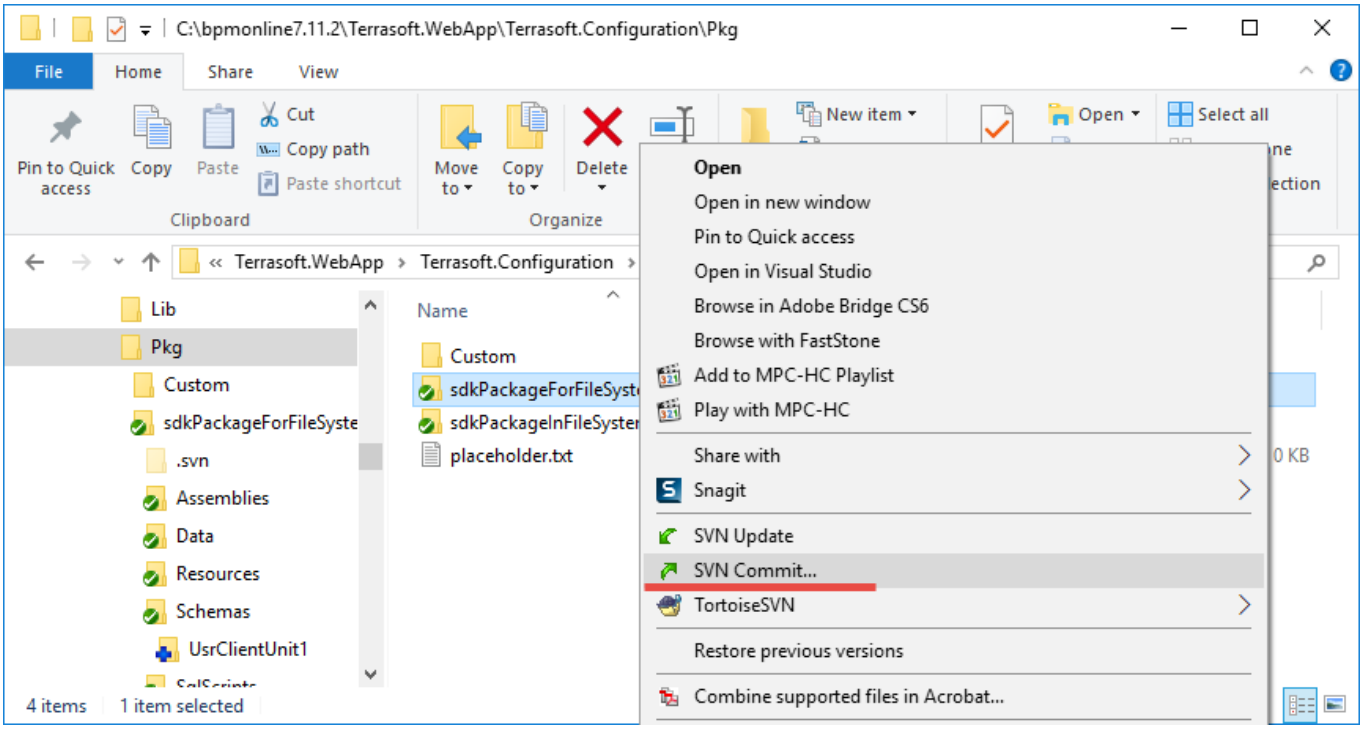
Added elements will be marked as bound but not committed to the SVN storage (Fig. 9).

Fig. 9. Displaying of the added but not committed package elements



Perform the [SVN Commit...] command to commit all modified elements of the package in the storage. (Fig. 10).

Fig. 10. Command of adding the elements to the storage

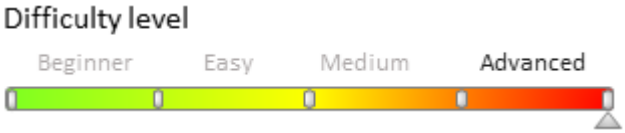


> **NOTE**
>
> More information about committing the elements in the storage can be found in the "**Updating and committing changes to the SVN from the file system**" article.

See also

- **Working with SVN in the file system**
- **Creating a package in the file system development mode**
- **How to install an SVN package in the file system development mode**
- **How to bind existing package to SVN**
- **Updating and committing changes to the SVN from the file system**

Developing the configuration server code in the user project



Introduction

Before the 7.11.1 version, only the preconfigured Visual Studio solution which is distributed with bpm’online, was used to develop configuration server code in the file system More information about the Terrasoft.Configuration.sln solution and development of the server code in the file system is given in the "**Development in the file system**" and "**Working with the server side source code in Visual Studio**" articles.

This development approach is inconvenient because of low performance connected with recompilation of all bpm’online configuration (Terrasoft.Configuration.dll). This is significantly if the application contains several bpm'online products. In addition, the development of server code in the file system could only be performed by interacting with the database of the application deployed on-site.

Due to the described inconveniences, this approach can be efficiently used to perform complex configuration revision of the bpm'online. It is more efficient to use the built-in bpm'online development tools to develop a simple server code (see the "**Built-in development tools**" article). But the built-in development tools do not support full IDE functions: debugging, IntellSense, Refactoring, etc.

Starting with version 7.11.1 the bpm’online you can develop simple server code in the Visual Studio custom projects.

To develop and debug separate classes or small blocks of server functionality, you can create a separate class library project and configure it. Then, connect corresponding bpm’online class libraries (for example, Terrasoft.Core) and perform development and debugging of the server code. To debug and test the development result you can use a local database (use the WorkspaceConsole utility to connect to the database) or an application located in the cloud by connecting to it via the Executor utility.

Advantages of this approach:

- High speed of testing modifications, compiling and execution
- Full usage of the Visual Studio functions
- Ability of using any tools for Continuous Integration, for example Unit testing.
- Simplicity of configuration – you do not need configuration source codes
- You can use the database of an application deployed on-site or in Cloud.

Preliminary settings

For connecting the libraries of the bpm'online classes, deploying the local database from an archive copy and working with the WorkspaceConsole utility, you can use the bpm'online installed locally. In all examples of this article used the bpm’online installed to the *C:\bpmonline7.11.1* local folder.

The Executor utility located in the *C:\Executor* folder is used as an example of working with the bpm’online Cloud service. You can use following link to download the utility configured for processing the example.

Development of the configuration server code for on-site

application

If you have an access to the bpm’online local database, to develop configuration server logic do the following:

1. Restore the database from a backup (if need)

The process of restoring the bpm’online database from backup is described in the "Installing bpm’online application" article. Backup of the application database is located in the *db* folder of the application (for example, *C:\bpmonline7.11.1\db*).

2. Configure the WorkspaceConsole utility

To operate with the database, you need to configure the WorkspaceConsole utility using the application files. More information about configuration of the utility is described in the "**WorkspaceConsole settings**" article. To configure the utility:

- Open the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder of the application (for example, *C:\bpmonline7.11.1\Terrasoft.WebApp\DesktopBin\WorkspaceConsole*).
- Execute one of the .bat files: PrepareWorkspaceConsole.x64.bat or PrepareWorkspaceConsole.x86.bat, depending on the Windows version.

> NOTE
>
> Ensure that the SharpPlink-xXX.svnExe and SharpSvn-DB44-20-xXX.svnDll files were copied to the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder from the corresponding folder (x64 and x86) after executing the .bat file.

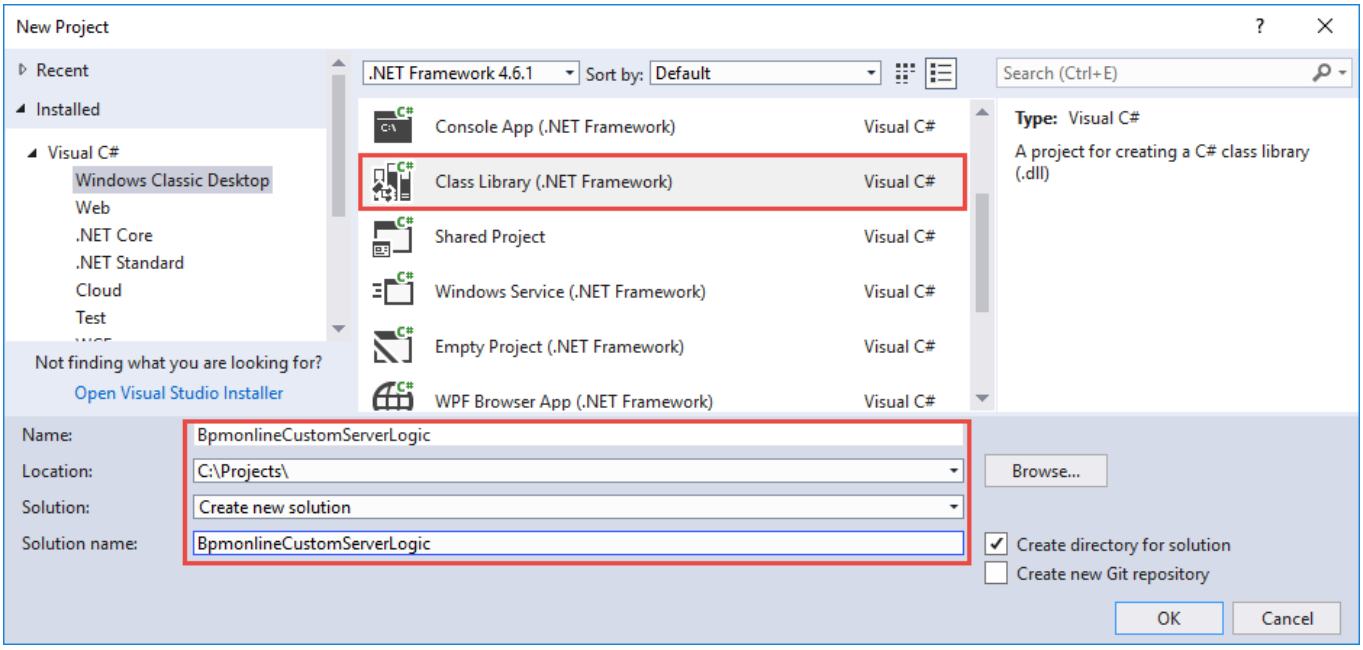
- Specify parameters of connection to the database in the Terrasoft.Tools.WorkspaceConsole.exe.config file from the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder of the application (for example, *C:\bpmonline7.11.1\Terrasoft.WebApp\DesktopBin\WorkspaceConsole*). For example, if the *bpmonline7.11.1DB* database is deployed on the *dbserver* server, the connection string will be as follows:

```
<connectionStrings>
<add name="db" connectionString="Data Source=dbserver; Initial
Catalog=bpmonline7.11.1DB; Persist Security Info=True; MultipleActiveResultSets=True;
Integrated Security=SSPI; Pooling = true; Max Pool Size = 100; Async = true;
Connection Timeout=500" />
</connectionStrings>
```

3. Create and configure Visual Studio project

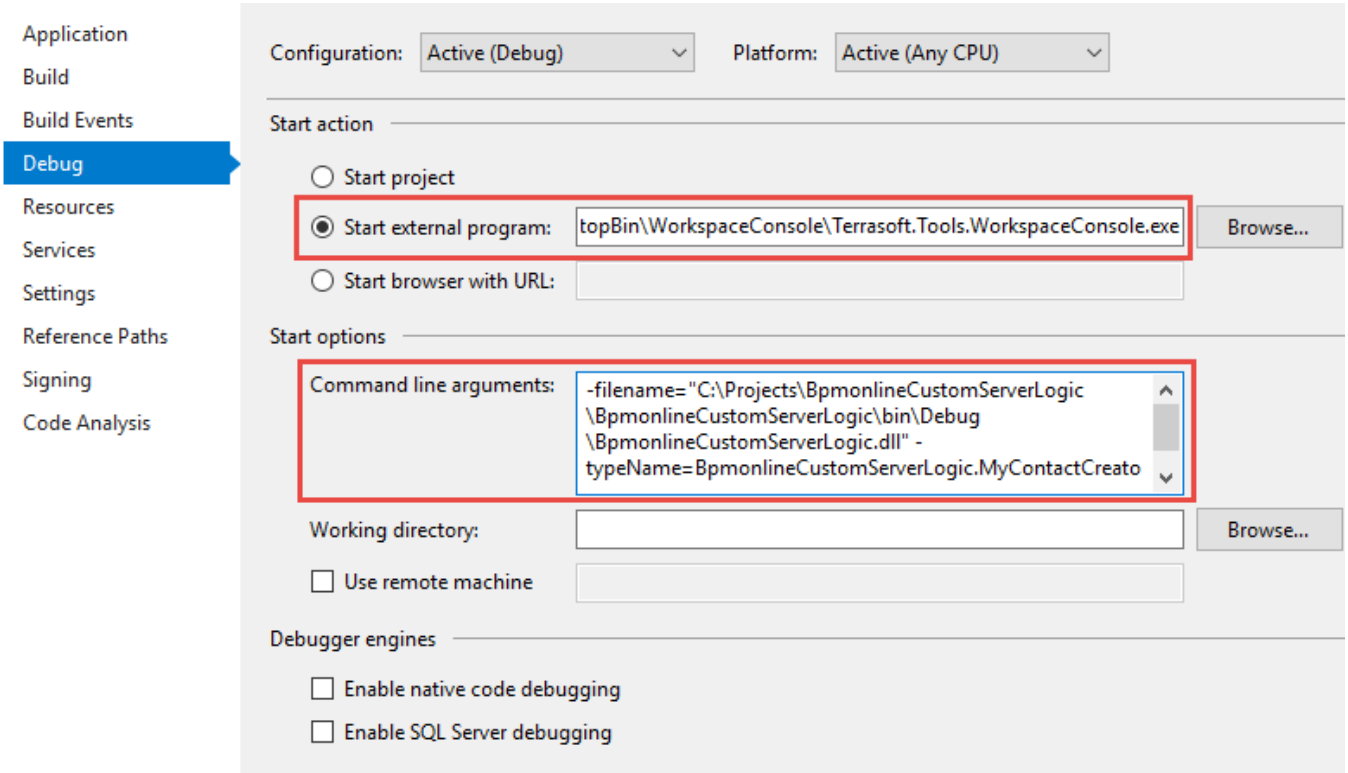
For this, create standard class library project (Fig. 1). More information about creating a new Visual Studio solution and managing projects is described in the "Solutions and Projects in Visual Studio" Microsoft documentation article.

Fig. 1. Creating the solution and project of the classes library in the Visual Studio



On the [Debug] tab of the properties window of the created class library project, specify the full path to the configured WorkspaceConsole utility in the [Start external program] property (Fig. 2). The WorkspaceConsole is used as the external application for debugging the developed program logic.

Fig. 2. The [Debug] tab properties



In the [Command line arguments] properties specify following launch arguments of the WorkspaceConsole.

- *filename* – full path to the debug version of developed class library.
- *typeName* – full name of the class in which the program logic (including the names of all namespaces) is being developed. For example, *BpmonlineCustomServerLogic.MyContactCreator*.
- *Operation* – WorkspaceConsole operation. The "ExecuteScript" value should be specified.
- *workspaceName* – the workspace name. The "Default" value should be specified.

The example of the WorkspaceConsole launch arguments:

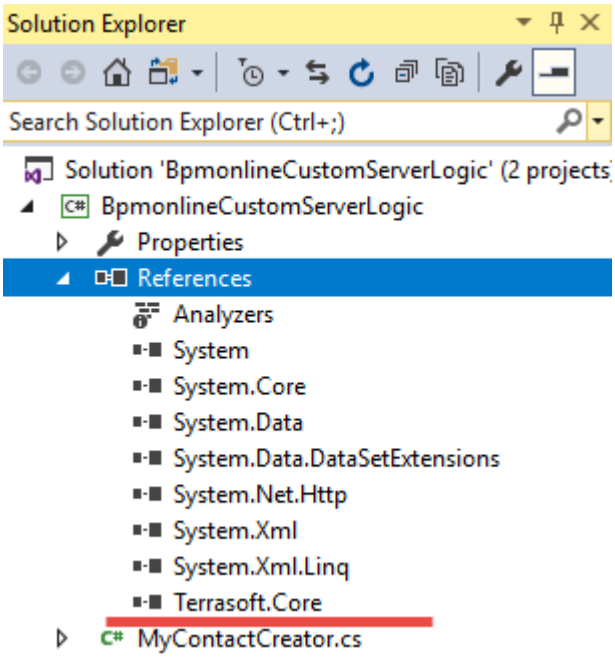
```
-
filename="C:\Projects\BpmonlineCustomServerLogic\BpmonlineCustomServerLogic\bin\Debug
\BpmonlineCustomServerLogic.dll" -
typeName=BpmonlineCustomServerLogic.MyContactCreator -operation=ExecuteScript -
workspaceName=Default
```

More information can be found in the "**WorkspaceConsole parameters**" article.

> ATTENTION
>
> In the properties of the Visual Studio project that operates with the bpm'online 7.11.0 or higher, you need to specify the version of the .NET Framework 4.7 (the [Target framework] property of the [Application] tab).

To work with the classes of the server side of bpm’online core, set the dependencies from the necessary bpm’online class libraries in the created project. For example, add the dependency from the Terrasoft.Core.dll library (Fig. 3). More information about adding the dependencies can be found in the "Managing references in a project" Microsoft documentation article.

Fig. 3. Terrasoft.Core library in the project dependencies



Class libraries of the bpm’online namespace can be found in the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder of the application.

> NOTE
>
> Class libraries are being copied to the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder when executing the .bat files (see Step 2. Configure the WorkspaceConsole utility").

4. Develop the functions

For this, add a new class to the created class library project. The name of the class should match the name specified in the *typeName* launch argument of the WorkspaceConsole (for example, *BpmonlineCustomServerLogic.MyContactCreator*). Class should implement the *Terrasoft.Core.IExecutor* interface.

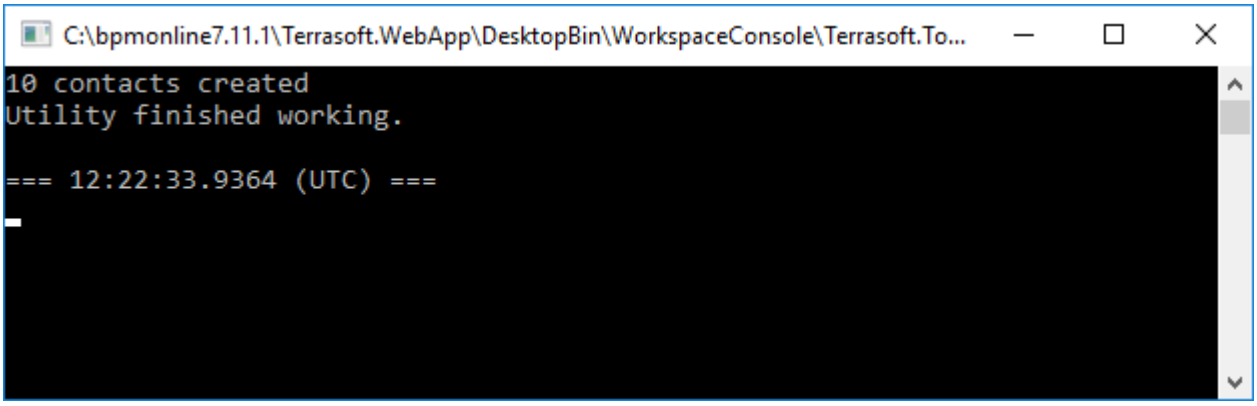
The implementation of the class is available below:

```
using System;
using Terrasoft.Core;
```

```
namespace BpmonlineCustomServerLogic
{
    public class MyContactCreator : IExecutor
    {
        public void Execute(UserConnection userConnection)
        {
            // Getting an instance of the [Contacts] schema.
            var schema =
userConnection.EntitySchemaManager.GetInstanceByName("Contact");
            var length = 10;
            for (int i = 0; i < length; i++)
            {
                // Create a new contact.
                var entity = schema.CreateEntity(userConnection);
                // Set contact properties.
                entity.SetColumnValue("Name", string.Format("Name {0}", i));
                entity.SetDefColumnValues();
                // Save the contact to the database.
                entity.Save(false);
            }
            // Output message to the console.
            Console.WriteLine($"{length} contacts created");
        }
    }
}
```

After running the project (F5 key) the WorkspaceConsole window with the corresponding message will be displayed (Fig. 4).

Fig. 4. Displaying the result of running the program in the WorkspaceConsole window.



You can also set a breakpoint on any line of the source code and view the current values of variables at the time of program execution (ie, debuging). More information about breakpoints in the Visual Studio can be found in the Use Breakpoints in the Visual Studio Debugger" Microsoft documentation article.

The result of execution the above code can be found in the [Contacts] section of the bpm'online application (Fig. 5) or by executing the request to the database (Fig. 6).

Fig. 5. Added contacts

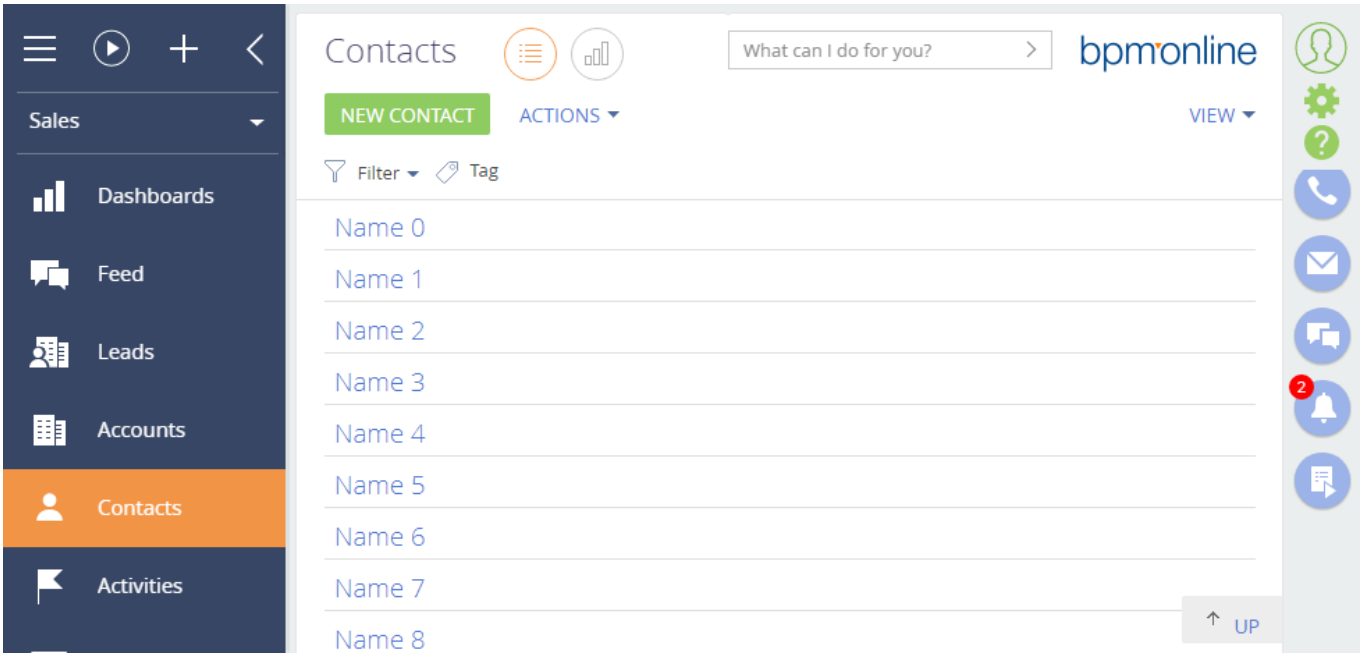
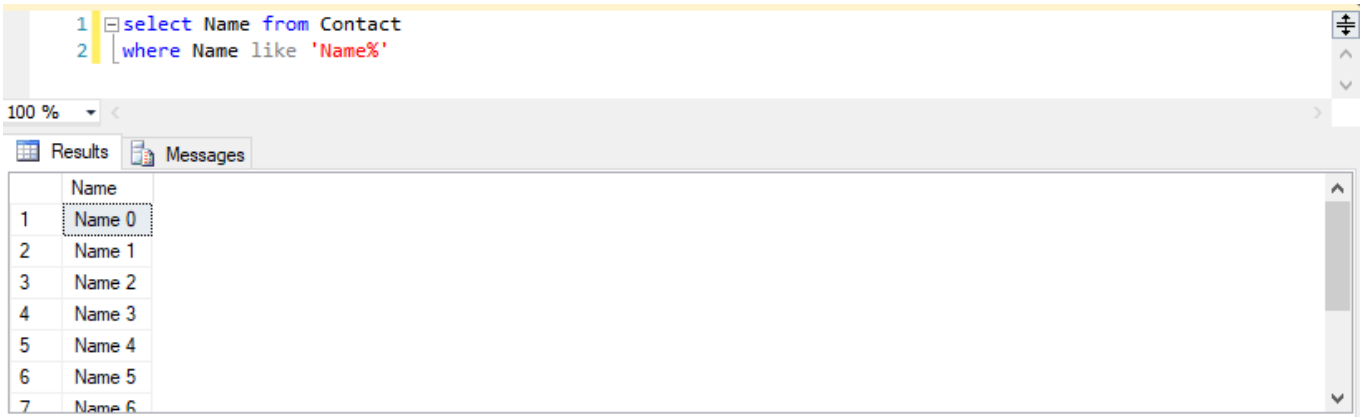


Fig. 6. Request to the table of contacts of the database



Development of the configuration server code for Cloud application

To develop configuration server logic without direct access to the bpm'online database:

1. Create class library project.

Create standard class library project (Fig. 1). More information about creating a new Visual Studio solution and managing projects is described in the "Solutions and Projects in Visual Studio" Microsoft documentation article. Set the name of the project (for example, "BpmonlineCustomServerLogic.Cloud").

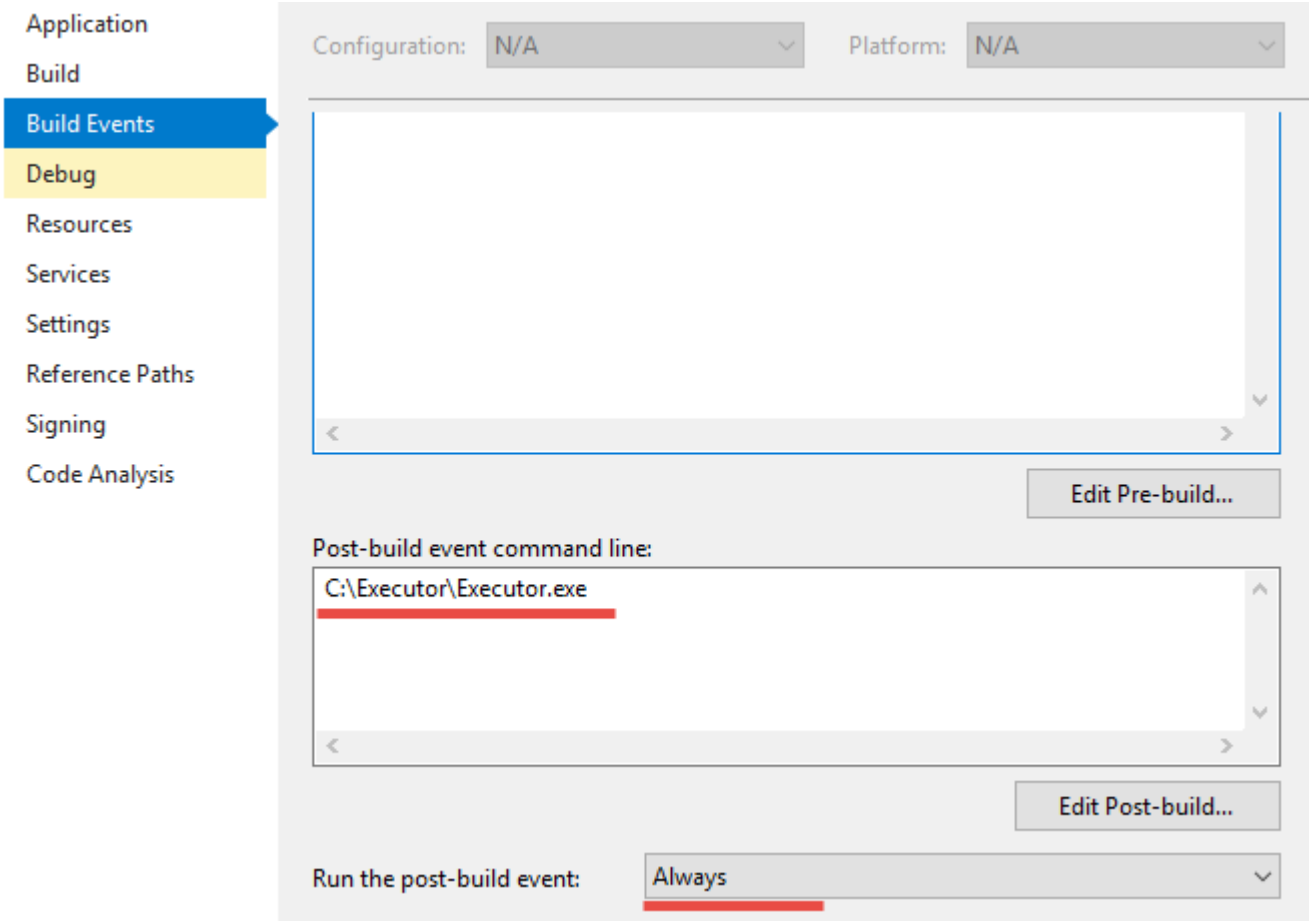
To work with the classes of the server side of bpm'online core, set the dependencies from the necessary bpm'online class libraries in the created project. For example, add the dependency from the Terrasoft.Core.dll library (Fig. 3). More information about adding the dependencies can be found in the "Managing references in a project" Microsoft documentation article.

> NOTE
>
> Class libraries of the bpm'online namespace can be found in the *bin* folder of the application. Class libraries are being copied to the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder when executing the .bat files (see Step 2. Configure the WorkspaceConsole utility" of the example of configuration server code development for on-site application).

In the created class library project, specify the full path to the configured Executor utility in the [Post-build event command line] property on the [BuildEvents] tab of the properties window (Fig. 7), for example *C:\Executor\Executor.exe*. Also, you must select the condition for starting the library build event on this tab.

> **NOTE**
>
> The configuration process is given below on the Step "3. Executor utility configuration".

Fig. 7. [Build Events] tab properties



2. Develop the functions

For this, add the class that will implement the *Terrasoft.Core.IExecutor* interface to the created library project. The implementation of the class is available below:

```
using System;
using System.Web;
using Terrasoft.Core;
using Terrasoft.Core.Entities;

namespace BpmonlineCustomServerLogic.Cloud
{
    public class MyContactReader : IExecutor
    {
        public void Execute(UserConnection userConnection)
        {
            // Getting an instance of the [Contacts] schema.
            var entitySchema =
userConnection.EntitySchemaManager.GetInstanceByName("Contact");
            // Create an instance of the query class.
            var esq = new EntitySchemaQuery(entitySchema);
```

```
            // Adding all the columns of the schema to the query.
            esq.AddAllSchemaColumns();
            // Getting the collection of records in the [Contacts] section.
            var collection = esq.GetEntityCollection(userConnection);
            foreach (var entity in collection)
            {
                // The output in the http-response of the request from the Executor
utility of the necessary values.
                HttpContext.Current.Response.Write(entity.GetTypedColumnValue<string>
("Name"));
                HttpContext.Current.Response.Write(Environment.NewLine);
            }
        }
    }
}
```

3. Executor utility configuration

> **NOTE**
>
> You can use following link to download the utility configured for processing the example.

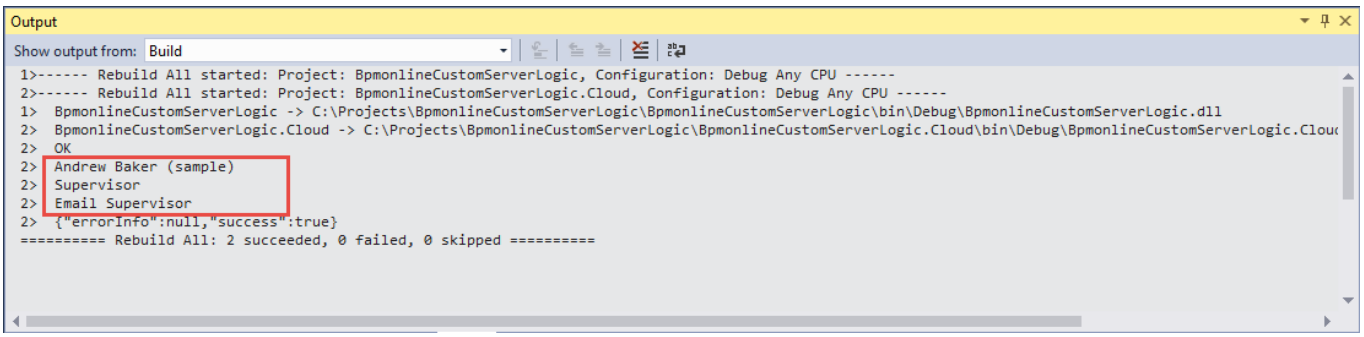
Open the Executor utility folder, for example, *C:\Executor*. Then, specify the values for the following configuration items in the configuration file:

- Loader – URL of the bpm'online application loader. Usually this is the URL of the bpm'online site, for example "https://mycloudapp.bpmonline.com".
- WebApp – URL of the bpm'online application. Usually this is a path to default configuration of bpm'online, for example "https://mycloudapp.bpmonline.com/0".
- Login – the name of the bpm'online user, for example, "Supervisor".
- Password – the password of bpm'online user.
- LibraryOriginalPath – the path to the initial copy of the class library. Usually, this is the path by which a class library is created after compilation in Visual Studio, for example, "C:\Projects\BpmonlineCustomServerLogic\BpmonlineCustomServerLogic.Cloud\bin\Debug\BpmonlineCustomServerLogic.Cloud.dll".
- LibraryCopyPath – the path by which a copy of the class library will be created for work with the remote server. This can be a temporary folder that contains the Executor utility, for example, "C:\Executor\BpmonlineCustomServerLogic.Cloud.dll".
- LibraryType – full name of the class in which the developed program logic is implemented, including the names of all namespaces. For example, "BpmonlineCustomServerLogic.Cloud.MyContactReader".
- LibraryName – name of the class library, for example, "BpmonlineCustomServerLogic.Cloud.dll".

4. Run the developed program code

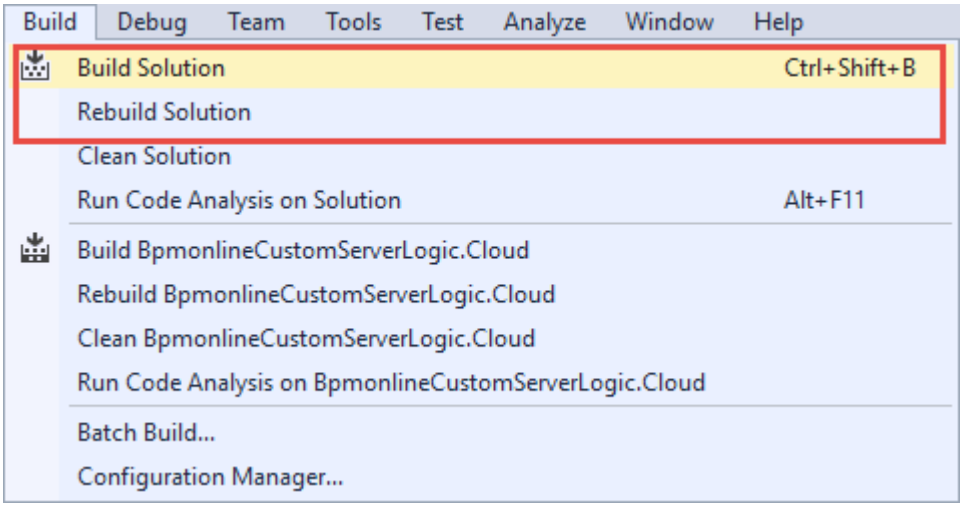
The result of execution the development program code can be observed in the [Output] window of the Visual Studio after successful building of the class library (Fig. 8).

Fig. 8. Result of program code execution

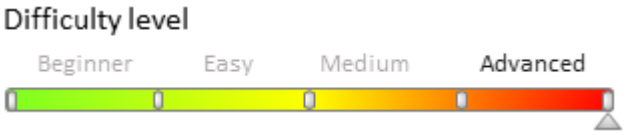


To launch the building process use the [Build Solution] and [Rebuild Solution] menu commands (Fig. 9).

Fig. 9. [Build] menu commands



Automatic displaying of changes in the development of the custom logic



Introduction

When developing configuration server code in the file system, each time after making changes to the source code of the custom schema you need to refresh the browser page on which the application is opened. This reduces the development performance.

To avoid this, we developed the new functionality of automatic browser page reload after changes. This functionality works in a following way.

When the application starts, it creates an object that tracks the changes of the .js file with the source code of the developed module in the file system. If the changes have made, a message is sent to the client bpm'online application. In the client application, a specific object which is signed to this message defines dependent objects of the changed module, destroys them, registers new paths to the modules and tries to load the modified module again. After that, all the pre-initialized modules will be requested by the browser via new paths and load changes from the file system. It does not take time to interpret and load other modules. Separate development page enables to avoid loading of additional modules (for example left or right panel, communication panel, etc.). This reduces the number of requests to the server.

This approach reveals the connectivity of the modules and detects unnecessary dependencies to eliminate them.

Known issues

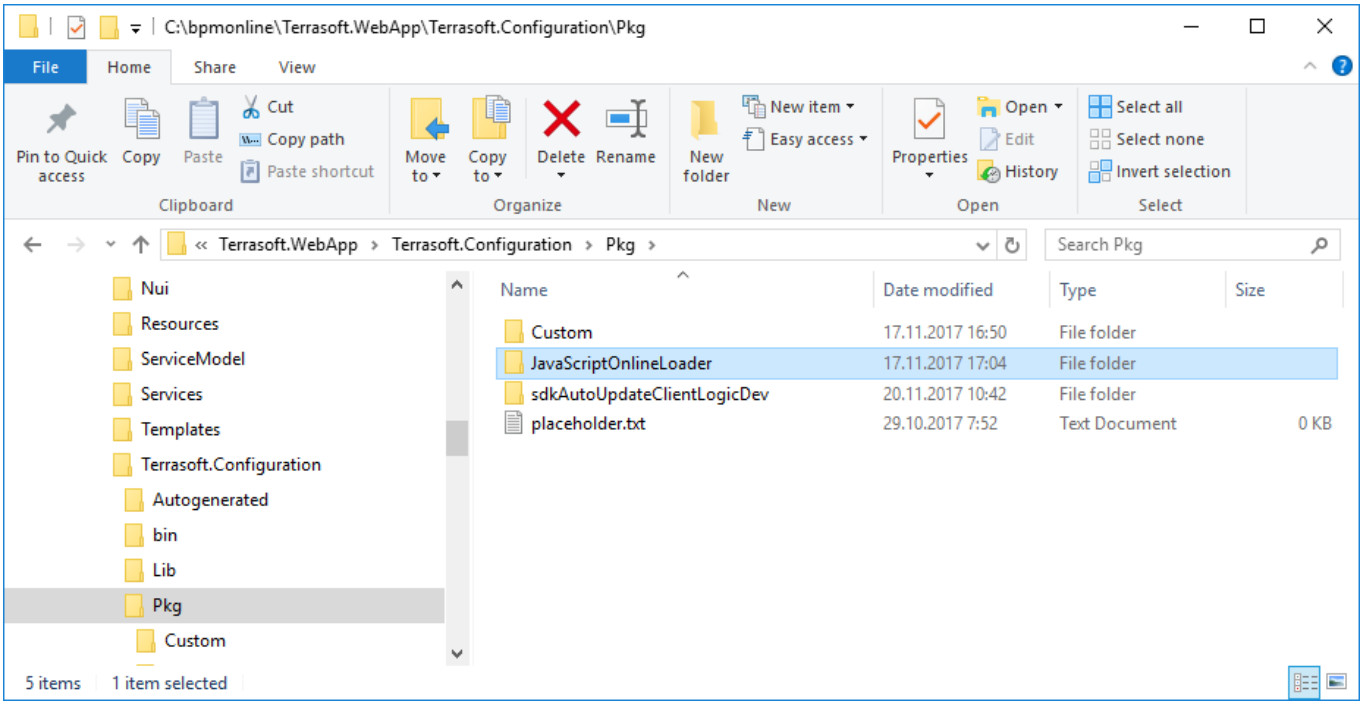
1. If there is a syntax error in the source code of the module, the page will not automatically refresh. The page will need to be forcibly refreshed (for example, by pressing the F5 key). If the error is corrected, the page will return to the operable status.
2. Not all bpm'online modules can be downloaded separately. The main reason is the effect of strong coupling of modules.

Configuration steps

1. Install the JavaScriptOnlineLoader package

Enable the development mode in the file system and add the *JavaScriptOnlineLoader* folder with corresponding package to the *[Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg* folder (Fig. 1).

Fig. 1. The JavaScriptOnlineLoader package in the file system

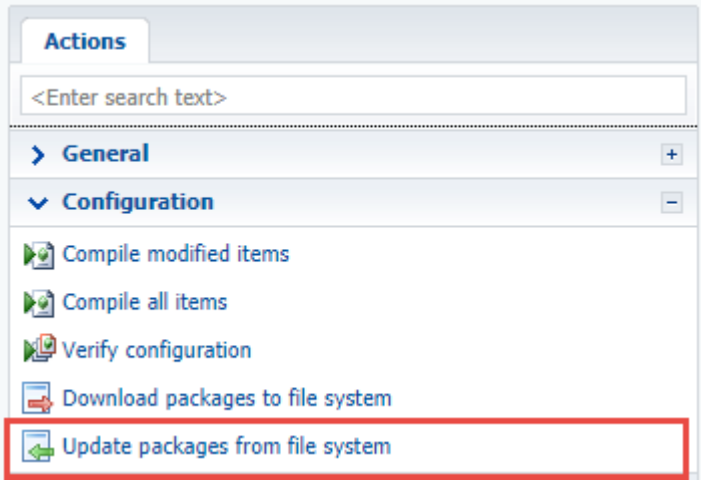


More information about the development mode in the file system can be bound in the "**Development in the file system**" article.

> ⚠ The package is available on the GitHub (https://github.com/vladimir-nikonov/pngstore/tree/master/JavaScriptOnlineLoader). Also the archive with the package can be downloaded by the link.

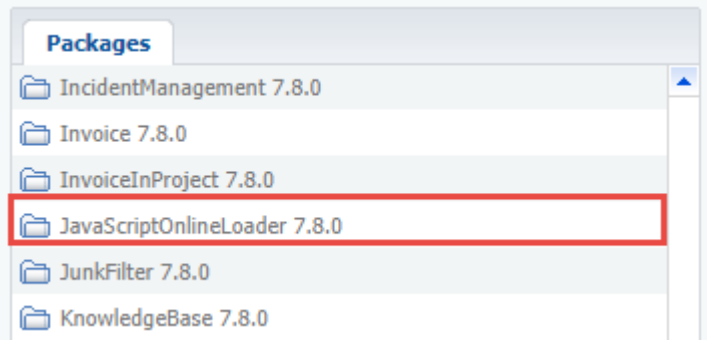
Load the package to the configuration with the [Update packages from file system] action (Fig. 2).

Fig. 2. The [Update packages from file system] action



As a result, the package will be displayed on the [Packages] tab (Fig. 3).

Fig. 3. The package in the [Configuration] section



2. Open the page of the developed module in the browser

To do this, open the *ViewModule.aspx* page with the added parameter with the following format:

```
?vm=DevViewModule#CardModuleV2/<Module name>
```

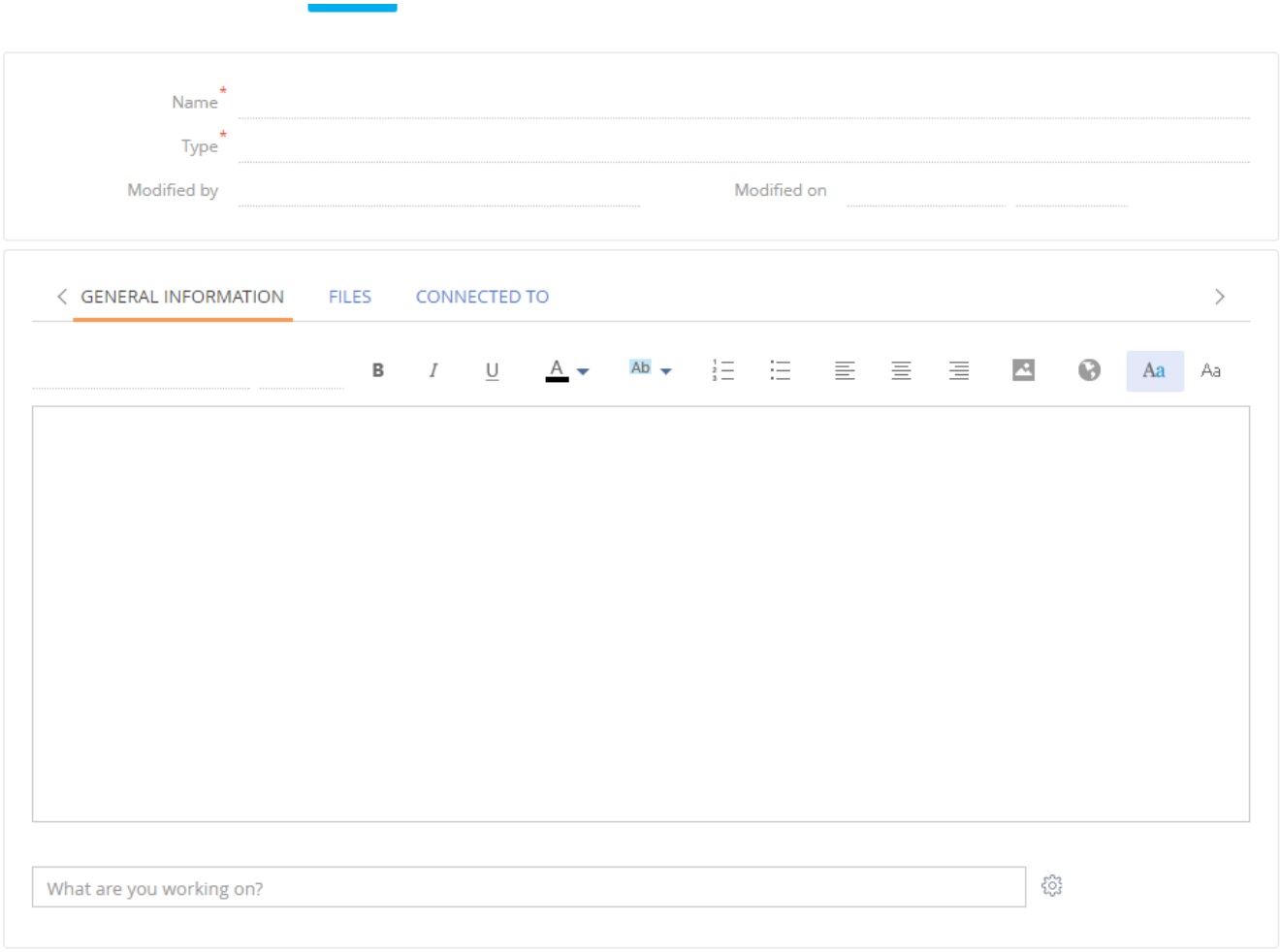
For example, the *KnowledgeBasePageV2* replacing schema (the schema of the [Knowledge base] section edit page) is added to the custom package. The page with the functions of automatic displaying of changes will be available at the following URL:

```
http://localhost/bpmonline/0/Nui/ViewModule.aspx?
vm=DevViewModule#CardModuleV2/KnowledgeBasePageV2
```

The *http://localhost/bpmonline* is a URL of the bpm’online application deployed on-site.

After clicking this URL, the *ViewModule.aspx* page will be displayed with the loaded module (Fig. 4).

Fig. 4. The ViewModule.aspx page with the loaded module



3. Change the source code of the developed schema

The source code of the developed schema can be changed in any text editor (for example, the Notepad). After saving the changes, the page opened in the browser will be automatically refreshed.

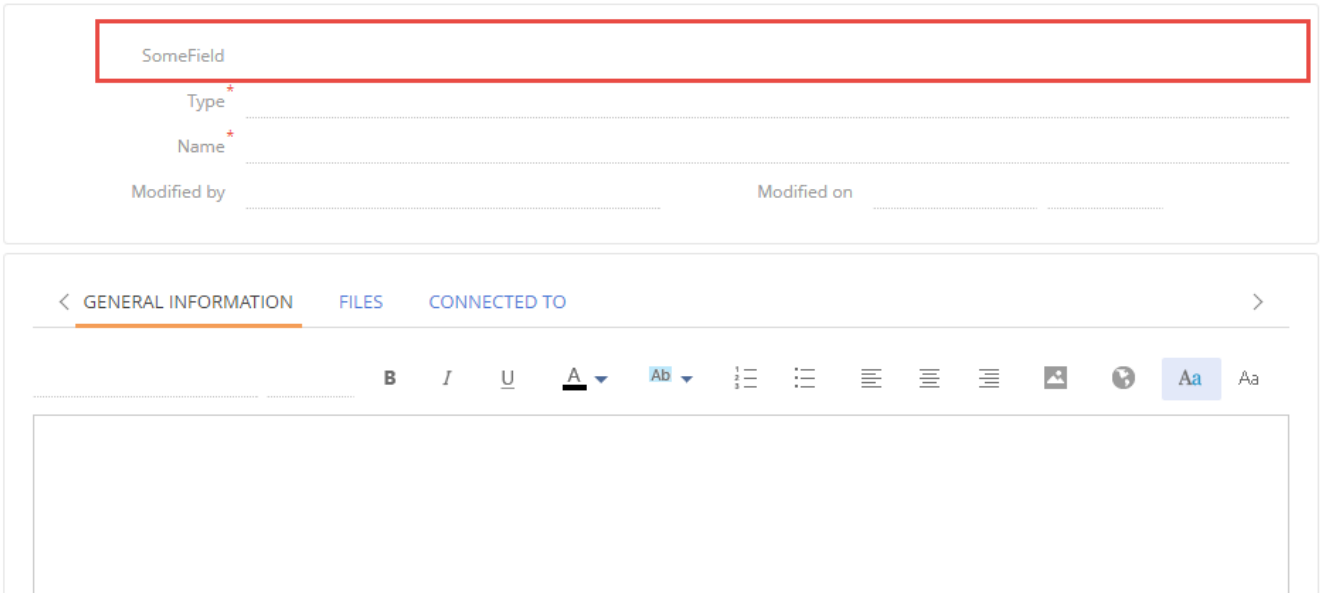
For example, the *KnowledgeBasePageV2* replacing schema (the schema of the [Knowledge base] section edit page) is added to the *sdkAutoUpdateClientLogicDev* custom package. After loading to the file system, the schema code will be available in the ..*Pkg**sdkAutoUpdateClientLogicDev**Schemas**KnowledgeBasePageV2* folder.

If the following source code will be added to the *KnowledgeBasePageV2.js* file and save it, the browser page will be automatically refreshed. The changes will be displayed immediately (Fig. 5).

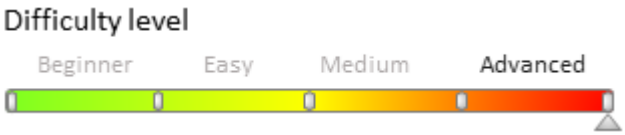
```
define("KnowledgeBasePageV2", [],
    function() {
        return {
            entitySchemaName: "KnowledgeBase",
            diff: /**SCHEMA_DIFF*/[
             {
                    "operation": "insert",
                    "parentName": "Header",
                    "propertyName": "items",
                    "name": "SomeField",
                    "values": {
                        "layout": {"column": 0, "row": 0, "colSpan": 24},
                        "caption": "SomeField"
                    }
            }
            ]/**SCHEMA_DIFF*/
```

```
        };
    });
```

Fig. 5. Page with changes



Packages file content



Introduction

Starting with version 7.11.3 you can add file content (.js, .css files, images, etc.) to the custom packages.

File content of packages is a number of any files used by the application. File content is static and is not processed by the web server (see “**Client static content in the file system**”). This increases application performance.

> ⚠ ATTENTION
>
> File content is an integral part of the bpm’online and is always stored in the *...\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\<Package name>\Files* folder.

> ⚠ NOTE
>
> Any files can be added to the package, but only the files needed for the client part of bpm'online will be used.

> ⚠ ATTENTION
>
> You need to generate auxiliary files (see “Generation of auxiliary files” below) to use file content.

Recommended file storage structure

To use file content the *Files* folder was added to the package structure (see “**Package structure and contents**”). It is recommended to keep following structure of the *Files* folder:

```
-PackageName
```

```
    ...
    -Files
        -src
            -js
                bootstrap.js
                [other *.js files]
            -css
                [*.css files]
            -less
                [*.less files]
            -img
                [image files]
            -res
                [resource files]
        descriptor.json
    ...
descriptor.json
```

Here

js – folder with .js files of JavaScript source codes

css – folder with *.css style files

less – folder with *.less style files

img – folder with images

res – folder with resource files

descriptor.json – descriptor of the file content.

How to add a new file content to the package

Copy a file to the corresponding subfolder of the *Files* folder of specific package. The *Files* folder will be available by the *...\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\<Package name>\Files* path.

Descriptor of the file content

Information about bootstrap files of the package is stored in the descriptor.json file of the Files folder. The has following structure:

```
{
    "bootstraps": [
        ... // An array of strings containing relative paths to bootstrap files.
    ]
}
```

Example of descriptor.json:

```
{
    "bootstraps": [
        "src/js/bootstrap.js",
        "src/js/anotherBootstrap.js"
    ]
}
```

Bootstrap files of the package

The .js files that enable to manage loading of client configuration logic. The file does not have a clear structure.

```
(function() {
    require.config({
        paths: {
            "Module name ":" A link to the file content",
            ...
```

```
        }
    });
})();
```

Example of bootstrap.js:

```
(function() {
    require.config({
        paths: {
            "MyPackage1-ContactSectionV2": Terrasoft.getFileContentUrl("MyPackage1",
"src/js/ContactSectionV2.js"),
            "MyPackage1-Utilities": Terrasoft.getFileContentUrl("MyPackage1",
"src/js/Utilities.js")
        }
    });
})();
```

> ATTENTION
>
> All bootstrap files are loaded asynchronously after the core is loaded, but before loading the configuration.

Loading of the bootstrap files

For correct loading of bootstrap files, the _FileContentBootstraps.js auxiliary file is generated in the static content folder (see "Generation of auxiliary files" below). This file contains information about bootstrap files of all packages.

Example of the _FileContentBootstraps.js:

```
var Terrasoft = Terrasoft || {};
Terrasoft.configuration = Terrasoft.configuration || {};
Terrasoft.configuration.FileContentBootstraps = {
    "MyPackage1": [
        "src/js/bootstrap.js"
    ]
};
```

File content versioning

For correct versioning of the file content, the _FileContentDescriptors.js auxiliary file is generated in the static content folder (see "Generation of auxiliary files" below). This file contains information about the files in the file content of all packages in the "key-value" collection view. Each key (file name) corresponds to a unique hash code. This guarantees downloading of the up to date version of the file to the browser.

> NOTE
>
> After installing the file content, there is no need to clear the browser cache.

Example of the _FileContentDescriptors.js file:

```
var Terrasoft = Terrasoft || {};
Terrasoft.configuration = Terrasoft.configuration || {};
Terrasoft.configuration.FileContentDescriptors = {
    "MyPackage1/descriptor.json": {
        "Hash": "5d4e779e7ff24396a132a0e39cca25cc"
    },
    "MyPackage1/Files/src/js/Utilities.js": {
        "Hash": "6d5e776e7ff24596a135a0e39cc525gc"
    }
};
```

Generation of auxiliary files

Execute the BuildConfiguration operation in the WorkspaceConsole:

```
Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -
workspaceName=Default -destinationPath=Terrasoft.WebApp\ -
configurationPath=Terrasoft.WebApp\Terrasoft.Configuration\ -
useStaticFileContent=false -usePackageFileContent=true -autoExit=true
```

In this code:

- *operation* – operation name. *BuildConfiguration* – operation of configuration compilation.
- *useStaticFileContent* – a flag of using static content. Should be *false*.
- *usePackageFileContent* – a flag of using file content of the packages. Should be *true*.

Other WorkspaceConsole parameters are described in the "**WorkspaceConsole parameters**” article.

As a result the _FileContentBootstraps.js and _FileContentDescriptors.js auxiliary files will be generated in the folder with the static content ...*Terrasoft.WebApp\conf\content*.

ATTENTION

Also the generation of auxiliary files is performed at installation of packages from SVN and executing compilation action in the [Configuration] section.

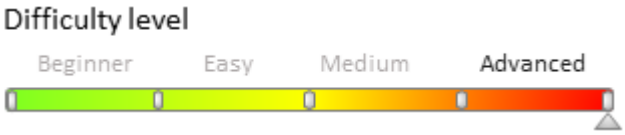
Transition of modifications between environments

File content is an integral part of the package. The content is stored in the SVN store with all package content. The content can be transferred to another development environment via SVN (see “**Working with SVN in the file system**”).

ATTENTION

It is recommended to use bpm’online built-in tools to transfer on test and production environments (see”**Exporting packages from the application interface**” and “**Installing marketplace applications from a zip archive**”).

Localization of the file content



Introduction

Starting with version 7.11.3 you can add file content (.js, .css files, images, etc.) to the custom packages.

File content of packages is a number of any files used by the application. File content is static and is not processed by the web server (see “**Client static content in the file system**”). This increases application performance.

More information about file content can be found in the “**Packages file content**”.

Localization with configuration resources

To translate the resources it is recommended to use separate module with localizable resources created via internal bpm’online tools in the **[Configuration] section**. The complete source code of this module is available below:

```
define("Module1", ["Module1Resources"], function(res) {
  return res;
});
```

To include localizable resources to the module that is defined in the file content of the package you need to define the module with resources. Example:

```
define("MyPackage-MyModule", ["Module1"], function(module1) {
  console.log(module1.localizableStrings.MyString);
});
```

Localization via i18n plugin

i18n is a plugin for AMD loader (for example, RequireJS) used for loading lcoalizable string resources. The source code of the plugin can be found in the https://github.com/requirejs/i18n storage. Documentation is available by the http://requirejs.org/docs/api.html#i18n link.

To localize file content with RequireJS i18n plugin, perform the following steps:

1. Add the plugin to the folder with the source code .js files:
..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\MyPackage1\content\js\i18n.js.

MyPackage1 – working folder of the MyPackage1 package (see "**Packages file content**").

2. Create the *..\MyPackage1\content\nls* folder and put there one or several .js files with localizable resources. File names can be arbitrary. File content – AMD modules with objects of the following structure:

- The "root" field contains the key-value collection where the "key" is the name of a localizable string and the "value" is localizable string of the default language. The value will be used if the requested language is not supported.
- Fields with the names of standard cultures (for example, "en-US", "de-DE") and the boolean value. The value is *true* if the supported culture is enabled and *false* if it is disabled.

For example the added *..\MyPackage1\content\js\nls\ContactSectionV2Resources.js* file with the following content:

```
define({
    "root": {
        "FileContentActionDescr": "File content first action (Default)",
        "FileContentActionDescr2": "File content second action (Default)"
    },
    "en-US": true,
    "ru-RU": true
});
```

3. In the *..\MyPackage1\content\nls* folder, create folders with the names corresponding to the cultures of the localization files that will be put in these folders (for example, "en-US", "de-DE"). For example, if the German and English culture are supported the folder structure will be following:

```
content
    nls
        en-US
        ru-RU
```

4. In each created localization directory put the same number of .js files as in the *..\MyPackage1\content\nls* root folder. File content is the AMD modules with objects of the key-value collections, where the "key" is the name of a localizable string and the "value" is a string of the language corresponding to the name of the folder (the code of the culture).

For example, if the German and English culture are supported you need to create two *ContactSectionV2Resources.js* files. The content of the ..\MyPackage1\content\js\nls\en-US\ContactSectionV2Resources.js, file corresponding to English culture:

```
define({
    "FileContentActionDescr": "File content first action",
    "FileContentActionDescr2": "File content second action"
});
```

The content of the ..\MyPackage1\content\js\nls\de-DE\ContactSectionV2Resources.js, file corresponding to German culture:

```
define({
    "FileContentActionDescr": "Die erste Aktion des Dateiinhalts"
});
```

> ATTENTION
>
> As the translation of the "FileContentActionDescr2" string is not specified for the German culture the default value ("File content second action (Default)") will be used.

5. Edit the bootstrap.js file.

- Connect the i18n plugin by specifying its name as the "i18n" alias in the RequireJS path configuration and specifying corresponding path to the plugin in the *paths* propertiy.
- For the plugin specify a culture that is current for the user. Set the object with the *i18n* property to the *config* property of the configuration object of the RequireJS library. Set the object with the *locale* property and the value received from the *Terrasoft.currentUserCultureName* (the code of the current culture) to the the object with the *i18n* property.
- For each file with localization resources set corresponding aliases and paths in the RequireJS path configuration. The alias must be a URL-path relative to the *nls* directory.

Example of the *..\MyPackage1\content\js\bootstrap.js* file content:

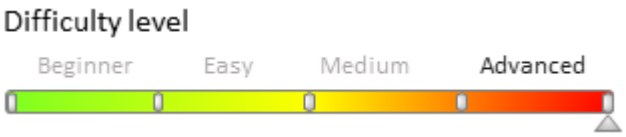
```
(function() {
    require.config({
        paths: {
            "MyPackage1-Utilities": Terrasoft.getFileContentUrl("MyPackage1",
"content/js/Utilities.js"),
            "MyPackage1-ContactSectionV2": Terrasoft.getFileContentUrl("MyPackage1",
"content/js/ContactSectionV2.js"),
            "MyPackage1-CSS": Terrasoft.getFileContentUrl("MyPackage1",
"content/css/MyPackage.css"),
            "MyPackage1-LESS": Terrasoft.getFileContentUrl("MyPackage1",
"content/less/MyPackage.less"),
            "i18n": Terrasoft.getFileContentUrl("MyPackage1", "content/js/i18n.js"),
            "nls/ContactSectionV2Resources":
Terrasoft.getFileContentUrl("MyPackage1",
"content/js/nls/ContactSectionV2Resources.js"),
            "nls/ru-RU/ContactSectionV2Resources":
Terrasoft.getFileContentUrl("MyPackage1", "content/js/nls/ru-
RU/ContactSectionV2Resources.js"),
            "nls/en-US/ContactSectionV2Resources":
Terrasoft.getFileContentUrl("MyPackage1", "content/js/nls/en-
US/ContactSectionV2Resources.js")
        },
        config: {
            i18n: {
                locale: Terrasoft.currentUserCultureName
            }
        }
    });
})();
```

6. Use the resources by specifying the corresponding module of resources with the "i18n!" alias in the dependency array. For example, to use the *FileContentActionDescr* (see steps 2,4) string as a title for the new action in the [Contacts] section, add the following content to the *..\MyPackage1\content\js\ContactSectionV2.js* file:

```
define("MyPackage1-ContactSectionV2", ["i18n!nls/ContactSectionV2Resources",
    "css!MyPackage1-CSS", "less!MyPackage1-LESS"], function(resources) {
    return {
        methods: {
            getSectionActions: function() {
                var actionMenuItems = this.callParent(arguments);
```

```
                actionMenuItems.addItem(this.getButtonMenuItem({"Type":
"Terrasoft.MenuSeparator"}));
                actionMenuItems.addItem(this.getButtonMenuItem({
                    "Click": {"bindTo": "onFileContentActionClick"},
                    "Caption": resources.FileContentActionDescr
                }));
                return actionMenuItems;
            },
            onFileContentActionClick: function() {
                console.log("File content clicked!")
            }
        },
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    }
});
```

How to create Unit-tests via NUnit and Visual Studio



Introduction

Unit-testing (module testing) is a software development process for verifying the operation capacity of the isolated program components (see “Module testing”). The tests are usually written by developers for every advanced method of the developed class. This allows to quickly reveal the source code recession – errors in the tested program components.

One of the NET-application Unit-testing frameworks is NUnit – Unit-testing environment with an open source code. A special adapter has been developed to integrate it with Visual Studio. Such adapter can be installed as a Visual Studio extension or as a project NuGet package with the implemented Unit-tests. Use this link to access the 3.x version framework documentation.

To create Unit-tests for methods or bpm'online custom package class properties:

1. Install NUnit Visual Studio adapter
2. Switch to the **file system development** mode
3. Set up the Unit-test project
4. Create the tests
5. Perform testing

Case description

Add tests for the custom class, implemented in the [Source code] type *UsrNUnitSourceCode* schema of the bpm'online application *sdkNUnit* custom package.

Source code

You can access the custom class implementation package at sdkNUnit repository at Github.

Case implementation algorithm

1. Install NUnit Visual Studio adapter

You can install NUnit Visual Studio adapter either as a Visual Studio extension or as a NuGet package.

Installing NUnit adapter as a Visual Studio extension

The advantage of installing NUnit adapter as a Visual Studio extension is its availability for any test project since the adapter becomes part of IDE. Another advantage is the automatic extension update. The disadvantage is the necessity to install it for every test project team member.

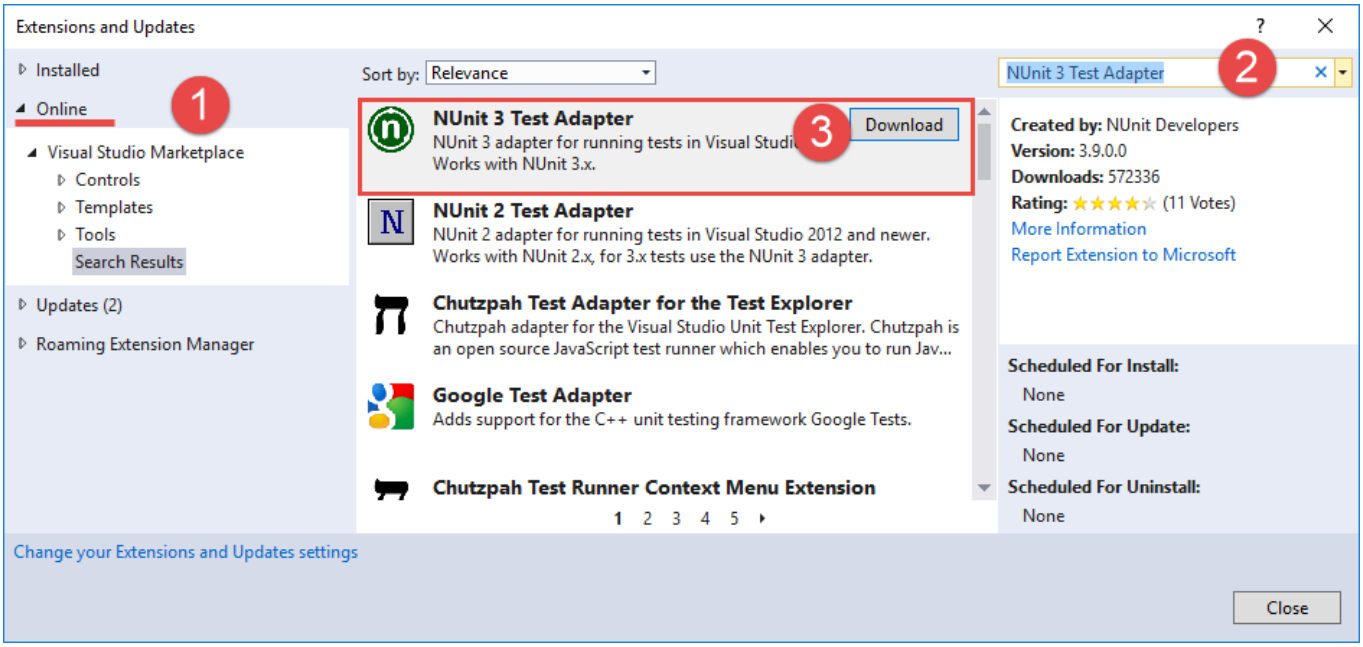
To install NUnit adapter:

1. Download extension from Visual Studio Marketplace *.VSIX-file.

2. Double-click the *.VSIX-file and run the installation. Select the needed Visual Studio versions during installation.

> **NOTE**
>
> As an alternative, you can install NUnit adapter via the Tools > Extensions and Updates menu. Select [Online] filter (Fig. 1. 1) and indicate "NUnit 3 Test Adapter" (2) in the search string. Select NUnit 3 Test Adapter extension in the search results and click [Download]. The extension installation starts automatically.

Fig. 1. Extension search by Visual Studio built-in tools



Installing NUnit adapter as a NuGet package

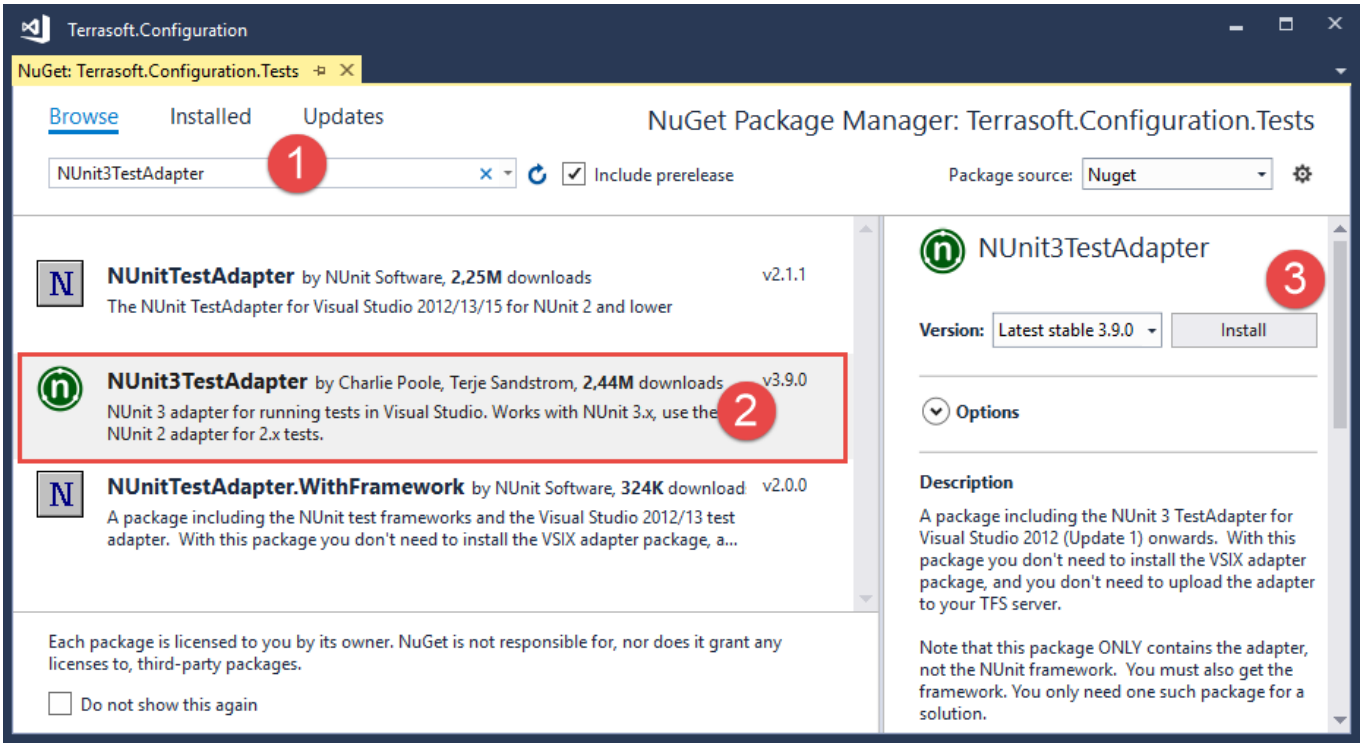
The advantage of NUnit adapter installation as a NuGet-package is that in this case it becomes part of Visual Studio project and is available for access to all developers who use the project. The disadvantage is the necessity to install it for all Unit-test projects.

To install NUnit adapter:

1. Right-click the test project (for instance, *Terrasoft.Configuration.Tests.csproj*) and select the [Manage NuGet Packages...] command.

2. Indicate "NUnit3TestAdapter" (1) in the search string of the opened NuGet package manager tab (Fig.2). Select the package in the search results (2) and install it (3).

Fig. 2. Installing NUnit3TestAdapter package in the NuGet package manager



> **NOTE**
>
> You can find detailed description of NuGet-package installation into Visual Studio projects in the “Package Manager UI” Microsoft article.

2. Switch to the file system development mode

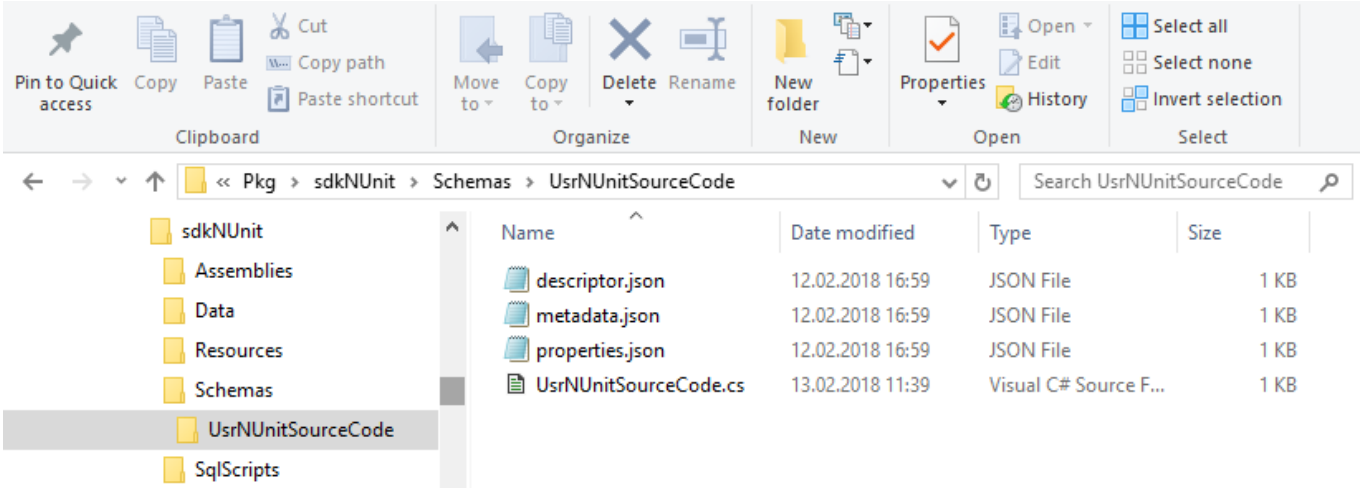
Creating Unit-tests for .NET classes, implemented in bpm’online packages is only possible in the file system development mode. You can find more information about the bpm’online configuration development in the file system, setting up Visual Studio and the server code operation case in “**Development in the file system**”, “**Visual Studio settings for development in the file system**” and “**Working with the server side source code in Visual Studio**”.

The *sdkNUnit* custom package containing [Source code] type *UsrNUnitSourceCode* schema is used in this case. The *UsrNUnitSourceCode* C# class containing methods that require writing tests is implemented in this schema source code.

> **ATTENTION**
>
> You can access the custom class implementation package at sdkNUnit repository at Github.

The *sdkNUnit* custom package has the following view (see Fig.3) after it has been uploaded to the file system:

Fig. 3. The *sdkNUnit* package structure



Class source code for testing:

```
namespace Terrasoft.Configuration
{
    public class UsrNUnitSourceCode
    {
        // String property.
        public string StringToTest
        {
            get
            {
                return "String to test";
            }
        }
        // The method that verifies the equality of the two strings.
        public bool AreStringsEqual(string str1, string str2)
        {
            return str1 == str2;
        }
    }
}
```

3. Set up the Unit-test project

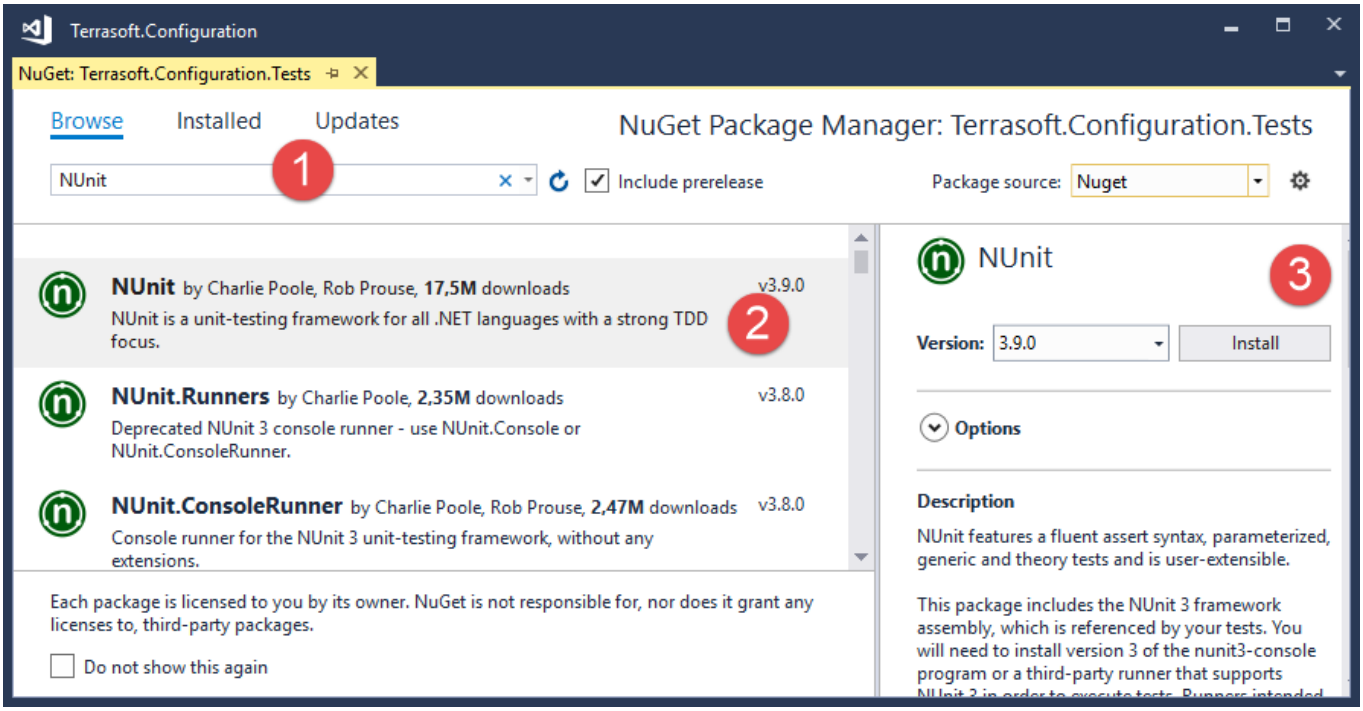
The *Terrasoft.Configuration.Tests.csproj* pre-configured project is used for creating Unit-tests in this case. It is delivered with the *Terrasoft.Configuration.sln* solution (see "Server code operation in Visual Studio").

Add the *NUnit* NuGet-package in the project dependency to use *NUnit* framework for creating tests in the *Terrasoft.Configuration.Tests.csproj* project. To do this:

1. Right-click the *Terrasoft.Configuration.Tests* test project in Solution Explorer and select the [Manage NuGet Packages...] command.

2. Indicate "NUnit" (1) in the search string of the opened NuGet package manager tab (Fig.4), select the package in the search results (2) and install it (3).

Fig. 4. Installing NUnit package in the NuGet package manager



4. Create the tests

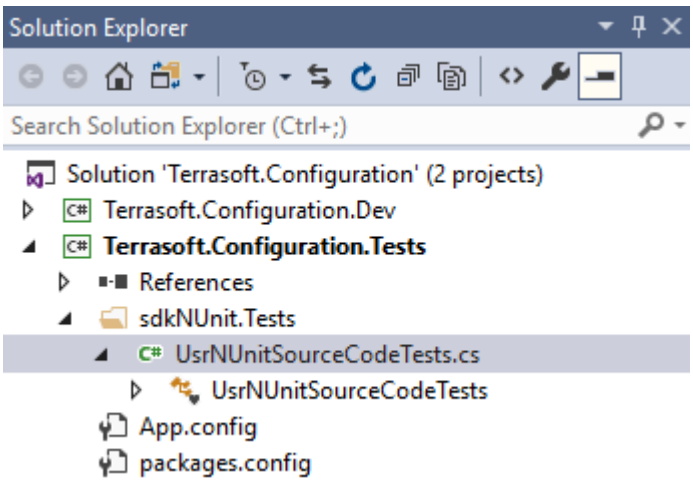
> NOTE
>
> It is common practice that the test-containing class name must have the tested class name with "Tests" word in it. It is also convenient to place tests in catalogs to group them in a project. The catalog name should match the tested package name and have ".Tests" ending in it.

To create tests for *UsrNUnitSourceCode* class:

1. Create *sdkNUnit.Tests* catalog in the *Terrasoft.Configuration.Tests.csproj* project.

2. Create the new *UsrNUnitSourceCodeTests*.class in the *sdkNUnit.Tests* catalog. This class source code will be stored in the *UsrNUnitSourceCodeTests.cs* file (Fig.5).

Fig. 5. Test project structure



3. Add the implementation test methods to the *UsrNUnitSourceCodeTests* class:

```
using NUnit.Framework;

namespace Terrasoft.Configuration.Tests.sdkNUnitTests
```

```
{
    [TestFixture]
    class UsrNUnitSourceCodeTests
    {
        // The tested class instance.
        UsrNUnitSourceCode objToTest = new UsrNUnitSourceCode();
        // Testing string.
        string str = "String to test";

        [Test]
        public void ClassReturnsCorrectStringProperty()
        {
            // Testing the string property value.
            // The value must be populated and match the required value.
            string res = objToTest.StringToTest;
            Assert.That(res, Is.Not.Null.And.EqualTo(str));
        }

        [Test]
        public void StringsMustBeEqual()
        {
            // Testing the value equality of the two strings.
            bool res = objToTest.AreStringsEqual(str, "String to test");
            Assert.That(res, Is.True);
        }

        [Test]
        public void StringsMustBeNotEqual()
        {
            // Testing the value inequality of the two strings.
            // This test will fail since the values are equal.
            bool res = objToTest.AreStringsEqual(str, "String to test");
            Assert.That(res, Is.False);
        }
    }
}
```

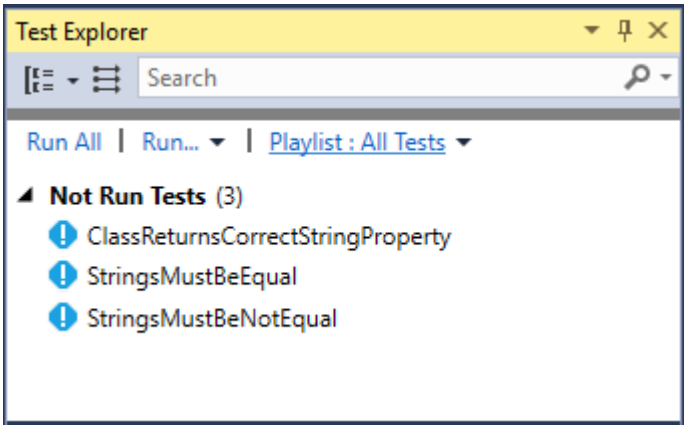
The *UsrNUnitSourceCodeTests* class is decorated by the [TestFixture] attribute, which marks it as a test-containing class. Every method testing a specific functionality must be decorated by the [Test] attribute. You can find the description of the NUnit framework attributes in the "Attributes" NUnit article.

The testing is performed via the Assert.That() method that accepts the tested value and such value limiting objects as arguments. You can find more information about the assertions, Assert.That() method and limiting model in the "Assertions" and "Constraint Model" NUnit articles.

5. Testing

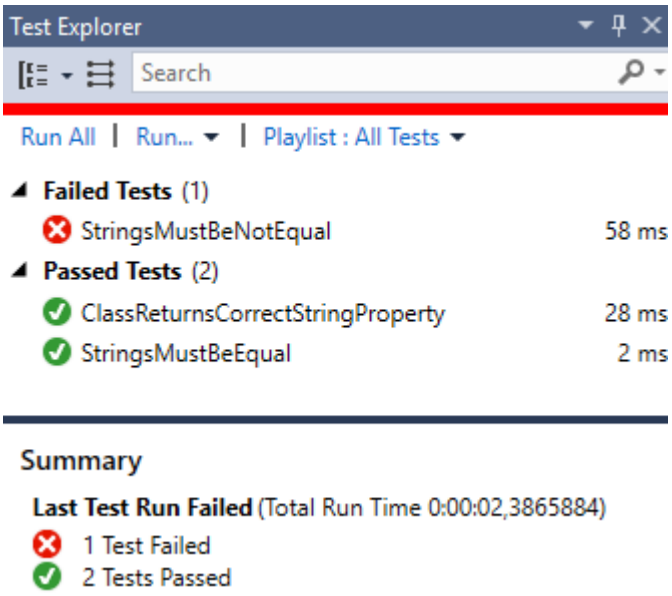
To perform testing, execute the [Test] > [Windows] > [Test Explorer] menu command to open the [Test Explorer] window in Visual Studio (Fig.6).

Fig. 6. [Test Explorer] window



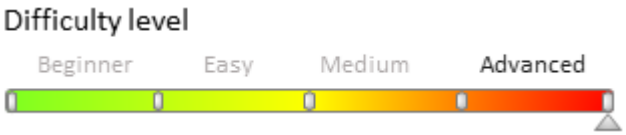
Execute the [Run All] command to run the tests. The successfully passed tests will be moved to the [Passed Test] group, the failed tests will be moved to the [Failed Test] group (Fig.7).

Fig. 7. Passed and Failed tests



You can find more information about the [Test Explorer] window functionality in the "Run unit tests with Test Explorer" Visual Studio article.

How to use TypeScript when developing custom functions



Introduction

Starting with version 7.11.3 you can add file content (.js, .css files, images, etc.) to the custom packages.

File content of packages is a number of any files used by the application. File content is static and is not processed by the web server (see "**Client static content in the file system**"). This increases application performance.

More information about file content can be found in the "**Packages file content**".

File content enables to use languages which can be compiled to JavaScript (for example TypeScript) in custom functions development. More information about TypeScript can be found at https://www.typescriptlang.org.

TypeScript installation

One way to install the TypeScript tools is to use the NPM package manager for the Node.js. For this, run the following command in the Windows console:

```
npm install -g typescript
```

> ATTENTION
>
> Check the Node.js execution environment in your system, before installing TypeScript via the NMP. Download the installer by the https://nodejs.org link.

Case description

When saving an account record, display the message about the correctness of filling the [Also known as] field for the user. The field should contain only letters. Implement the validation logics in the TypeScript language.

Source code

You can download the package with case implementation using the following link.

Case implementation algorithm

1. Switch to the file system development mode

For more information about entering the file system development mode, see the "**Development in the file system**" article.

2. Create the structure of the file content storage

Recommended structure of the file content storage is described in the "**Packages file content**" article. For this:

1. Create the *Files* folder in the custom package loaded to the file system.

2. Add the *src* folder with the *js* subfolder to the *Files* folder.

3. Add the *descriptor.json* file with following content to the *Files* folder:

```
{
    "bootstraps": [
        "src/js/bootstrap.js"
    ]
}
```

4. Add the *bootstrap.js* file with the following content to the *Files\src\js* folder:

```
(function() {
    require.config({
        paths: {
            "LettersOnlyValidator": Terrasoft.getFileContentUrl("sdkTypeScript",
"src/js/LettersOnlyValidator.js")
        }
    });
})();
```

> NOTE
>
> The LettersOnlyValidator.js file specified in the bootstrap.js will be compiled at the step 4.

3. Implement the validation class in the TypeScript language

Create the *Validation.ts* file in the *Files\src\js* folder and declare the *StringValidator* interface in this file:

```
interface StringValidator {
    isAcceptable(s: string): boolean;
}
export = StringValidator;
```

Create the *LettersOnlyValidator.ts* file in this folder. Declare the *LettersOnlyValidator* class in this file. The class will implement the *StringValidator* interface:

```
// Import the module in which the StringValidator interface is implemented.
import StringValidator = require("Validation");

// The created class must belong to the Terrasoft (module) namespace.
module Terrasoft {
    // Declaring the class of value validation.
    export class LettersOnlyValidator implements StringValidator {
        // A regular expression that allows the use of only letter characters.
        lettersRegexp: any = /^[A-Za-z]+$/;
        // Validating method.
        isAcceptable(s: string) {
            return !Ext.isEmpty(s) && this.lettersRegexp.test(s);
        }
    }
}
// Creating and exporting an instance of a class for require.
export = new Terrasoft.LettersOnlyValidator();
```

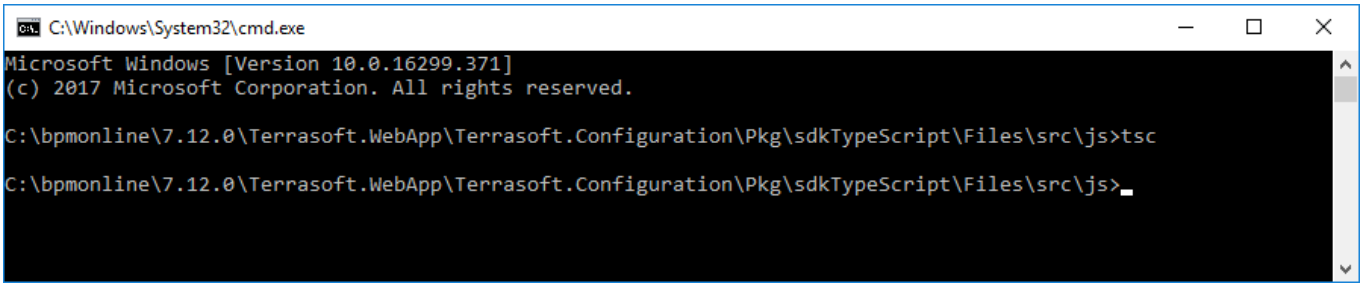
4. Compile the TypeScript source codes to the JavaScript source codes.

Add the *tsconfig.json* configuration file to the *Files\src\js* folder to set up the compilation:

```
{
    "compilerOptions":
    {
        "target": "es5",
        "module": "amd",
        "sourceMap": true
    }
}
```

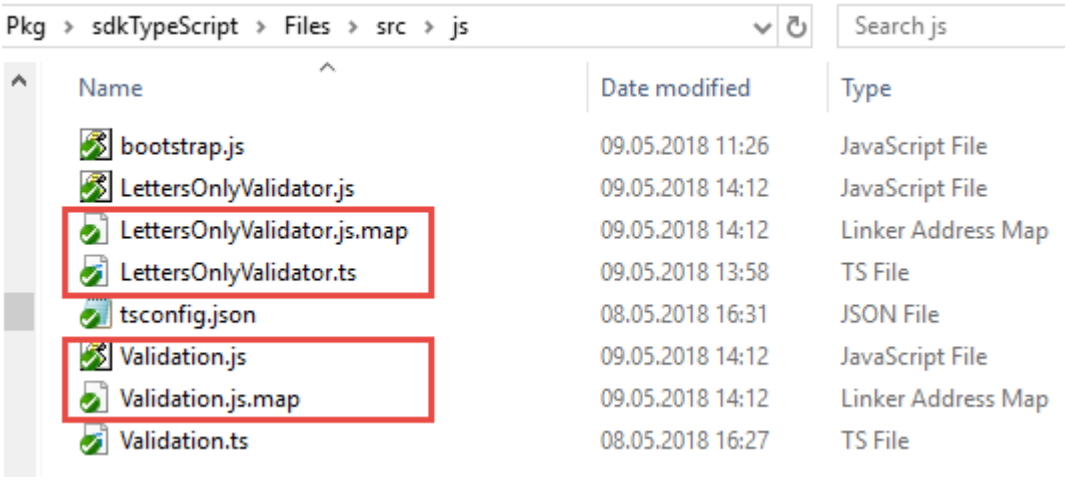
Go to the *Files\src\js* folder via the Windows console and execute the **tsc** command (Fig. 1).

Fig. 1. Execution of the tsc command



As a result of compilation the JavaScript version of the *Validation.ts* and *LettersOnlyValidator.ts* files and the .map files facilitating debugging in the browser will be created in the *Files\src\js* folder (Fig. 2).

Fig. 2. Result of the tsc command execution



The content of the *LettersOnlyValidator.js* file that will be used in the bpm'online (automatically generated):

```
define(["require", "exports"], function (require, exports) {
    "use strict";
    var Terrasoft;
    (function (Terrasoft) {
        var LettersOnlyValidator = /** @class */ (function () {
            function LettersOnlyValidator() {
                this.lettersRegexp = /^[A-Za-z]+$/;
            }
            LettersOnlyValidator.prototype.isAcceptable = function (s) {
                return !Ext.isEmpty(s) && this.lettersRegexp.test(s);
            };
            return LettersOnlyValidator;
        }());
        Terrasoft.LettersOnlyValidator = LettersOnlyValidator;
    })(Terrasoft || (Terrasoft = {}));
    return new Terrasoft.LettersOnlyValidator();
});
//# sourceMappingURL=LettersOnlyValidator.js.map
```

5. Perform the generation of auxiliary files

To generate the *_FileContentBootstraps.js* and *FileContentDescriptors.js* auxiliary files (see "**Packages file content**"):

1. Enter the [Configuration] section.

2. Load the package to the configuration with the [Update packages from file system] action.

3. Click the [Compile all items].

> ⚠ NOTE
>
> Perform this step to apply changes in the *bootsrtap.js* file. You can also use the WorkspaceConsole utility ("**Packages file content**").

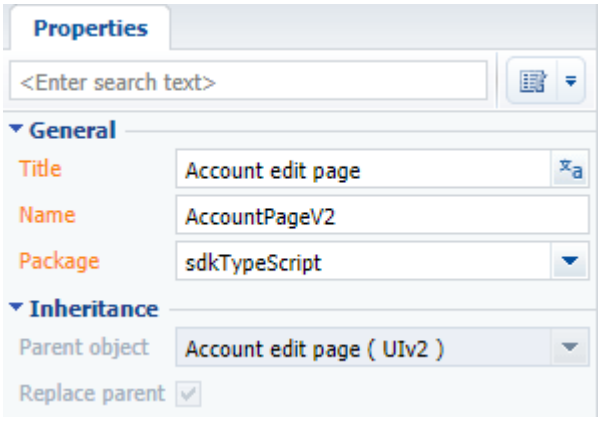
6. Use validator in the bpm'online schema

In the [Configuration] section:

1. Load the package to the configuration with the [Update packages from file system] action.

2. **Create replacing schema** of the edit page of the account record (Fig. 3).

Fig. 3. Properties of the replacing schema



3. Export the package to the file system using the [Download packages to file system] action.

4. Modify the ..\sdkTypeScript\Schemas\AccountPageV2\AccountPageV2.js file in the following way:

```
// Declaration of the module and its dependencies.
define("AccountPageV2", ["LettersOnlyValidator"], function(LettersOnlyValidator) {
    return {
        entitySchemaName: "Account",
        methods: {
            // Validation method.
            validateMethod: function() {
                // Determining the correctness of filling the AlternativeName column.
                var res =
LettersOnlyValidator.isAcceptable(this.get("AlternativeName"));
                // Output of the result to the user.
                Terrasoft.showInformation("Is 'Also known as' field valid: " + res);
            },
            // Overriding the method of the parent schema that is called when the
record is saved.
            save: function() {
                // Calling the validation method.
                this.validateMethod();
                // Calling the basic functions.
                this.callParent(arguments);
            }
        },
        diff: /**SCHEMA_DIFF*/ [] /**SCHEMA_DIFF*/
    };
});
```

When the file with the schema source code is saved and the system web-page is updated, the warning message will be displayed on the account edit page when the page is saved (Fig.4, Fig. 5).

Fig. 4. Incorrectly populated field

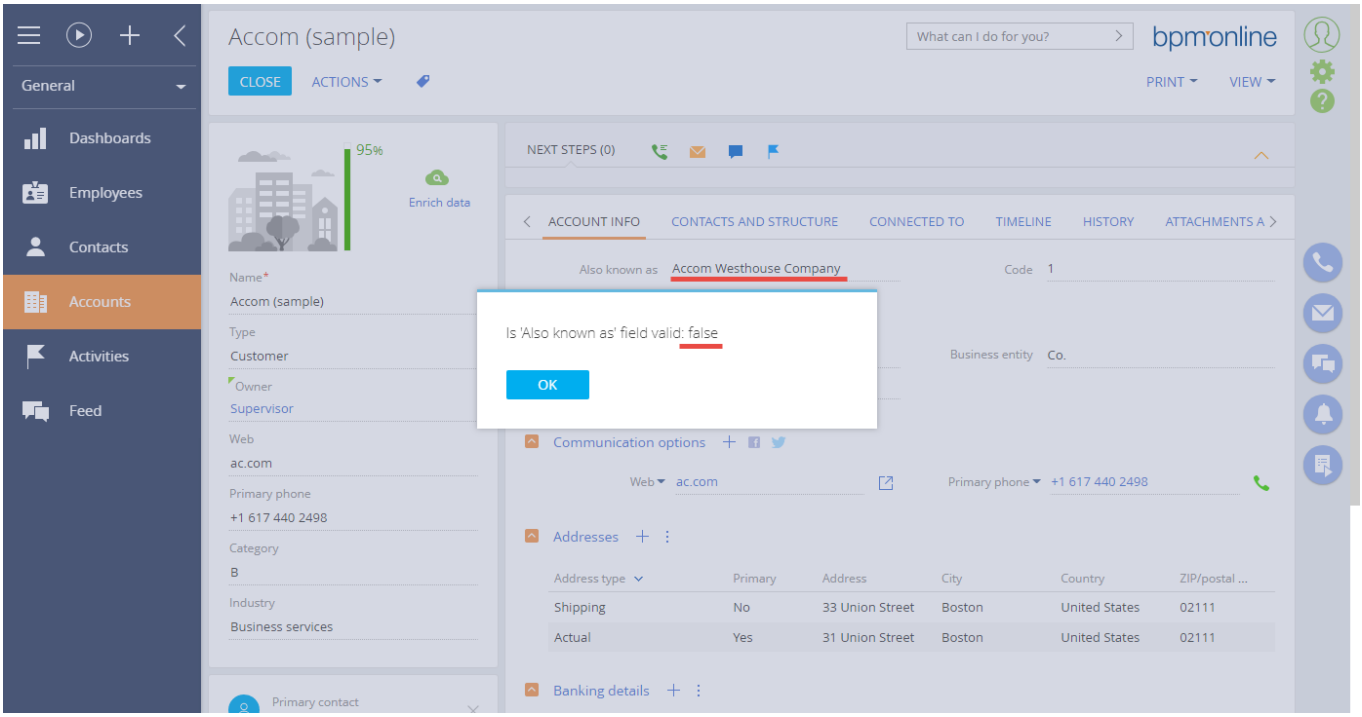
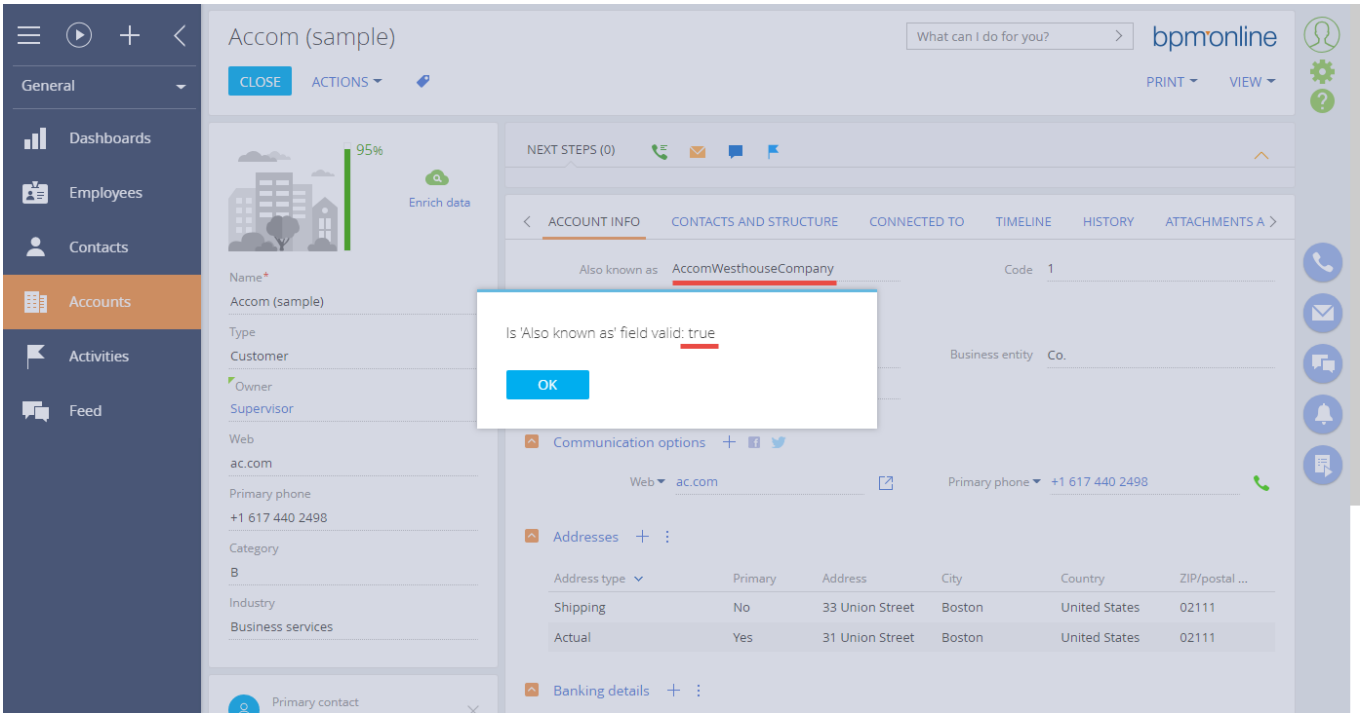


Fig. 5. Correctly populated field



> **NOTE**
>
> Field validation is described in the "**How to add the field validation**" article.

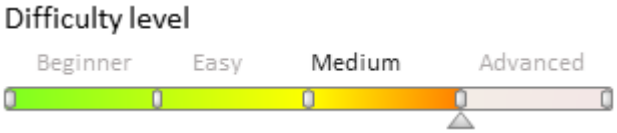
Working with WorkspaceConsole

Contents

- **WorkspaceConsole settings**

- **WorkspaceConsole parameters**
- **Exporting packages from database**
- **Saving packages to the database**
- **Saving SVN packages**

WorkspaceConsole settings



Introduction

The WorkspaceConsole utility is designed for working with bpm'online packages. Use the utility to:

- Export packages from development environments and migrate them to test environments or production environments (the packages are saved as archives).
- Install new packages when upgrading or migrating from development environments.
- Import and export schema resources and data for localization.
- Work with configuration schemas.

The utility executable file (*Terrasoft.Tools.WorkspaceConsole.exe*) is located in the bpm'online application directory:

```
[Path to the catalog with installed
application]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\
```

The build version of the utility must match the build version of the application.

> ATTENTION
>
> When updating the application, the build version of WorkspaceConsole must correspond to the target build version. For example, if the current version of the bpm’online build is 7.11.1.1794, and you need to update the packages to 7.11.2.1658, then you must use WorkspaceConsole version 7.11.2.1658.

Setting up the utility

WorkspaceConsole works directly with the bpm'online application database. Thus, it is necessary to specify database connection string in the configuration file (*Terrasoft.Tools.WorkspaceConsole.exe.config*) for the utility to work properly.

Recommended sequence:

1. Check the *connectionStringName* attribute of the <db> element of the configuration file for the connection string name. In the current example, the *connectionStringName* attribute value is “db”.

```
<terrasoft>
 ...
  <db>
    <general connectionStringName="db"
securityEngineType="Terrasoft.DB.MSSql.MSSqlSecurityEngine, Terrasoft.DB.MSSql" ...
/>
  </db>
  ...
</terrasoft>
```

2. Find the connection string in the <connectionStrings> element of the configuration file. The *name* attribute will

match the *connectionStringName* attribute of the <db> element. In the current example, the *name* attribute value is “db”.

```
<connectionStrings>
 ...
  <add name="db" ... />
 ...
</connectionStrings>
```

> NOTE
>
> By default, the configuration file contains two connection strings. The "db” string is used for connecting to the MS SQL Server database. The "dbOracle” string is used for connecting to the Oracle database.

3. Modify the value of the *connectionString* attribute, so that it matches the value used in the connection string of the application's *ConnectionStrings.config* file (or simply is set to the correct database). For more information on modifying the settings in *ConnectionStrings.config* file, as well as their purpose, please refer to the "Bpm’online setup FAQ" article. Example of the <connectionStrings> section:

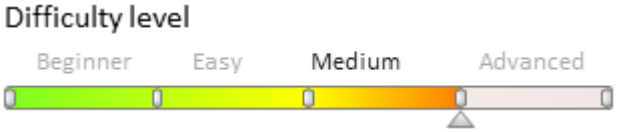
```
<connectionStrings>
    <add name="db" connectionString="Data Source=dbserver\MSSQL2016; Initial
Catalog=YourDBName; Persist Security Info=True; MultipleActiveResultSets=True;
Integrated Security = true; Pooling = true; Max Pool Size = 100" />
    <add name="dbOracle" connectionString="Data Source=(DESCRIPTION = (ADDRESS_LIST =
(ADDRESS = (PROTOCOL = TCP)(HOST = dbOracleServer.yourdomain.com)(PORT = 1521)))
(CONNECT_DATA = (SERVICE_NAME = TSOra10) (SERVER = DEDICATED)));User
Id=BPMonlineUser;Password=BPMonlineUserPassword;" />
  </connectionStrings>
```

> NOTE
>
> To perform a one-time operation with *WorkspaceConsole*, run the utility with the *webApplicationPath* parameter. Specify the path to the application directory in this parameter. In this case, the utility will independently determine all necessary database connection settings from the *ConnectionStrings.config* file. The database connection parameters in the *Terrasoft.Tools.WorkspaceConsole.exe.config* file will be ignored.

4. Run one of the two pre-installed .cmd files to install the proper bit version of the utility. For 32-bit operating systems, run *PrepareWorkspaceConsole.x86.bat*. For 64-bit operating systems, run *PrepareWorkspaceConsole.x64.bat*.

> ATTENTION
>
> If you plan on using *WorkspaceConsole* for operations with SVN, then copy the following files from the ...\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\x86 catalog (for 32-bit operating systems) or ...\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\x64 catalog (for 64-bit operating systems):
>
> - SharpPlink-x64.svnExe;
> - SharpSvn.dll;
> - SharpSvn-DB44-20-x64.svnDll.
>
> Place these files in the ...\Terrasoft.WebApp\DesktopBin\WorkspaceConsole catalog.

WorkspaceConsole parameters



Introduction

The WorkspaceConsole utility is designed to work with bpm'online packages. Use the utility to:

- Export packages from development environments and migrate them to test environments or production environments (the packages are saved as archives).
- Install new packages when upgrading or migrating from development environments.
- Import and export schema resources and data for localization.
- Create and transfer workspaces between applications.
- Work with configuration schemas.

Because the WorkspaceConsole utility is multifunctional, it must be run with certain parameters. Parameter values are passed as command-line arguments when the utility starts. Parameters are used to configure WorkspaceConsole to perform specific operations. Utility parameters are not case sensitive.

WorkspaceConsole parameters

The -help parameter

Run WorkspaceConsole with this parameter to see the full list of parameters with their brief description. If you specify other parameters, they will be ignored.

The -operation parameter

Specify the required operation name here. This parameter is required. The default value is *LoadLicResponse*. Possible method parameters are listed in table 1.

Table 1. WorkspaceConsole parameters

Operation	Description
LoadLicResponse	Saves licenses to the database (specified in the connection string). The only operation that does not require the *-workspaceName* parameter.
SaveRepositoryContent	Saves the contents of zip archives specified in the *-contentTypes* parameter from the directory specified in the *-sourcePath* parameter to the directory specified in the *-destinationPath* parameter.
SaveDBContent	Saves database content to the file system. Content type is determined by the *contentTypes* parameter value. The *destinationPath* parameter is used to specify the path in a file system. One of the following parameters must be specified: *-webApplicationPath* or *-configurationPath*.
SaveVersionSvnContent	Saves the package hierarchy (zip-archives) to the *destinationPath* directory from several SVN repositories, separated by commas in the *sourcePath* parameter.
RegenerateSchemaSources	Performs the regeneration of source codes and their compilation.
InstallFromRepository	Saves the latest version of the SVN structure and metadata into the database. Bound SQL-scripts, source code regeneration, and bound data installation are performed if necessary. This parameter only works with new or modified packages and their elements. One of the following parameters must be specified: *-webApplicationPath* or *-configurationPath*.
InstallBundlePackages	Installs the set of comma-separated packages specified in the *-packageName* parameter to the workspace specified in the *-workspaceName* parameter. One of the following parameters must be specified: *-webApplicationPath* or *-configurationPath*.
PrevalidateInstallFromRepository	Checks if zip archive package installation is available.
ConcatRepositories	Merges multiple repositories.
ConcatSVNRepositories	Merges multiple SVNrepositories.
ExecuteProcess	Starts the business process execution in the configuration (if the process is found).

UpdatePackages	Updates the packages (the *-packageName* parameter) that are located in the product package hierarchy (the *-productPackageName* parameter) in the application database. One of the following parameters must be specified: *-webApplicationPath* or *-configurationPath*.
BuildWorkspace	Compiles the workspace (configuration). Used for developing schemas in VisualStudio (see: “**Working with the server side source code in Visual Studio**”).
ReBuildWorkspace	Compiles the workspace (configuration) entirely. Used for developing schemas in VisualStudio (see: “**Working with the server side source code in Visual Studio**”).
UpdateWorkspaceSolution	Updates the Visual Studio project solution and files (see: “**Working with the server side source code in Visual Studio**”).
BuildConfiguration	Generates static content in the file system (see: "**Client static content in the file system**"). Uses the following parameters: *-workspaceName*, *-destinationPath*, *-webApplicationPath*, *-logPath*, *-force*. If the *-force* parameter is set to “true”, static content is generated for all schemas. If the *-force* parameter is set to “false”, static content is generated for modified schemas only. One of the following parameters must be specified: *-webApplicationPath* or *-configurationPath*.

The -user parameter

Authorization username. Only specified if this information is missing from the configuration utility file or if it is necessary to perform the operation on behalf of another user.

The - password parameter

Authorization password. Only specified if this information is missing from the configuration utility file or if it is necessary to perform the operation on behalf of another user.

The -workspaceName parameter

The name of the workspace (configuration) used to perform the operation.

The -autoExit parameter

Used to automatically terminate the utility process after the operation is completed. Available values – *true* or *false*. Default value – *false*.

The -processName parameter

The name of the process that needs to start.

The -repositoryUri parameter

The SVN directory path for storing the package structure and metadata (optional). Overrides the same configuration property specified in the *-workspaceName* parameter.

The -sourceControlLogin parameter

SVN repository username.

The -sourceControlPassword parameter

SVN repository password.

The -workingCopyPath parameter

Local directory of working package copies, stored in SVN.

The -contentTypes parameter

Content type (for example, resources) extracted from packages. Possible values are listed in table 2.

Table 2. Possible content type values

Content type	Description
SystemData	System diagram data in JSON format. All system schemas and their columns are saved (except for those specified in the *-excludedSchemas* parameter).
ConfigurationData	Configuration schema data in JSON format. All system schemas and their columns are saved (except for those specified in the *-excludedSchemas* parameter).
Resources	Resources of localizable configuration schemas in XML format.
LocalizableData	Resources of localizable configuration schemas in XML format. Only text columns are saved. Additional restrictions are specified in the *-excludedSchemas* and *-excludedSchemaColumns* parameters.
Repository	Workspace data in zip format.
SqlScripts	Package SQL scripts.
Data	Both system and configuration data in JSON format. A combination of the *SystemData* and *ConfigurationData* values.
LocalizableSchemaData	Localizable object data.
All	All content types.

The -sourcePath parameter

Local disk catalog path with the necessary data (e.g. packages, schemas, resources). This paramater can take several comma-separated values for the *ConcatRepositories* and *SaveVersionSvnContent* operations.

The -destinationPath parameter

Local disk catalog path for the necessary data (e.g. packages, schemas, resources).

The -webApplicationPath parameter

The bpm’online application path. This path is used by the ConnectionStrings.config file to read database connection data. If this parameter has not been indicated, the connection to the database specified in the connection string of the utility configuration file will be established. If this parameter has been indicated, the connection will be established with the database specified in the ConnectionStrings.config file of the bpm'online application.

> ⚠ Attention
>
> For *BuildWorkspace*, *ReBuildWorkspace*, and *UpdateWorkspaceSolution* operations, the *-webApplicationPath* parameter must contain the path to the Terrasoft.WebApp folder.

The -configurationPath parameter

Path to the *Terrasoft.Configuration* subfolder in the application folder. For example, *C:\bpmonline7.11.1\Terrasoft.WebApp\Terrasoft.Configuration*. In this folder, source codes and resources of custom package schemas are exported in the **file system development mode**.

The -filename parameter

File name. This parameter is required for the *LoadLicResponse* operation.

The -excludedSchemas parameter

Names of excluded schemas.

The -excludedSchemaColumns parameter

Names of excluded schema columns.

The -excludedWorkspaceNames parameter

Names of excluded workspaces.

The -includedSchemas parameter

Names of forcibly used schemas.

The -includedSchemaColumns parameter

Names of forcibly used schema columns.

The -cultureName parameter

The language culture code. Required if you use the *Resources* and/or *LocalizableData* values of the *-contentTypes* parameter.

The -schemaManagerNames parameter

Names of schema managers. Default value – *EntitySchemaManager*.

The -packageName parameter

The workspace package name (optional parameter). The package is specified in the *-workspaceName* parameter. Please note that all dependent packages will used as well. If this parameter has not been indicated, all workspace packages will be used.

The -clearWorkspace parameter

Indicates whether the workspace needs to be cleared before updating. Available values – *true* or *false*. Default value – *false*.

The -installPackageSqlScript parameter

Indicates the need to execute SQL scripts before and after saving the packages. Available values – *true* or *false*. Default value – *true*.

The -installPackageData parameter

Indicates the need to install bound data before and after saving the packages. Available values – *true* or *false*. Default value – *true*.

The -updateDBStructure parameter

Indicates the need to update the database structure before and after saving the packages. Available values – *true* or *false*. Default value – *true*.

The -regenerateSchemaSources parameter

Indicates the need to regenerate source codes after saving the packages. Available values – *true* or *false*. Default value – *true*.

The -continueIfError parameter

Indicates the need to abort the installation process upon encountering the first error. If the parameter value is *true*, the user will receive the error list once the installation is complete. Available values – *true* or *false*. Default value – *false*.

ATTENTION

The *InstallFromSvn* and *InstallFromRepository* operations work with new or modified packages and their elements. The system compares the new and modified package structures to identify modified element. If the user runs a command (e.g. *InstallFromSvn*) without specifying the *continueIfError=true* key and receives an error, the command will restart for same configuration without errors, but also without modifying the

database. This happens because the previous operation synchronized the package structures and storage of the specified configuration, and the current operation does not have any modified elements.

The -skipCompile parameter

Indicates the need to perform a compilation phase. Works only if the *-updateDBStructure* parameter value is *false*. Available values – *true* or *false*. Default value – *false*.

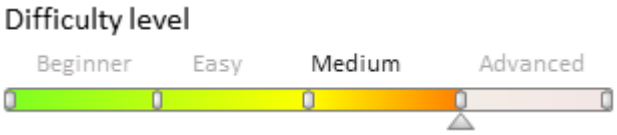
The -autoUpdateConfigurationVersion parameter

Updates the configuration version valuw to the bpm'online application version in the database. Available values – *true* or *false*. Default value – *false*.

The -warningsOnly parameter

The WorkspaceConsole utility only reports detected errors. Available values – *true* or *false*. Default value – *false*.

Exporting packages from database



Introduction

To transfer custom packages between non-shared environments (e.g. development and test environments), you must first export these packages to the file system. To save packages from the database, use the *SaveDBContent* operation of the *WorkspaceConsole* utility. Learn more about the *WorkspaceConsole* utility in the “**WorkspaceConsole parameters**” article.

NOTE

Make sure the settings of the *WorkspaceConsole* utility are correct before you run it. Please refer to the “**WorkspaceConsole settings**” article for more details.

To save packages from the database, run the *WorkspaceConsole* utility with the following parameter values:

Table 1. *WorkspaceConsole* utility parameters for saving database packages

Parameter	Value	Description
operation	*SaveDBContent*	Saves database content to the file system. Content type is determined by the contentTypes parameter value. The *destinationPath* parameter is used to specify the path in a file system.
contentTypes	*Repository*	Type of content uploaded to a file system. The *Repository* value is used to upload the workspace to a catalog specified in the *destinationPath* parameter. The name of the workspace is specified in the *workspaceName* parameter.
workspaceName	*[Workspace name]*	The name of the workspace (configuration) with the saved packages. By default, all users work in the *Default* workspace.
destinationPath	*[Path to local directory]*	Path to a local directory in the file system. Packages with the **.gz* format are saved in this directory.
webApplicationPath	*[Path to local directory]*	The bpm’online application path. This path is used by the ConnectionStrings.config file to read database connection data. If this parameter has not been indicated, the connection to the

		database specified in the connection string of the utility configuration file will be established. If this parameter has been indicated, the connection will be established with the database specified in the ConnectionStrings.config file of the bpm'online application.
configurationPath	*[Path to local directory]*	Path to the *Terrasoft.Configuration* subfolder in the application folder. For example, *C:\bpmonline7.11.1\Terrasoft.WebApp\Terrasoft.Configuration*. In this folder, source codes and resources of custom package schemas are exported in the **file system development mode**.

All workspace packages are saved in the process. It may take up to 10 minutes to complete this operation.

NOTE

Check data binding properties before saving. This includes system settings, lookups, section data etc.

Please refer to the "Binding data to package" article for more details.

Command signature for Windows command prompt that will export packages from the database:

```
[WorkspaceConsole path]\Terrasoft.Tools.WorkspaceConsole.exe -operation=SaveDBContent
-contentTypes=Repository -workspaceName=[Workspace name] -destinationPath=[Local
directory path] -webApplicationPath=[Path to application directory]
```

NOTE

We recommend using batch files (*.bat) to create and save commands.

Uploading packages to a file system

Case description

The bpm'online application is installed in the C:\bpmonline7.12.2 directory. Export all *Default* workspace packages into the C:\SavedPackages directory.

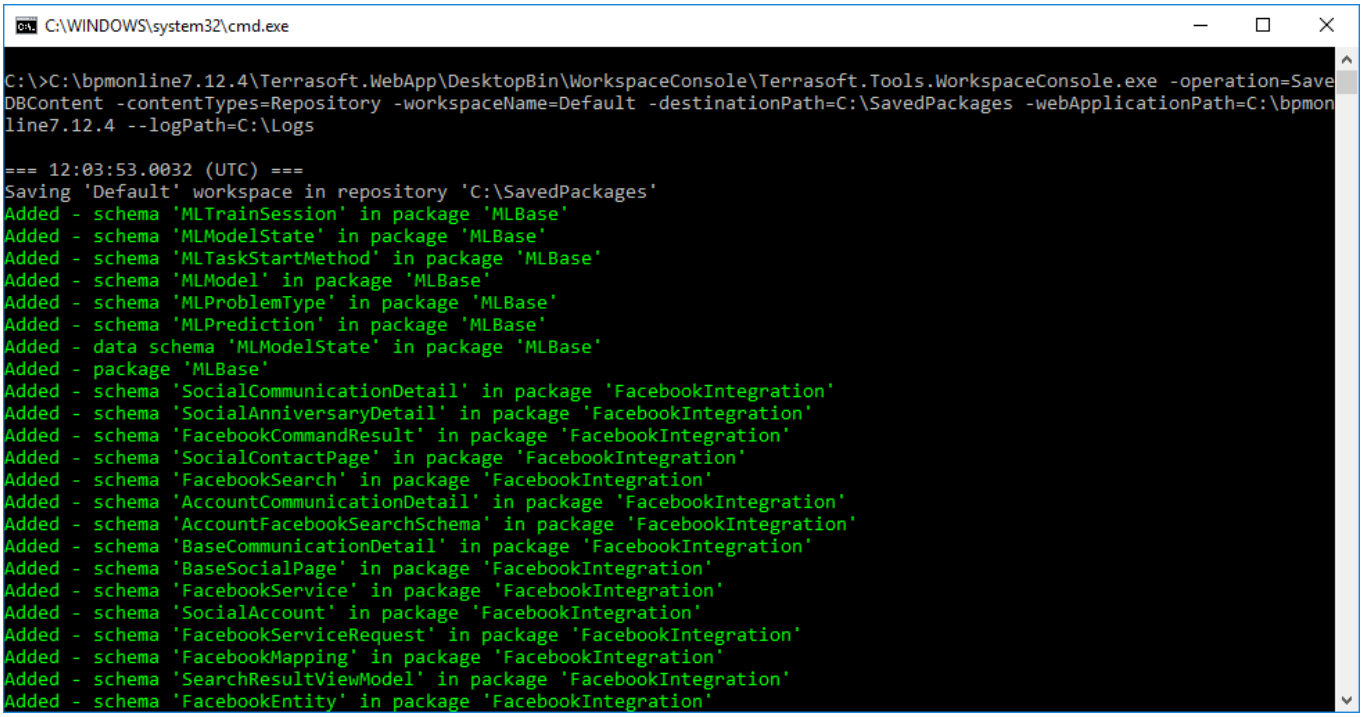
Case implementation:

Use any text editor to create a batch command file (*.bat or *.cmd) with a command that will launch the *WorkspaceConsole* utility. Enter the following command in the file:

```
C:\bpmonline7.12.4\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.Works
paceConsole.exe -operation=SaveDBContent -contentTypes=Repository -
workspaceName=Default -destinationPath=C:\SavedPackages -
webApplicationPath=C:\bpmonline7.12.4 --logPath=C:\Logs
pause
```

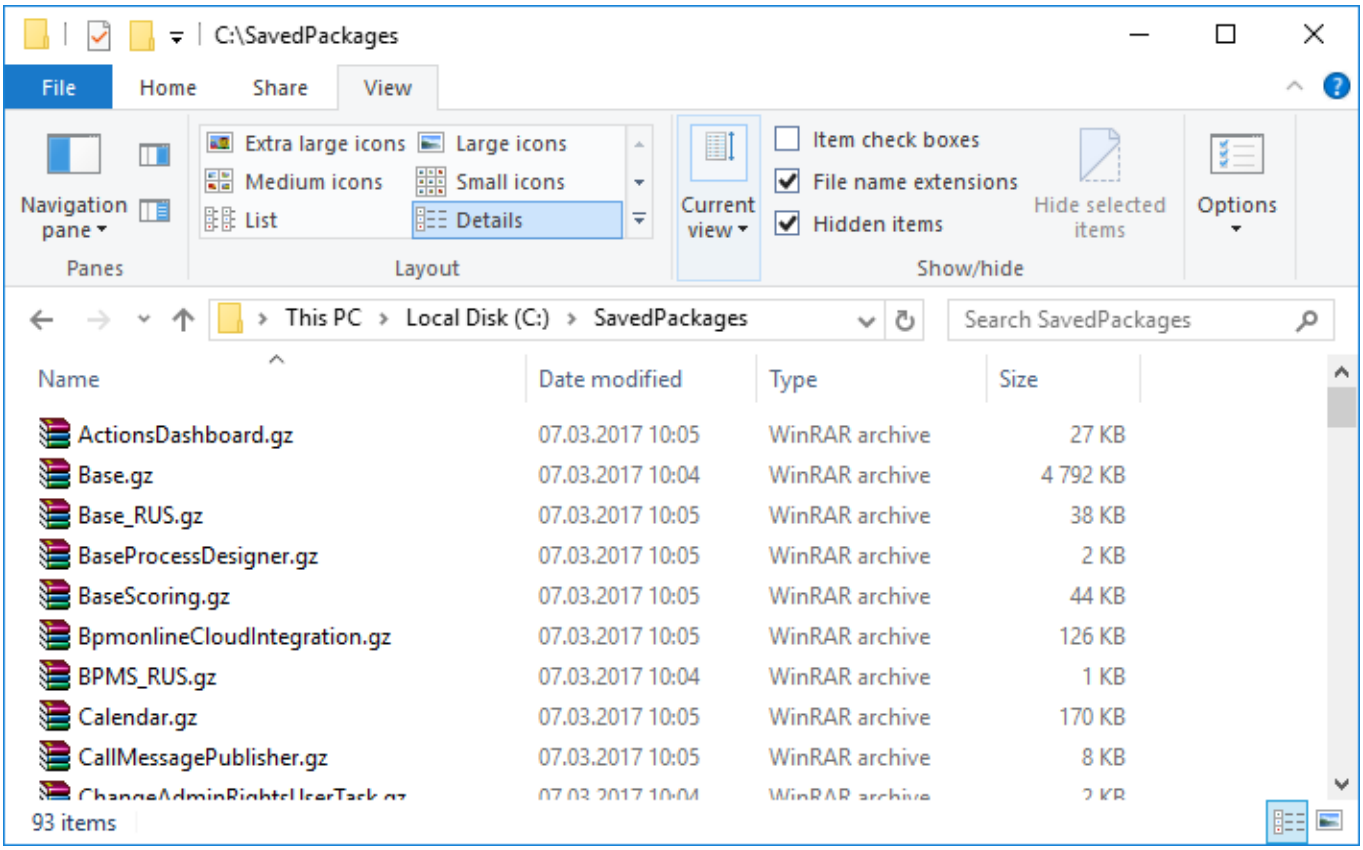
Upon saving the batch file and running it, a console window will appear, and the *WorkspaceConsole* execution process with specified parameter values will be displayed (Fig. 1).

Fig. 1. WorkspaceConsole execution



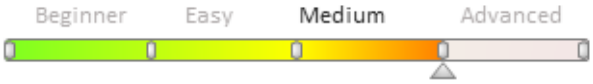
Zip-archives containing all *Default* configuration packages will be exported to the C:\SavedPackages directory (Fig. 2).

Fig. 2. Zip-archives with bpm'online packages exported to the file system



Saving packages to the database

Difficulty level



Introduction

Saving packages from the file system to the application database is performed when transferring custom packages between non-shared environments (e.g. development and test environments). Usually, packages are saved from the development environment, and loaded into the test and production environments. Learn more about saving packages in the "**Exporting packages from database**" and "**Saving SVN packages**" articles.

To load packages to the database, run the *WorkspaceConsole* utility with the following parameters:

Table 1. *WorkspaceConsole* utility parameters for loading packages to the database

Parameter	Value	Description
operation	*InstallFromRepository*	It saves the contents of packages from archives in the database. Bound SQL-scripts, source code regeneration, and bound data installation are performed if necessary. The InstallFromSvn and InstallFromRepository operations work with new or modified packages and their elements.
packageName	*[Package Name]*	The name of the package specified in the *workspaceName* configuration parameter. All dependent packages are used as well. This parameter is optional. This parameter is optional. The -clearWorkspace parameter
workspaceName	*[Workspace name]*	The name of the workspace (configuration) with the saved packages. By default, all users work in the *Default* workspace.
sourcePath	*[Path to local directory]*	Path to a local directory in the file system. This directory should include the required packages in the **.gz* format.
destinationPath	*[Path to local directory]*	Path to a local directory in the file system. The packages from the directory specified in the *sourcePath* parameter will be saved here.
skipConstraints	*false*	The option to skip foreign key creation in database tables. Available values – *true*, *false*.
skipValidateActions	*true*	The option to skip the process of table index creation verification when updating the database structure. Available values – true or false.
regenerateSchemaSources	*true*	Indicates the need to regenerate source codes after saving the packages. Available values – *true*, *false*.
updateDBStructure	*true*	Indicates the need to update the database structure before and after saving the packages. Available values – *true*, *false*.
updateSystemDBStructure	*true*	Indicates the need to update the database structure before and after saving the packages. Creates all missing system table indexes. Available values – *true*, *false*.
installPackageSqlScript	*true*	Indicates the need to execute SQL scripts before and after saving the packages. Available values – *true*, *false*.
installPackageData	*true*	Indicates the need to install bound data before and after saving the packages. Available values – *true*, *false*.
continueIfError	*true*	Indicates the need to abort the installation process upon encountering the first error. If the parameter value is true, the user will receive the error list once the installation is complete. Available values – *true*, *false*.
logPath	*[Path to local directory]*	Path to to the operation log. The log name contains the start date and time of the operation.

webApplicationPath	*[Path to local directory]*	The bpm’online application path. This path is used by the ConnectionStrings.config file to read database connection data. If this parameter has not been indicated, the connection to the database specified in the connection string of the utility configuration file will be established. If this parameter has been indicated, the connection will be established with the database specified in the ConnectionStrings.config file of the bpm'online application.
configurationPath	*[Path to local directory]*	Path to the *Terrasoft.Configuration* subfolder in the application folder. For example, *C:\bpmonline7.11.1\Terrasoft.WebApp\Terrasoft.Configuration*. In this folder, source codes and resources of custom package schemas are exported in the **file system development mode**.

Command signature for Windows command prompt that will export packages from the database:

```
[The WorkspaceConsole utility path]\Terrasoft.Tools.WorkspaceConsole.exe -packageName=
[Package name] -workspaceName=Default -operation=InstallFromRepository -sourcePath=[Path
to package archives] -destinationPath=[Archive extraction path] -skipConstraints=false -
skipValidateActions=true -regenerateSchemaSources=true -updateDBStructure=true -
updateSystemDBStructure=true -installPackageSqlScript=true -installPackageData=true -
continueIfError=true -webApplicationPath=[Path to application folder] -logPath=[Log path]
```

ATTENTION

The *WorkspaceConsole* utility makes direct changes to the database, and therefore they become available only after restarting the application in IIS.

ATTENTION

Packages loaded into the application using *WorkspaceConsole* are considered pre-installed and can not be modified (see: "**Package structure and contents**").

Best practices of loading packages with enabled development mode in the file system

If the WorkspaceConsole is used for loading packages to the application with the **development mode in the file system** enabled (*fileDesignMode=true*), the installation of the packages works in a special way. Source code of the modified schemas will be modified in the database but will remain unchanged in th file system. In this regard, if you open the schema in the designer, the unchanged source code from the file system will be displayed. At the same time, the schema modification date is changed, which brings more confusion, as the schema transfer looks completed.

Due to this, it is not recommended to use the *InstallFromRepository* operation for transferring changes from the application with enabled development mode in the file system (*fileDesignMode=true*). If this operation is required, then you will need to perform the [Download packages to file system] action to upload source codes to the file system after operation is complete.

Saving packages to the database

Case description

The bpm’online application is installed in the C:\bpmonline7.12.4 directory. Save the *userPackage* package to the *Default* workspace. The package archive is located in the C:\SavedPackages directory. Extract package contents to the C:\TempPackages directory. Save the operation log file to the C:\Log directory.

Case implementation:

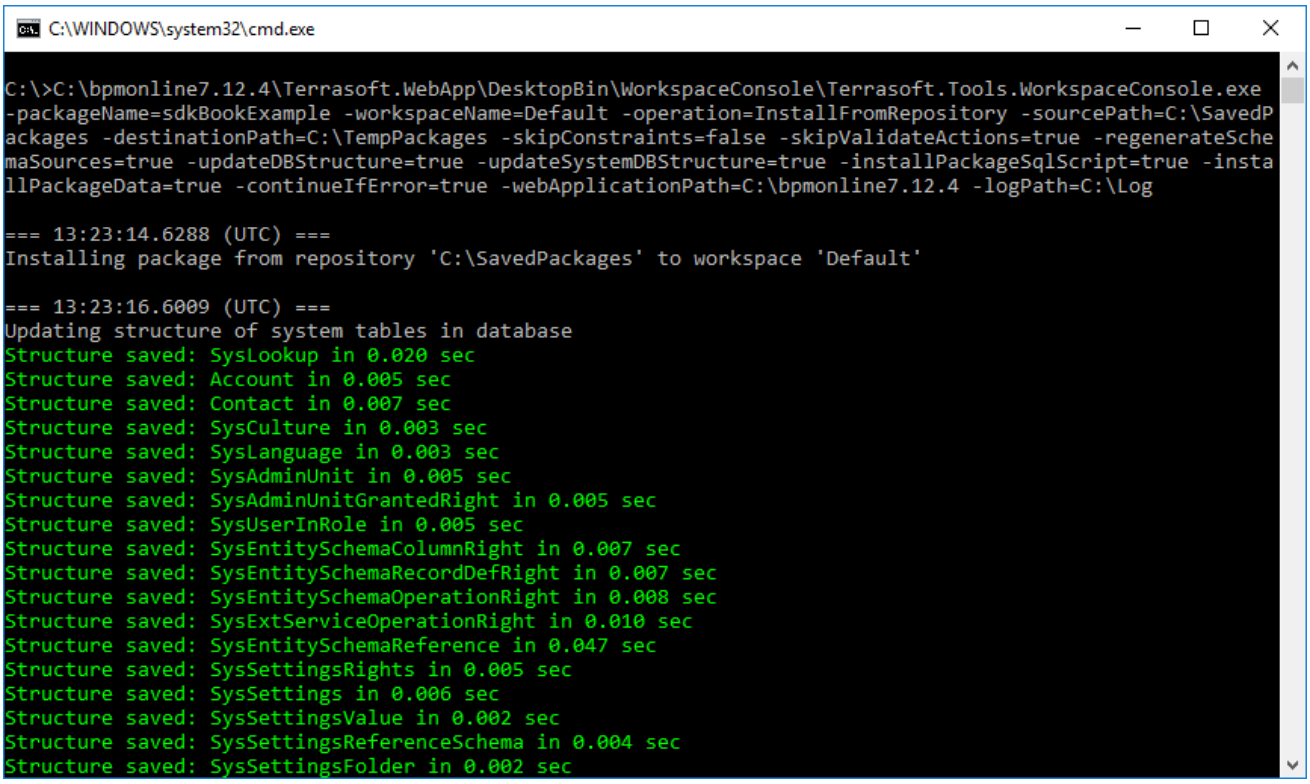
Use any text editor to create a batch command file (*.bat or *.cmd) with a command that will launch the *WorkspaceConsole* utility. Enter the following command in the file:

```
C:\bpmonline7.12.4\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.Workspace
```

```
Console.exe -packageName=sdkBookExample -workspaceName=Default -
operation=InstallFromRepository -sourcePath=C:\SavedPackages -
destinationPath=C:\TempPackages -skipConstraints=false -skipValidateActions=true -
regenerateSchemaSources=true -updateDBStructure=true -updateSystemDBStructure=true -
installPackageSqlScript=true -installPackageData=true -continueIfError=true -
webApplicationPath=C:\bpmonline7.12.4 -logPath=C:\Log
pause
```

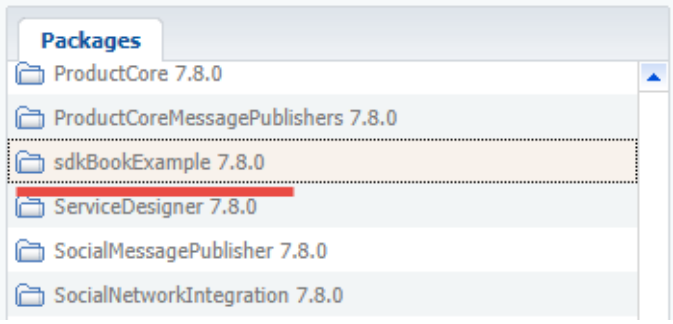
Upon saving the batch file and running it, a console window will appear, and the *WorkspaceConsole* execution process with specified parameter values will be displayed (Fig. 1).

Fig. 1. Saving a package to the application database



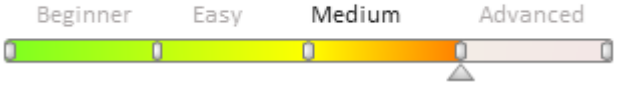
Run the command to load the *sdkBookExample* package to the *Default* configuration.

Fig. 2. The package in the [Configuration] section



Saving SVN packages

Difficulty level



Introduction

To transfer custom packages between non-shared environments (e.g. development and test environments), you must first export these packages to the file system. To save packages from the SVN repository, use the *SaveVersionSVNContent* operation of the *WorkspaceConsole* utility. Learn more about the *WorkspaceConsole* utility in the "**WorkspaceConsole parameters**" article.

NOTE

Make sure the settings of the *WorkspaceConsole* utility are correct before you run it. Please refer to the "**WorkspaceConsole settings**" article for more details.

To save SVN packages, run the *WorkspaceConsole* utility with the following parameter values:

Table 1. *WorkspaceConsole* utility parameters for saving SVN packages

Parameter	Value	Description
operation	*SaveVersionSvnContent*	Saves the package hierarchy (zip-archives) to the *destinationPath* directory from several SVN repositories, separated by commas in the *sourcePath* parameter.
destinationPath	*[Path to local directory]*	Path to a local directory in the file system. Packages with the **.gz* format are saved in this directory.
workingCopyPath	*[Path to local directory]*	Local directory of working package copies, stored in SVN.
sourcePath	*[SVN repository path]*	The SVN directory path for storing the package structure and metadata. Separate the values with a comma to specify multiple directories.
packageName	*[Package Name]*	The name of the repository package used in the operation. All dependent packages are used as well.
packageVersion	*[Package version]*	The version of the repository package used in the operation.
sourceControlLogin	*[SVN username]*	SVN repository username.
sourceControlPassword	*[SVN password]*	SVN repository password.
cultureName	*[Language culture]*	The language culture code. For example, *es-ES*.
excludeDependentPackages	*true or false*	This checkbox identifies whether the packages need to be saved. The package specified in the *packageName* parameter depends on this checkbox.
logPath	*[Path to local directory]*	The path to the directory in which the operation log file will be saved (optional).

Command signature for Windows command prompt that will export packages from the SVN:

```
[Path to WorkspaceConsole]\Terrasoft.Tools.WorkspaceConsole.exe -
operation=SaveVersionSvnContent -destinationPath=[Path to local folder] -
workingCopyPath=[Path to local folder] -sourcePath=[Path to SVN storage] -
packageName=somePackage -packageVersion=7.8.0 -sourceControlLogin=User -
sourceControlPassword=Password -logPath=[Path to local folder] -cultureName=ru-RU -
excludeDependentPackages=true
```

Uploading packages to a file system

Case description

Save the userPackage package to the C:\SavedPackages directory from the SVN repository, found at http://server-svn:8050/svn/Packages. Language culture - Russian. Save the operation log file to the C:\Log directory. Place the working copy of the package in the C:\WorkingCopy folder. SVN username – “User”. SVN password – “Password”. The bpm’online application is installed in the C:\bpmonline7.12.0 directory.

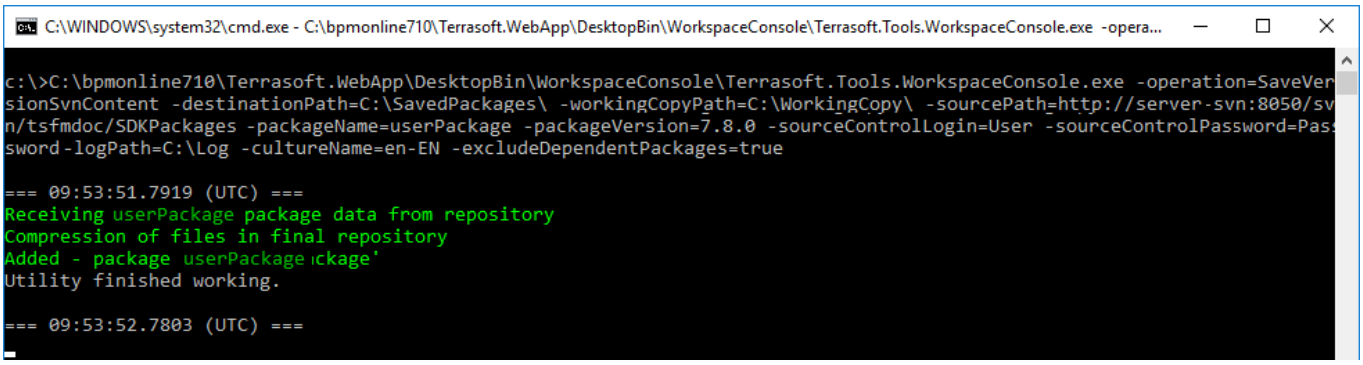
Case implementation:

Use any text editor to create a batch command file (*.bat or *.cmd) with a command that will launch the *WorkspaceConsole* utility. Enter the following command in the file:

```
C:\bpmonline7.12.0\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.Works
paceConsole.exe -operation=SaveVersionSvnContent -destinationPath=C:\SavedPackages\ -
workingCopyPath=C:\WorkingCopy\ -sourcePath=http://server-svn:8050/svn/Packages -
packageName=userPackage -packageVersion=7.8.0 -sourceControlLogin=User -
sourceControlPassword=Password -logPath=C:\Log -cultureName=ru-RU -
excludeDependentPackages=true
pause
```

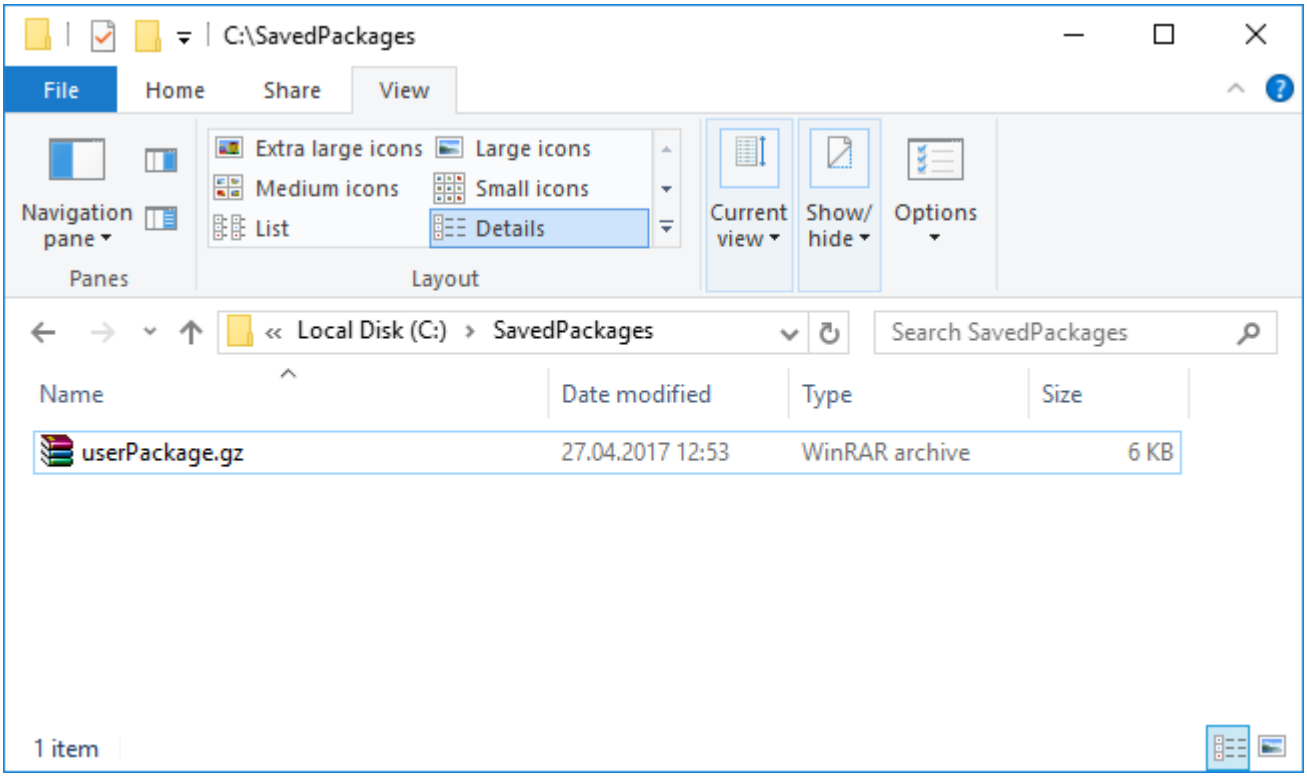
Upon saving the batch file and running it, a console window will appear, and the *WorkspaceConsole* execution process with specified parameter values will be displayed (Fig. 1).

Fig. 1. The process of saving a package from the repository



The *userPackage* package will be exported to the C:\SavedPackages directory (Fig. 2).

Fig. 2. Saved zip-archive with userPackage package



Client code debugging

Difficulty level

Introduction

The client part of the bpm'online application is represented by configuration schemas (modules), described in JavaScript language. Debugging of the source code of configuration schemas is executed directly from the browser. Developer tools that provide for debugging for all browsers, supported by bpm'online, are used for this purpose.

To run tools for client debugging, execute the following command in a browser:

- Chrome: *F12* or *Ctrl* + *Shift* + *I*.
- Firefox: *F12*.
- Internet Explorer: *F12*.

Possibilities for debugging of bpm'online client code

All supported browsers provide mostly similar capabilities for debugging client code. Most common and frequently used debugging methods are listed below. For more details about debugging with browser tools, see the following documentation:

- Chrome developer tools
- Firefox developer tools
- Internet Explorer developer tools

Scripts and breakpoints

You can view the full list of scripts, connected to the page and downloaded to a content by means of developer tools. Open any script to set a breakpoint in the place where you want to stop execution of a source code. In the stopped code, you can view current values of variables, execute commands etc.

To set a viewpoint, take the following actions:

- open necessary script file (for example, execute name lookup by combination of buttons Ctrl+O and Ctrol+P);
- go to code string where you want to set a breakpoint (for example, execute script lookup on the basis of method name);
- set a breakpoint by one of the following methods: click string name, press F9 button or select "Add breakpoint" item in right-click menu (cursor should be in the string, to which you want to add breakpoint).

You can also set a conditional breakpoint, for which you should set a condition for activation of the breakpoint.

You can also break an execution process directly from the code by the debugger command:

```
function customFunc (args) {
  ...
  debugger; // <-- debugger stops here.
  ...
}
```

Execution control

The debugging process is reduced to breaking of code execution at the breakpoint, verification of variable values and call stack. Code tracing is executed further for detection of fragments where program behavior deviates from predicted behavior.

The following command is used in browser debuggers for code-based turn-by-turn navigation (figure 1, figure 2 and figure 3):

- suspend/continue script execution (1);
- perform step whithout entering the function (2);
- perform step whith entering the function (3);
- perform step before exiting from current function (4).

Fig. 1. — Navigation panel in Chrome browser debugger

Fig. 2. — Navigation panel in Firefox browser debugger

Fig. 3. — Navigation panel in Internet Explorer browser debugger

Chrome browser provides an additional two commands for execution control:

- deactivate all breakpoints (5);
- deactivate/activate automatic break in case of error (6).

For more information about possibilities and commands of navigation panel for a browser, see corresponding documentation.

Browser console use

In the course of debugging, you can execute JavaScript commands, display debugging, trace information, execute

measurements and code profiling. The console object is used for this purpose.

JavaScript commands calling

To start operation of the browser console, you should open it by going to the [Console] tab or opening it in addition to the debugger, using the [Esc] button. You can then enter commands in javaScript and start their execution by pressing [Enter].

Debug information output

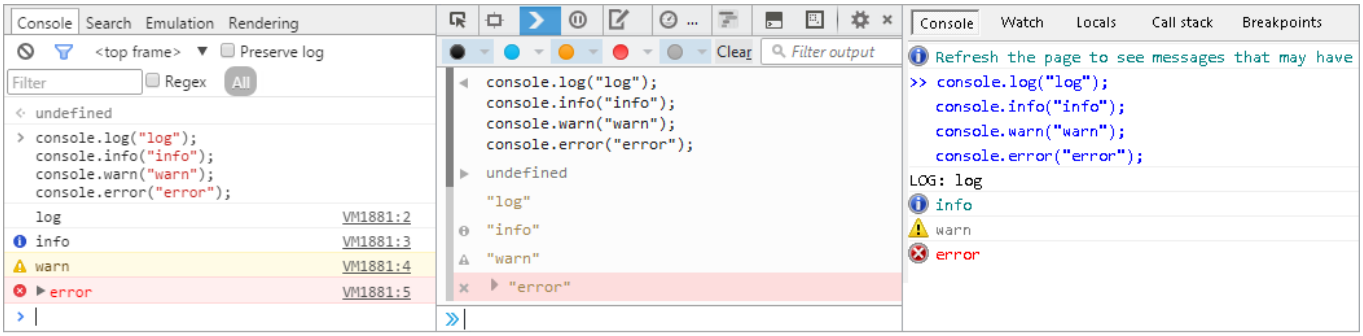
You can enable debugging information of a different nature, i.e. info messages, warnings and error messages, in the console. For this purpose you can use corresponding console object methods (table 1).

Table 1. — Console methods for output of debug messages.

Method	Description	Chrome	Firefox	Internet Explorer
console.log(object [, object, ...])	Outputs arguments in console and separate them with comma. Used for enabling different general messages.	+	+	+
console.info(object [, object, ...])	Similar to log() method but outputs messages in other style (figure 4) and emphasizes their significance.	+	+	+
console.warn(object [, object, ...])	Outputs warning message in console.	+	+	+
console.error(object [, object, ...])	Outputs error message in console.	+	+	+ (8+)

An individual style is used for each type of outputted message (figure 4).

Fig. 4. — Styles of different types of console messages



The represented console methods support formatting of outputted messages. This means that you can use special controlling sequences (templates) that will be replaced by corresponding values (arguments, additionally transferred to the function).

Console methods support the following formation templates (table 2).

Table 2. — Console message formation templates

Template	Data type	Example of use
%s	String	console.log("%s is one of flagship products of a company %s", "bpm'online sales", "Terrasoft");
%d, %i	Number	console.log("Platform %s was issued for the first time ever in %d year", "bpm'online", 2011);
%f	Float	console.log("Pi character is equal to %f", Math.PI);

%o	DOM-item (it is not supported by IE)	console.log("DOM-View of item <body/>: %o", document.getElementsByTagName("body")[0]);
%O	JavaScript Object (is not supported by IE and Firefox)	console.log("Object: %O", {a:1, b:2});
%c	CSS style (is not supported by IE)	console.log("%cGreen text, %cRed Text on a blue background, %cCapital letters, %cPlain text", "color:green;", "color:red; background:blue;", "font-size:20px;", "font:normal; color:normal; background:normal");

Tracing and validations

Table 3 shows console methods for tracing and verification of expressions.

Table 3. — Console methods for tracing and verification

Method	Description	Chrome	Firefox	Internet Explorer
console.trace()	Outputs call stack from code point where method was called. Call stack includes file names, string numbers and also call counters of trace() method from one and the same point.	+	+	+ (11+)
console.assert(expression[, object, ...])	Verified expression, transferred as an expression parameter and, if the expression is false, outputs error with (console.error ()) call stack in the console, otherwise it outputs nothing.	+	+ (28+)	+

Console.trace() method outputs informative stack-trace with full list of functions and their arguments at the moment of call.

Due to the console.trace() method you can comply with rules in the code and ensure that code execution results meet expectations. Using console.assert () you can execute code testing, i.e. if execution result is unsatisfactory, the corresponding value will be discarded.

An example of the console.assert() method for testing of results:

```
var a = 1, b = "1";
console.assert(a === b, "A is not equal to B");
```

Profiling and measurement

You can measure code execution time with browser console methods (table 4).

Table 4. — Console methods for measurement of code execution time

Method	Description	Chrome	Firefox	Internet Explorer
console.time(label)	Starts counter (milliseconds) with label.	+	+	+ (11+)
console.timeEnd(label)	Stops counter (milliseconds) with label and plans result in console.	+	+	+ (11+)

An example of console.time() and console.timeEnd() methods in code:

```
var myArray = new Array();
// Starts counter with Initialize myArray tag.
console.time("Initialize myArray");
```

```
myArray[0] = myArray[1] = 1;
for (i = 2; i<10; i++)
{
    myArray[i] = myArray[i-1] + myArray[i-2];
}
// Stops counter with Initialize myArray tag.
console.timeEnd("Initialize myArray");
```

You also can execute code profiling and output profiling stacks that contain detailed information about how much time was spent by a browser for definite operations.

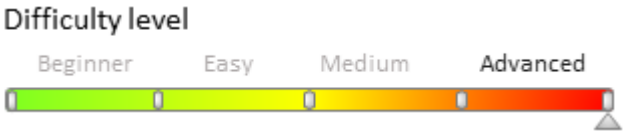
Table 5. — Console methods for code profiling

Method	Description	Chrome	Firefox	Internet Explorer
console.profile(label)	Runs Java Script profiler and displays results, marked with label.	+	+ (when DevTools panel is opened)	+ (10+)
console.profileEnd(label)	Stops Java Script profiler.	+	+ (when DevTools panel is opened)	+ (10+)

You can view profiling results in:

- Chrome — Profiles tab:
- Firefox — Performance tab;
- Internet Explorer — Profiler tab.

Server code debugging



Introduction

During the development process on the bpm'online platform, developers often need to create the source code for the server schemas of the "source code" type. These may be, for example, existing base schemas, custom configuration classes, web services or business process scripts written in C #. Debug such code is easiest with integrated debugging features of the development environment, for example, Visual Studio. The Visual Studio debugger enables developers to freeze the execution of program methods, check variable values, modify them and monitor other activities performed by the program code.

To begin debugging an application, you need to perform a number of steps:

1. Export the bpm'online configuration source code to the local directory files
2. Create a new Visual Studio project for debugging
3. Add the exported files with the source code to the Visual Studio project
4. From the project, attach to the working process of the IIS server and start debugging.

ATTENTION

Debugging the code using the method described in this article is only possible for **applications deployed on-site**.

ATTENTION

Debugging the code using the method described in this article is only possible if the development in the file

system mode is turned off (see: “**Development in the file system**”).

ATTENTION

Enable the [Suppress JIT Optimization] checkbox (the [Options] menu, the [Debugging] > [General] tabs) to be able to get the values of variables during the debugging. More information about optimized and unoptimized code during debugging can be found in the "JIT Optimization and Debugging" Visual Studio guide.

1. Exporting the configuration source code

To do this, perform the application setup.

In the Web.config file located in the root of the application (“external” Web.config), set the “true” value for the *debug* attribute of the *compilation* element.

```
<compilation debug="true" targetFramework="4.5" />
```

Save the schema to apply changes.

In the Web.config file located in the Terrasoft.WebApp directory of the application (“nternal” Web.config), specify the values for the following items:

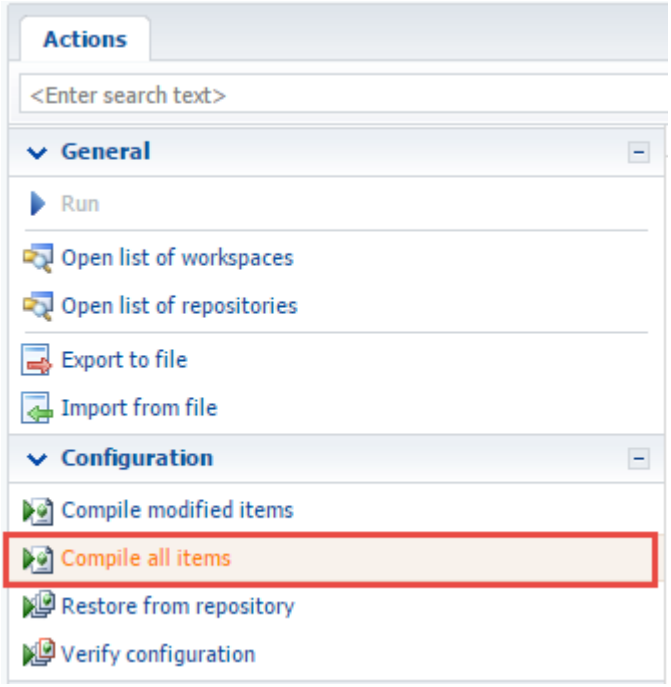
- To configure *IncludeDebugInformation*, specify the “*true*” value.
- To configure *CompilerSourcesTempFolderPath*, specify the path to the directory where the source files will be exported.
- To configure *ExtractAllCompilerSources*, set the value to *“true”* if you want to export all schemas when performing the [Compile modified items] action in the **[Configuration] section**. To export only the modified schemas, set the value to “*false*” (the default value).

```
<add key="IncludeDebugInformation" value="true" />
<add key="CompilerSourcesTempFolderPath" value="Path_to_local_catalog” />
<add key="ExtractAllCompilerSources" value="false" />
```

Save the schema to apply changes.

To export the files with server schema source code, perform the [Compile all items] (Fig. 1) action in the [Configuration] section.

Fig. 1 [Compile all items] action



At the time of compilation, the source code files for the application's configuration schemas, as well as configuration libraries, their modules and debug files (* .pdb) will be exported to the folder specified in the *CompilerSourcesTempFolderPath* configuration of the "internal" Web.config. The schema source code will be exported again every time you compile the application.

NOTE

When compiling, the source code files of the schemas of the work space under which compilation was started will be exported. The files of the downloaded source codes of configuration schemas are named in a certain format: [Name of the schema in the configuration].[Package name]_[Schema type].cs.

For example: Contact.Base_Entity.cs, ContractReport.Base_Report.cs.

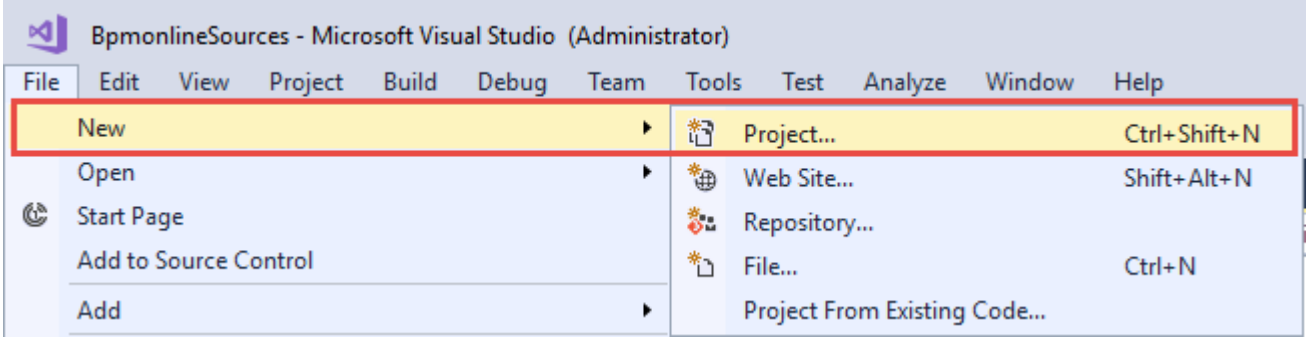
2. Creating a Visual Studio project for debugging

ATTENTION

Creating a Visual Studio project is unnecessary to debug the source code – opening the necessary files in Visual Studio is sufficient. However, if debugging is performed frequently, or you need to work with a large number of files at the same time, creating a project will make it easier.

To create a project for debugging an application in Visual Studio, execute the *File* > *New* > *Project*menu command (Fig. 2).

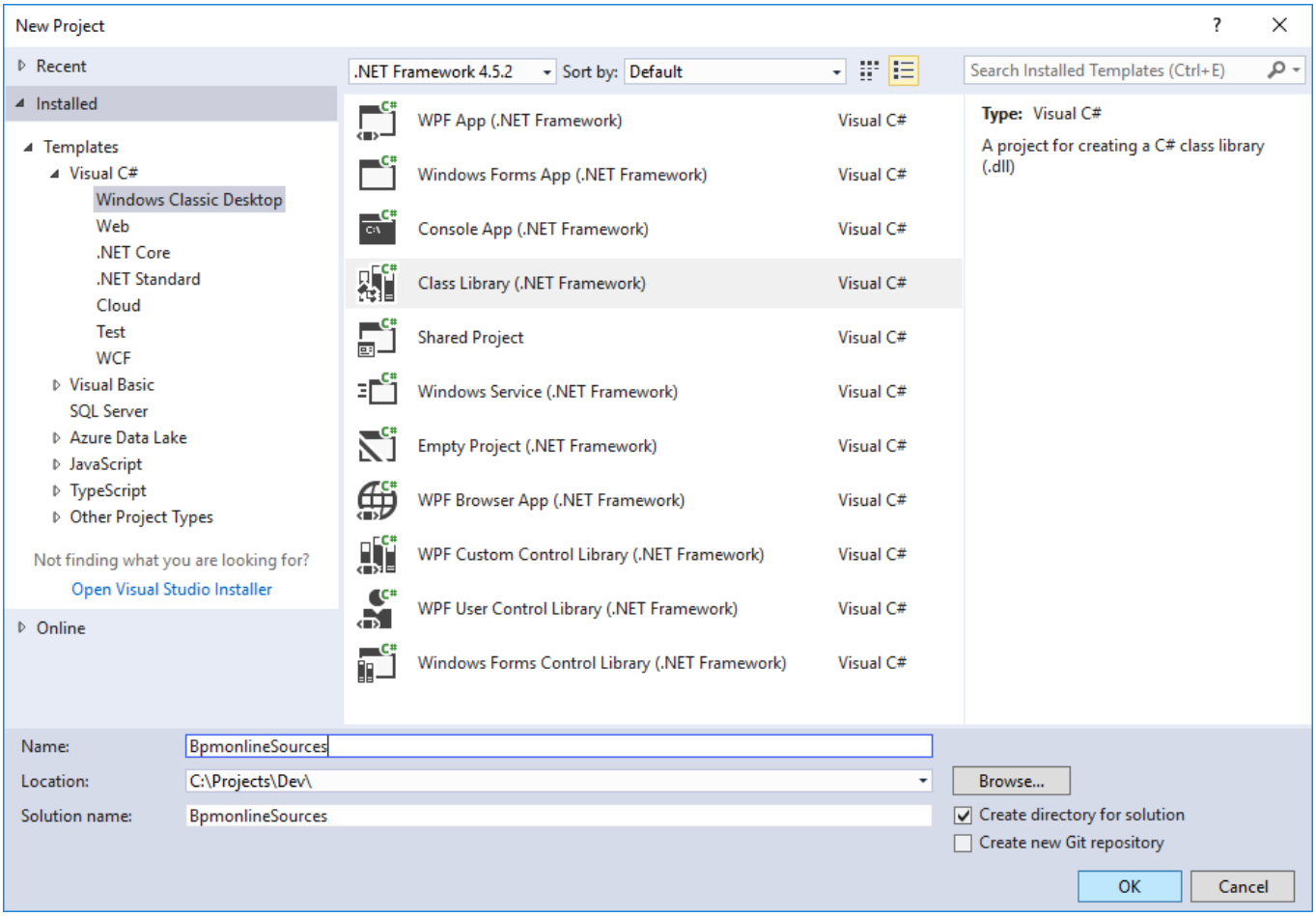
Fig. 2 Creating a new project in Visual Studio



In the properties window of the created project, select the [Class Library (.NET Framework)] project type (class

library for the classic Windows application), and specify the name and location of the project (Fig. 3).

Fig. 3 Visual Studio project properties

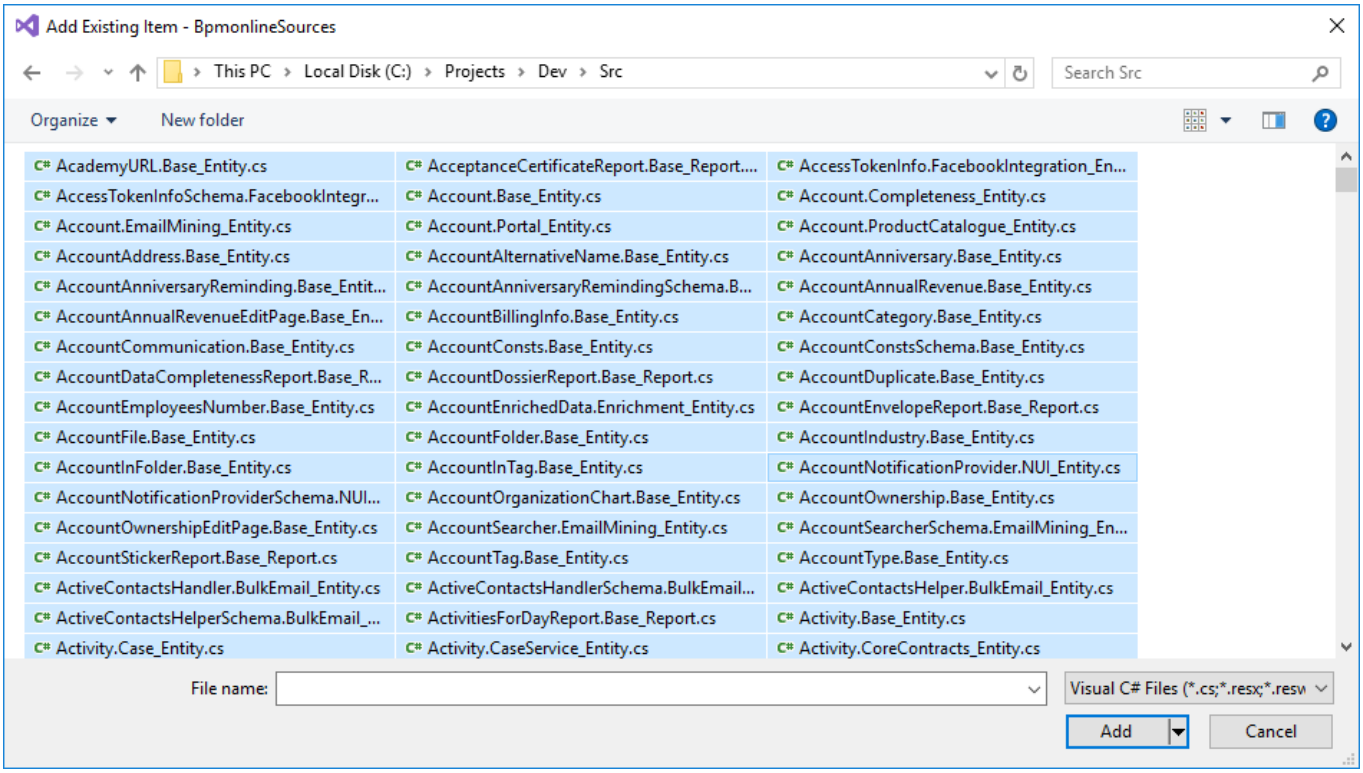


After creating the project, you need to remove an extra file from it (by default, the file Class1.cs is added to the new project) and save the project.

3. Adding the exported files with the source code to the Visual Studio project

To do this, select *Add* > *Existing* Item from the project's context menu in the solution explorer. In the dialog box that appears, you must go to the directory with the downloaded files with the source code and select all files (Fig. 4).

Fig. 4 Adding files to a project



NOTE

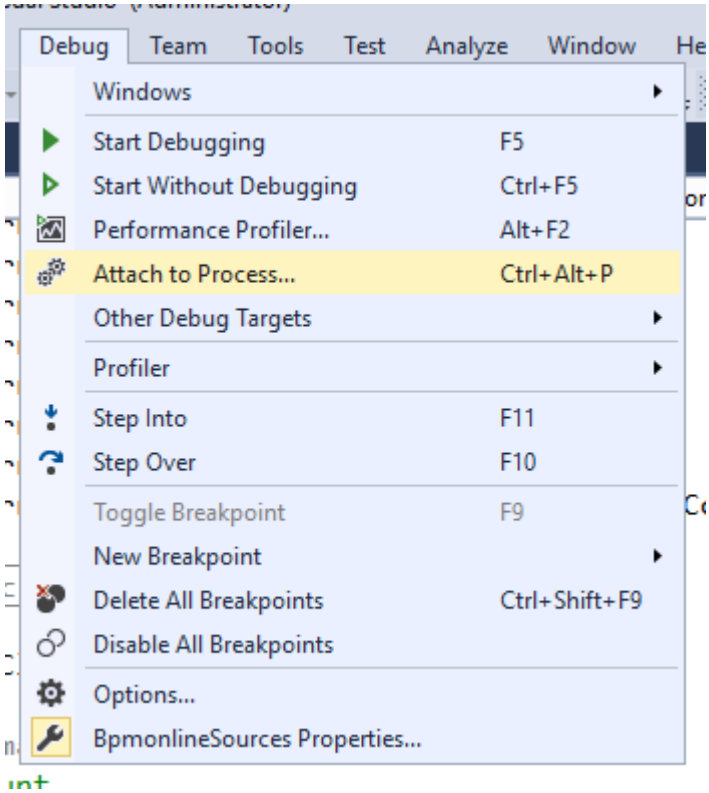
Add only the files needed for debugging to the Visual Studio project. However, the transition between methods during debugging will be limited only by the methods of classes implemented in the files added to the project.

Save the project after adding the files.

4. Attaching to the IIS process for debugging

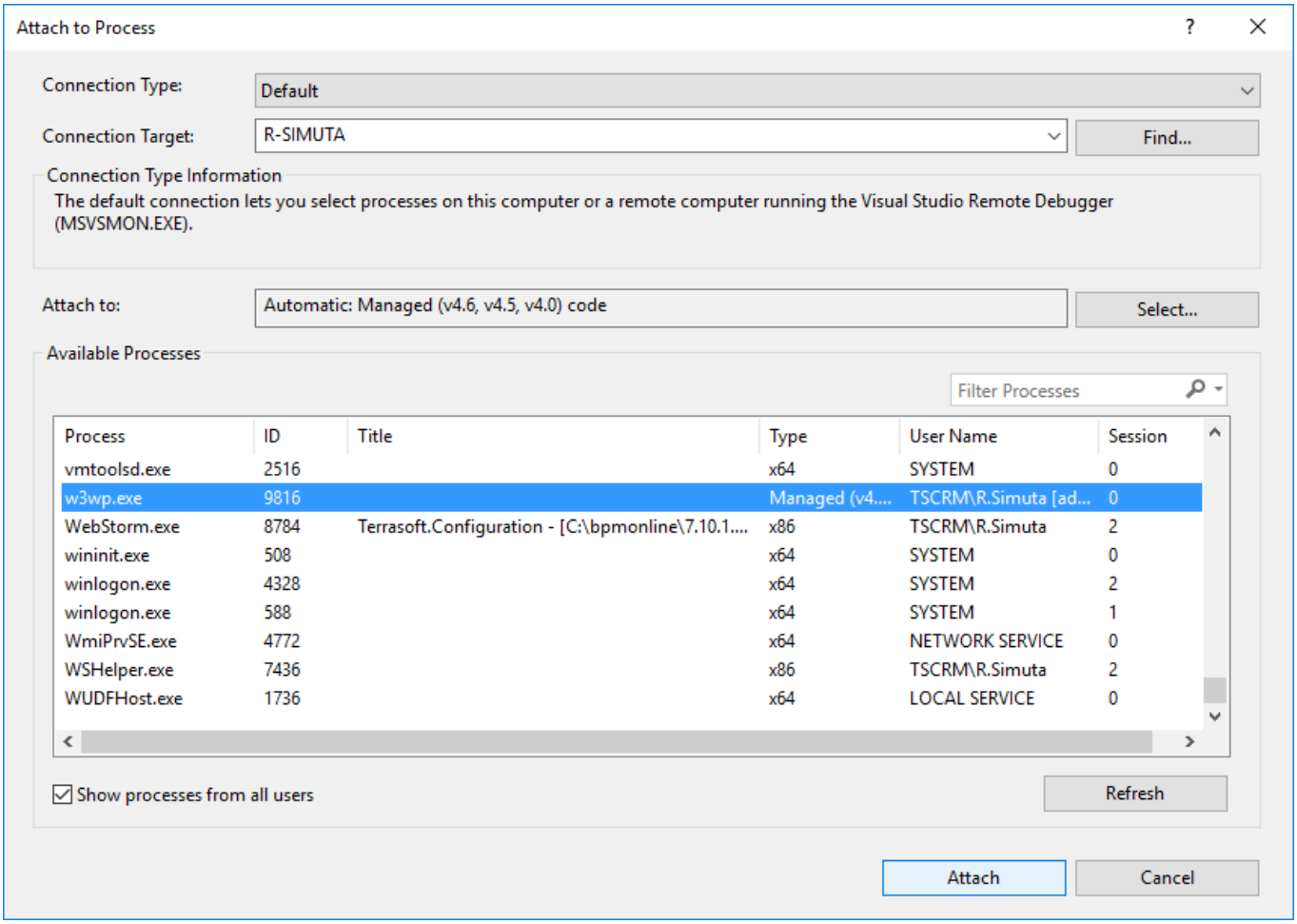
To begin debugging, attach to the IIS server process, where the application runs. To do this, select the *Debug* > *Attach to process* command in the Visual Studio menu (Fig. 5).

Fig. 5 Attaching to a process



In the opened window, select the working IIS process in the list of processes, where the application pool is running (Fig. 6).

Fig. 6 Attaching to an IIS process



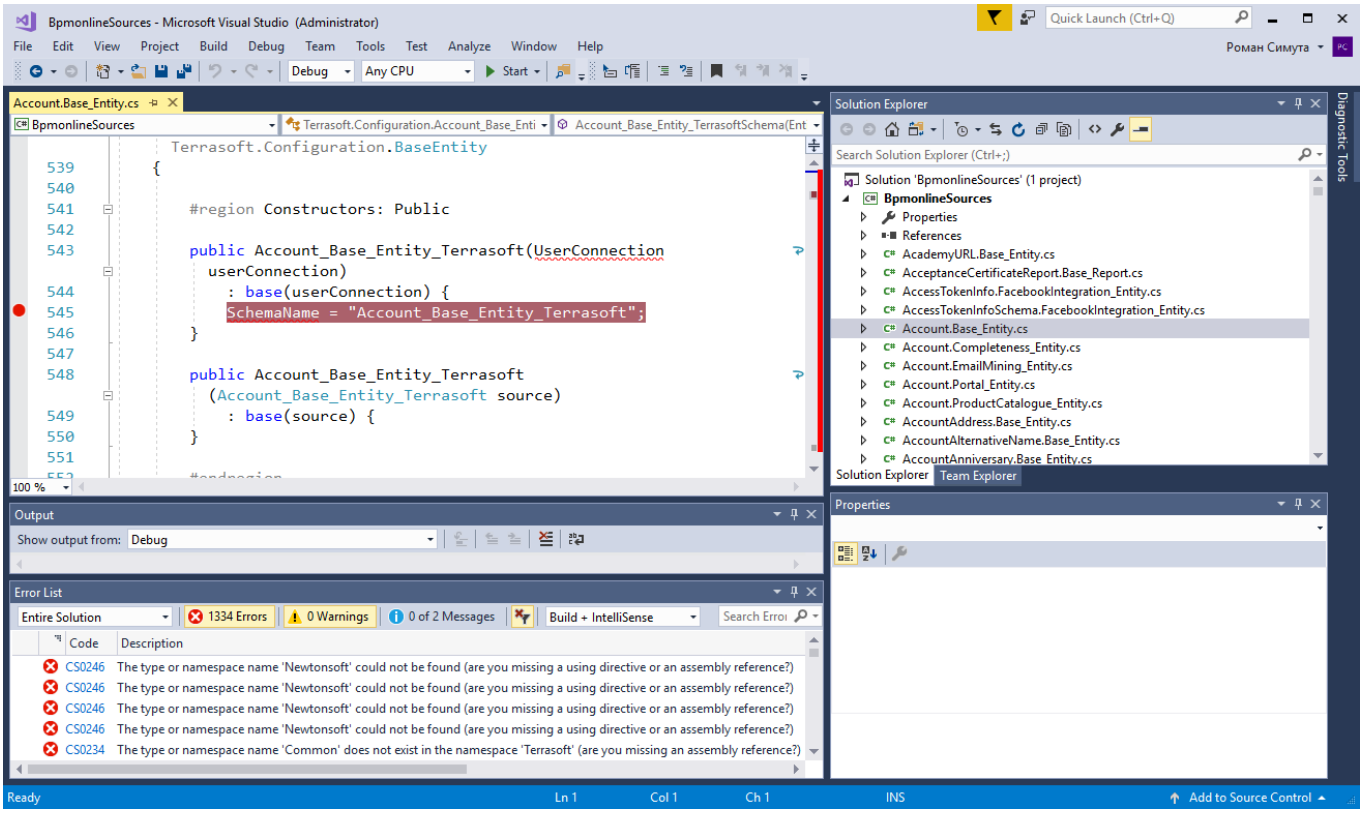
ATTENTION

The name of the working process can be different, depending on the configuration of the IIS server being used. With a regular IIS, the process is w3wp.exe, but with IIS Express, the process name is iiexpress.exe.

By default, the IIS working process is run under the account whose name matches the name of the application pool. To display processes of all users, set [Show processes from all users] checkbox (Fig. 6).

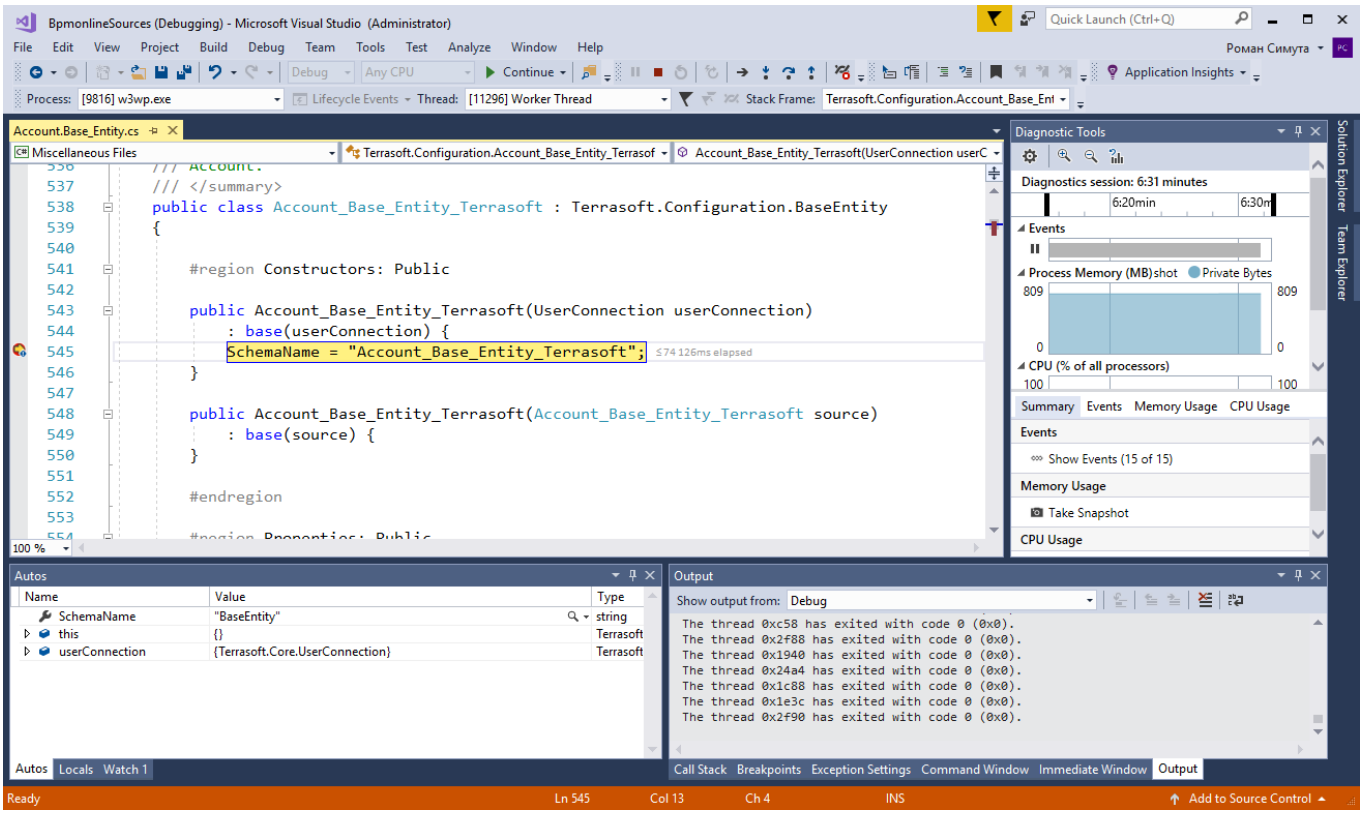
After you attach to the IIS process, you can start debugging. To do this, open the file with the desired source code and set a breakpoint (Fig. 7).

Fig. 7 Breakpoint in the constructor of the [Account] object



As soon as the method with the breakpoint is used, the program will be stopped and you can view the current state of the variables (see Fig. 8).

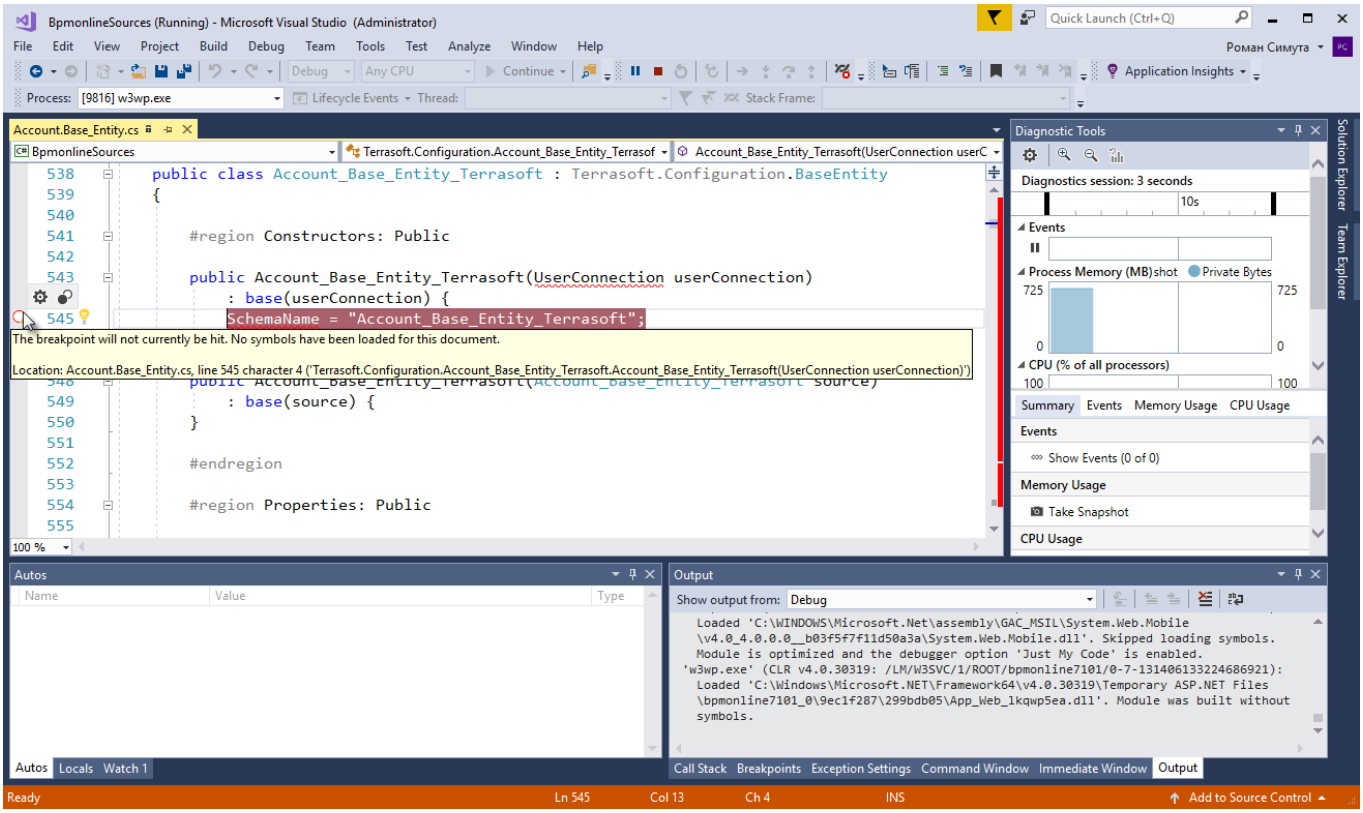
Fig. 8 Interrupting the execution of the program on the breakpoint



Possible debugging issues

After attaching to the IIS process, it is possible that the breakpoint symbol is displayed as a white circle bounded by a red circle. A breakpoint is inactive and the application execution will not be interrupted because of it. When you hover the cursor on the symbol of the inactive breakpoint, a hint will appear and notify you of the problem (Fig. 9).

Fig. 9 Inactive breakpoint Characters not loaded



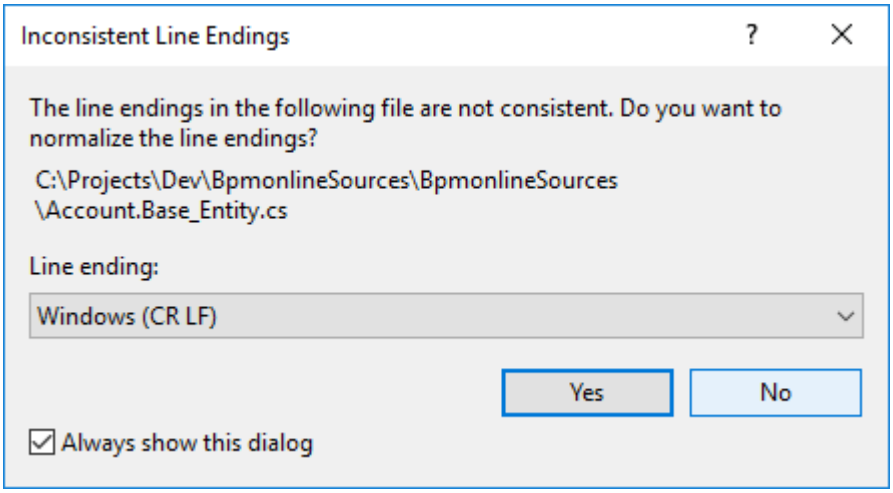
If the hint contains a message that the symbol information was not loaded (Fig. 8), it is necessary to do the following:

1. Finish debugging (*Debug* > *Stop Debugging*).
2. Close the source code file you are debugging.
3. Perform the [Compile all items] action in the [Configuration] section of the application (Fig. 1).
4. While compiling and re-exporting source files, attach to the IIS process again.
5. After the compliation is done, re-open the source code file you are debugging.

> **NOTE**
>
> In some cases, it may be helpful to re-compile without detaching and attaching to IIS.

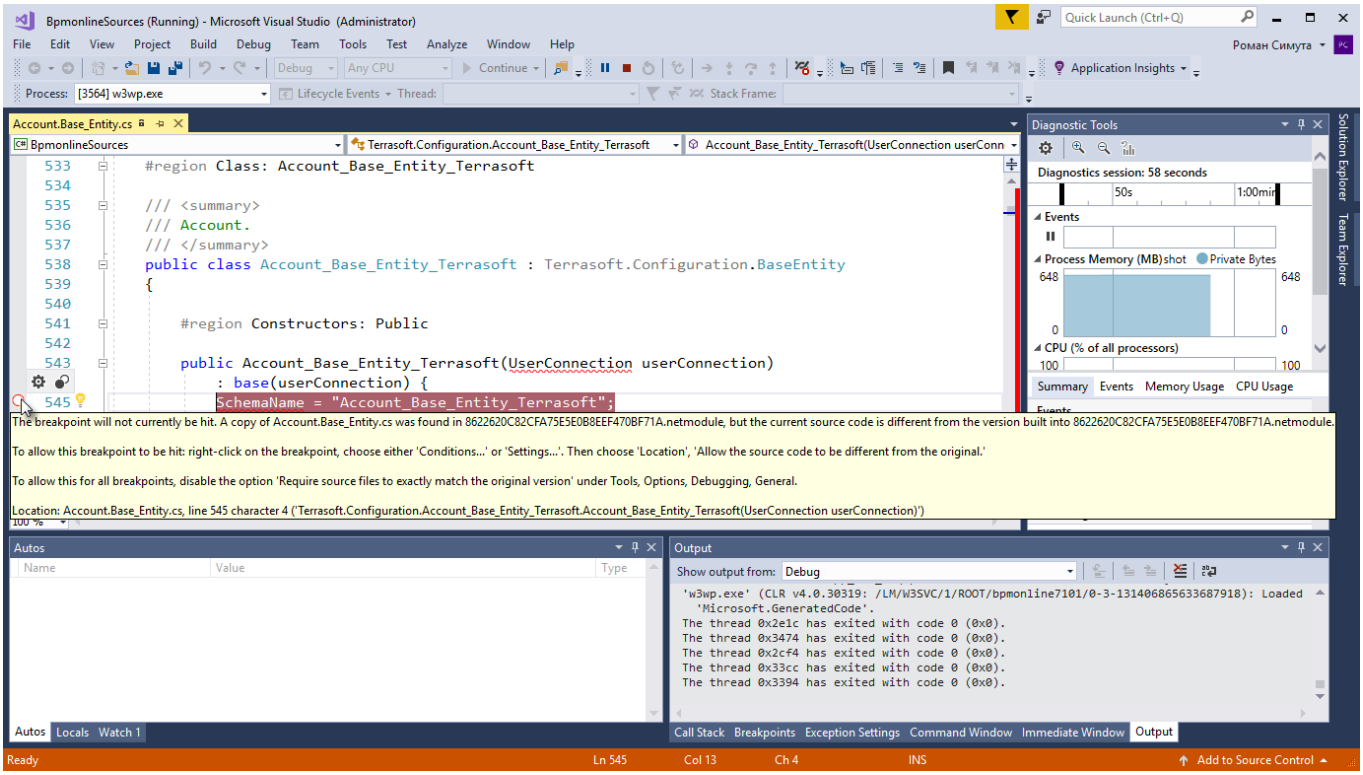
After the file with the source code is reopened, a message about non-uniform end-of-line characters may appear (Fig. 10).

Fig. 10 Message about non-uniforn end-of-line characters



Press the [No] button If you accept the normalization of the characters (the [Yes] button), then the breakpoint may become inactive again. The cause of the problem is displayed in the tooltip - file version mismatch (Fig. 10). The options for solving the problem are also displayed in the tooltip.

Fig. 11 Inactive breakpoint. Version mismatch



Bpm’online development cases

Contents

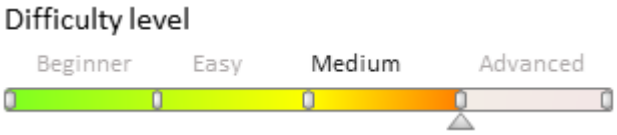
- **Section business logic**
- **Page configuration**
- **Adding details**
- **Business processes**
- **Typical customizations**
- **Analytics**
- **Working with data**
- **Sales products customization**
- **Lending product customization**
- **Marketing product customization**
- **Service products customization**
- **Prediction**

Section business logic

Contents

- **Creating a new section**
- **Adding an action to the list**
- **How to add a button to a section**
- **How to highlight a record in the list in color**
- **Adding quick filter block to a section**

Creating a new section



Introduction

One of typical development tasks in bpm’online is that of adding a new section. Use section wizard to implement this. The section wizard enables you to set up base properties of sections, pages, business rules and DCM cases.

The result of the performed settings will be object and section page schemas, added to the current custom package (table 1.2).

Table 1. Object schemas created by the section wizrad

Naming rules	Purpose	Parent
[Section object name]	Primary section object	Base object (*BaseEntity*)
[Section object name] Folder	Object group. Utility object for the correct groupage of section records. Forms the overall section folder tree.	Base folder (*BaseFolder*)
[Section object name] InFolder	Object in the group.	Base element in the group

	Utility object for the correct operation of section record groupage. Defines links between the section records and folders they belong to.	(*BaseItemInFolder*)
[Section object name] File	Object for the [Attachments] detail.	File (*File*)
[Section object name] Tag	Section tag.	Base tag (*BaseTag*)
[Section object name] InTag	Tag in the section object.	Base tag in the base object (*BaseEntityInTag*)

Table 2. Client schemas created by the section wizard

Naming rules	Purpose	Parent
[Section object name] Section	Section schema	Base section schema (*BaseSectionV2*)
[Section object name] Page	Section edit page schema	*Section edit page base schema* (*BaseModulePageV2*)

Section wizard and the [Custom] package

Section wizard does not only create different schemas but also links data to the current package. However, it is almost impossible to transfer the linked data to another custom package if your current package is the [Custom]. The [Custom] package is not used for committing to the version control system and transferring the changes to other environments. That is why the [Custom] package is not recommended to use as the current custom package. Learn more about the [Custom] package in the "**Package [Custom]**" article.

NOTE

To change the current package, use the [Current package] system setting (*CurrentPackageId*). We recommend you to check this system setting value before you run the section wizard.

Sequence of actions for adding a section

1. Create a section using the section wizard and add the necessary workplace.
2. Add the necessary columns to the section object schema and display them on the record list, on the edit page and in details.

Case description

Add a [Car showroom] workplace to the application. Add a [Trucks] custom section to the created workplace. The "trucks" object schema must contain the following obligatory fields:

- Name – a string.
- Owner – the [Contact] lookup.
- Organization – the [Account] lookup.
- Price – a string.

Case implementation algorithm

1. Create a section using the section wizard and add it to the necessary workplace.

Creating a workplace is covered in the "Workplace setup" article. Indicate the [Name] – *"Car showroom"* for the new workplace and add a group of users who will have access to the created workplace.

Section wizard operation is covered in the "Section wizard" article. Use the first section wizard step to create a new

section. The [Name] column with the“string’ type and columns inherited from the base object will be added to the primary section object. For the initial section setup, populate the following values at the first wizard step:

- [Code] – "UsrTruck";
- [Title] – “Trucks”.

ATTENTION

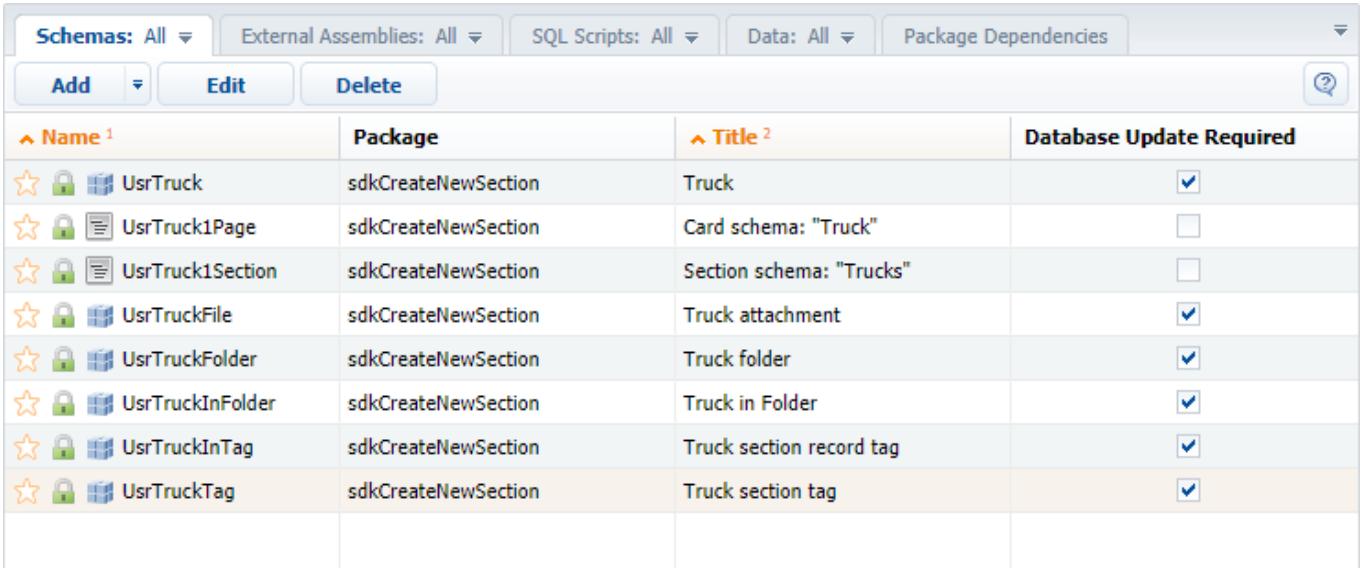
The section object name populated in the [Code] field of the section wizard should not contain prefixes "Base", "Sys" and "Vw". Neither should it contain suffixes "InFolder", "Lcz", "Lookup" and "Settings". Otherwise, you will not be able to set up import from Excel for this object..

NOTE

If you work with bpm’online default settings, you will receive a notification that the value should start with the “Usr” prefix when you populate the [Code] field. The prefix value is indicated in the [Prefix for object name] system setting (SchemaNamePrefix). You can customize the prefix value if needed. We do not recommend to use an empty string as a prefix because of possible name matches with other base configuration elements.

As a result, you will create all schemas that are necessary for section operation in the custom package (fig.1).

Fig. 1. – The [Trucks] section schemas in the custom package



2. Add the necessary columns to the section object schema and display them.

There are two ways to add new columns to the object schema:

1. Create a new column via the section wizard and add it to the edit page immediately. The column will be automatically added to the section primary object schema. Setting up the section record edit page fields is covered in the "How to set up page fields” article.

2. Add a column to the section primary object schema via object designer in the [Configuration] section. Add columns to the page via the section wizard. You can learn more about the [Configuration] section capabilities in the "**The [Configuration] section**” article.

Since the section wizard is involved in any case, using the first way is more convenient.

The [Name] column is created and added to the section pages automatically by the section wizard.

Populate the following properties for the rest of columns:

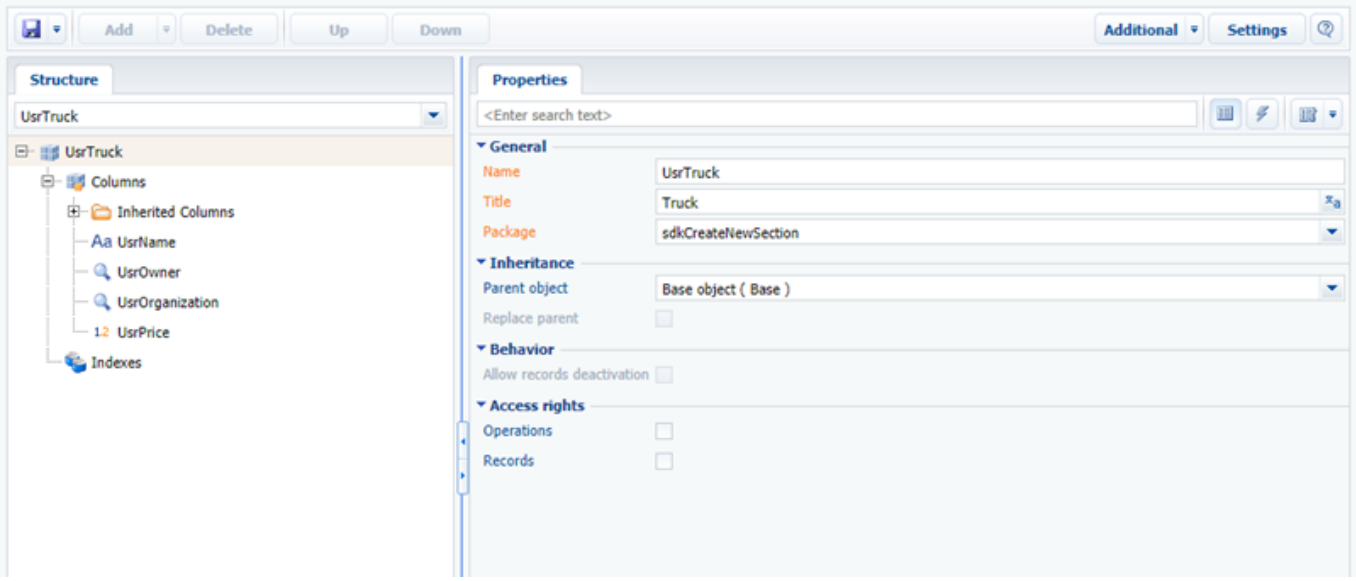
Column	Type	Title	Name in DB
Owner	The [Contact] lookup	Owner	UsrOwner
Organization	The [Account] lookup	Organization	UsrOrganization

Price	Decimal	Price, USD	UsrPrice

Select the [Is required] checkbox for all columns.

New columns will be added to the *UsrTruck* object schema after you save the changes in the section wizard (fig.2) and the corresponding configuration objects will be added to the *diff* modification array of the *UsrTruckPage* edit page schema.

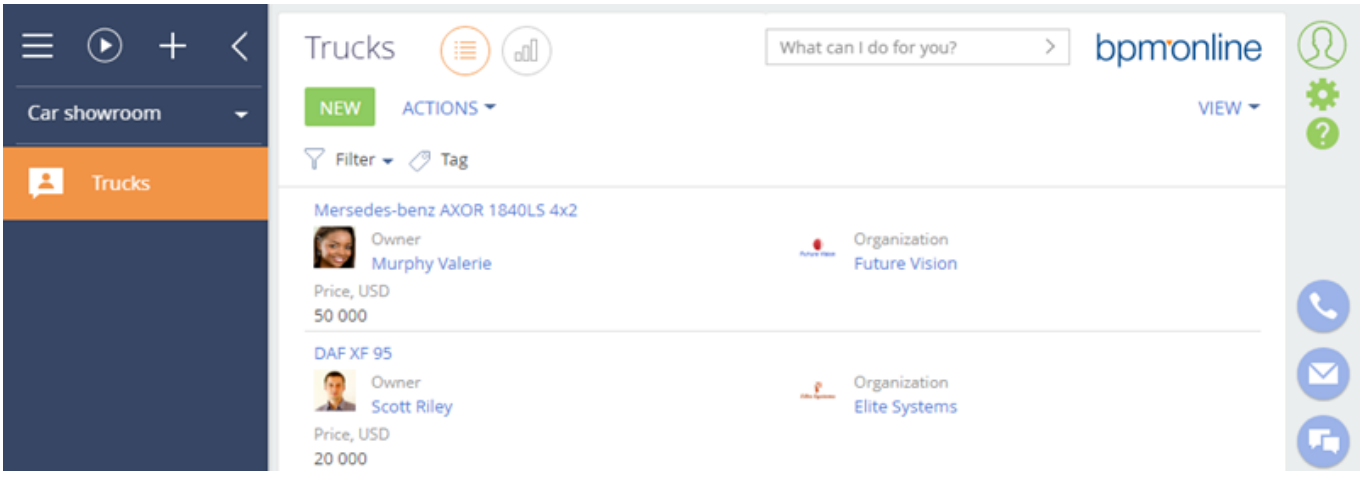
Fig. 2. – The [Trucks] section primary object schema.



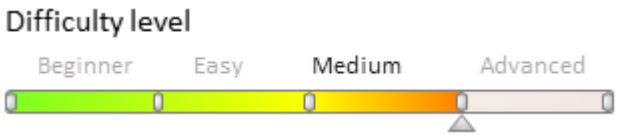
Display the columns on the section record list. Select the [Select fields to display] command in the list’s [View] menu to open the column setup page. Section column setup is covered in the "Setting up columns” article.

After you add new records, the section will look as follows (fig.3):

Fig. 3. – The [Trucks] section in the [Car showroom] workplace.



Adding an action to the list



Overview

Bpm’online has the possibility to set up a list of actions from the [Actions] menu for selected records of a section.

The list of section actions is an instance of the Terrasoft.BaseViewModelCollection class. Each item of the actions list is a view model.

An action is set up in the configuration object where the properties of the actions view model may be set both explicitly and through the use of the base binding mechanism.

The base content of the [Actions] menu for a section page is implemented in the base class of the BaseSectionV2 section.
The list of section actions returns the getSectionActions protected virtual method from the BaseSectionV2 schema.

A separate action is added to the collection by calling the addItem method.

The getButtonMenuItem callback method is passed to it as a parameter. The method creates an instance of the actions view model by the configuration object passed to it as a parameter.

Example 1. — Base implementation of an action addition

```
/**
* This returns the actions collection of the section in the list display mode
* @protected
* @virtual
* @return {Terrasoft.BaseViewModelCollection} Actions collection of the section
*/
getSectionActions: function() {
    // List of actions - Terrasoft.BaseViewModelCollection instance
    var actionMenuItems = this.Ext.create("Terrasoft.BaseViewModelCollection");
    // Adding an action to the collection. The method instantiating the action
    // model instance by the passed configuration object is passed as callback.
    actionMenuItems.addItem(
        this.getButtonMenuItem({
        // Configuration object of setting an action.
        })
    );
    return actionMenuItems;
}
```

Below are the properties of the configuration object of the section action to be passed as a parameter to the getButtonMenuItem method:

Type	a type of the [Actions] menu item	A horizontal line for separating the menu blocks may be added to the action menu using this property. For this purpose, the Terrasoft.MenuSeparator string must be specified as the property value. If no property value is specified, the menu item will be added by default.
Caption	the title of the [Actions] menu item	To set titles, the use of localizable schema strings is recommended.
Click	the action handler method is bound in this property by the method name	
Enabled	a logic property controlling the menu item availability	
Visible	a logical property controlling the menu item visibility	

Procedure for adding a custom action

1. Override the getSectionActions method.
2. Add an action to the actions collection using the addItem method.
3. Pass a configuration object with the added action settings to the getButtonMenuItem callback method.

NOTE

When base sections are replaced in the getSectionActions method of the replacingmodule, the parent implementation of this method must be called first to initialize actions of the parent section.

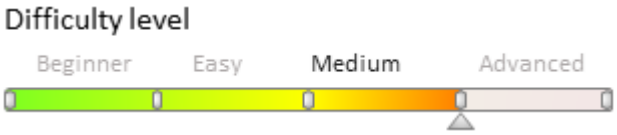
Examples of implementing an action addition

- **How to add a section action: handling the selection of a single record**
- **How to add a section action: handling the selection of several records**
- **Handling the selection of several records. Examples**

See also:

- **Adding an action to the edit page**

How to add a section action: handling the selection of a single record



Case description

Implement an action, which displays the order creation date in the message window for the [Orders] section list. The action is only available for orders with the [In progress] status.

ATTENTION

The [Orders] section is available in bpm’online sales products.

NOTE

You can address the selected record via the *ActiveRow* section view model attribute, which gets the primary column value of the selected record. This value can further be used for getting the values, downloaded into the selected object field list, for instance, from a regular list data collection, which is stored in the *GridData* list view model property.

Source code

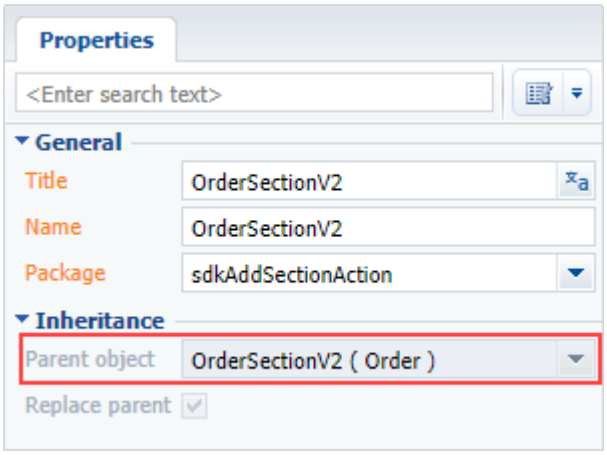
Use this link to download the case implementation package.

Case implementation algorithm

1. Create a replacing page of the [Orders] section in the custom package

Create a replacing client module and specify the *OrderSectionV2* schema as parent object (Fig. 1). The procedure for creating a replacing page is described in the“**Creating a custom client module schema**” article.

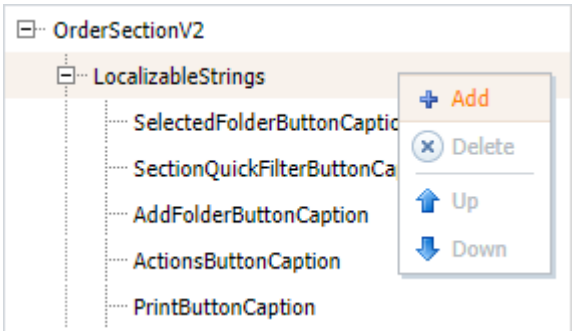
Fig. 1. Properties of the [Orders] section replacing page



2. Add a string with the [Actions] menu title to the localized string collection of the section replacing schema

Create a new localized string (Fig.2).

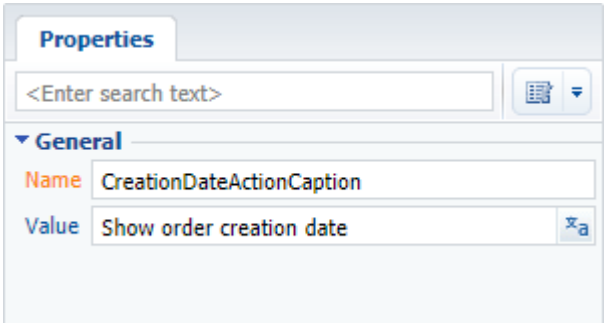
Fig.2 – Adding the localized string to the schema



Populate the following values for the created string (Fig.3):

- [Name] – "CreationDateActionCaption"
- [Value] – "Show order creation date"

Fig. 3. Custom localized string properties



3. Add method implementation to the section view model method collection

- *isRunning()* – verifies if the selected list order has the [In progress] status.
- *isCustomActionEnabled()* – determines if the added menu option is enabled.
- *showOrderInfo()* – the action handler method that displays the selected order estimated completion date in the message window.
- *getSectionActions()* – an overridden parent schema method that gets the section action collection.

The replacing schema source code is as follows:

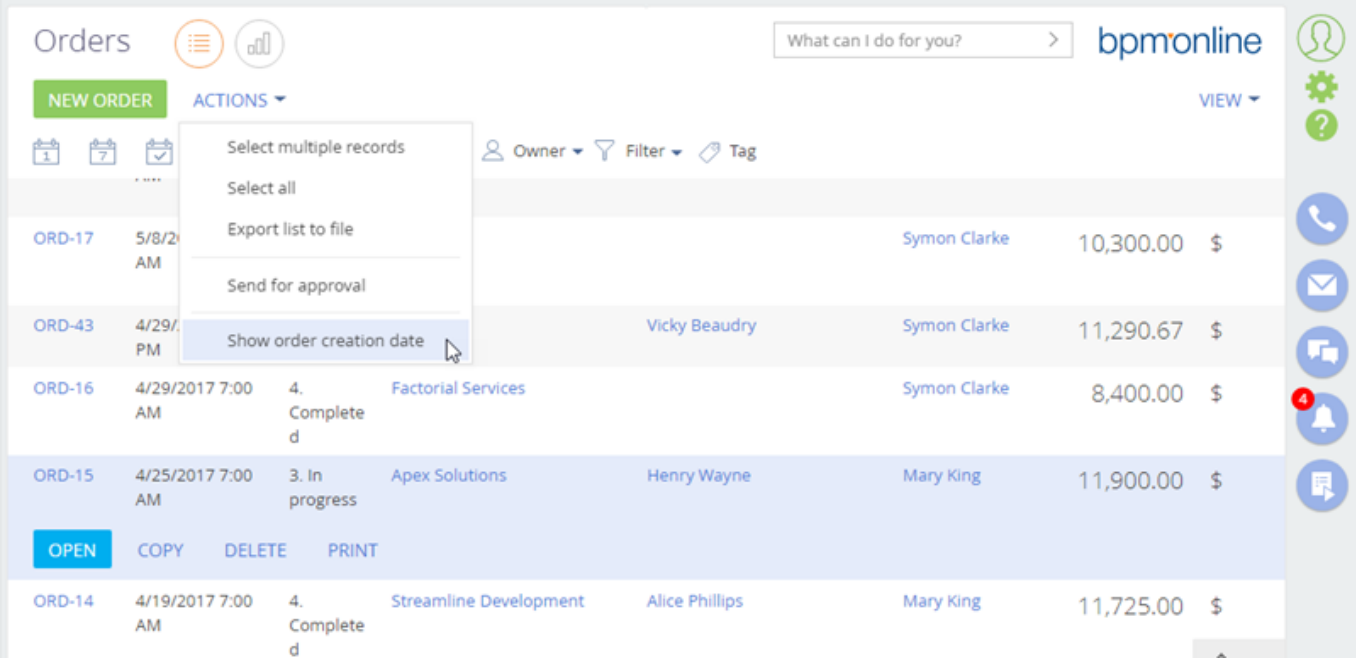
```
define("OrderSectionV2", ["OrderConfigurationConstants"],
```

```
    function(OrderConfigurationConstants) {
    return {
        // Section object schema name.
        entitySchemaName: "Order",
        // Section view model methods.
        methods: {
            // Verifies the order status.
            // activeRowId — the primary column value of the selected list record.
            isRunning: function(activeRowId) {
                // Getting the section list view data collection.
                var gridData = this.get("GridData");
                // Getting the selected order model accroding to the indicated
primary column value.
                var selectedOrder = gridData.get(activeRowId);
                // Getting the model property - the selected order status.
                var selectedOrderStatus = selectedOrder.get("Status");
                // The method gets true if the order status is [In Progress].
Otherwise it gets false.
                return selectedOrderStatus.value ===
OrderConfigurationConstants.Order.OrderStatus.Running;
            },
            // Determines if the menu option is enabled.
            isCustomActionEnabled: function() {
                // Attempt of getting the active (list selected) record identifier.
                var activeRowId = this.get("ActiveRow");
                // If the identifier is determined and the order status is
                // [In Progress], it gets true, otherwise - it gets false.
                return activeRowId ? this.isRunning(activeRowId) : false;
            },
            // Action handler method. Displays the order creation date in the message
window.
            showOrderInfo: function() {
                var activeRowId = this.get("ActiveRow");
                var gridData = this.get("GridData");
                // Getting the order creation date. The column must be added to the
list.
                var dueDate = gridData.get(activeRowId).get("Date");
                // Message window display.
                this.showInformationDialog(dueDate);
            },
            // Overriding the base virtual method that gets the section action
collection.
            getSectionActions: function() {
                // Calling of the method parent implementation for getting the
                //  initiated action collection of the section.
                var actionMenuItems = this.callParent(arguments);
                // Adding a separator line.
                actionMenuItems.addItem(this.getButtonMenuItem({
                    Type: "Terrasoft.MenuSeparator",
                    Caption: ""
                }));
                // Adding a menu option to the section action list.
                actionMenuItems.addItem(this.getButtonMenuItem({
                    // Linking the menu option title to the schema localized string.
                    "Caption": {bindTo:
"Resources.Strings.CreationDateActionCaption"},
                    // Action handler method linking.
                    "Click": {bindTo: "showOrderInfo"},
                    // Linking of the menu option enabling property to the value that
gets the isCustomActionEnabled method.
                    "Enabled": {bindTo: "isCustomActionEnabled"}
                }));
```

```
                // Getting the appended section action collection.
                return actionMenuItems;
            }
        }
    };
});
```

After you save the schema and update the application page with clearing the browser cache, a new action appears in the [Orders] section. It will be active when you select an order with the [In progress] status (Fig.4).

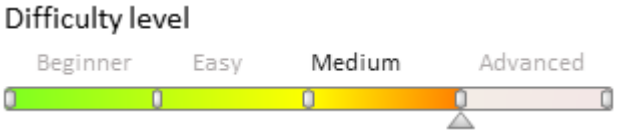
Fig. 4. Case result



See also

- **Adding an action to the list**
- **How to add a section action: handling the selection of several records**
- **Handling the selection of several records. Examples**

How to add a section action: handling the selection of several records



Introduction

The section single record mode is used by default. To select multiple active list records use the [Select multiple records] option in the [Actions] button menu. The list visual view will change – you will see record selection elements appear. To cancel the multiple record mode, click [Cancel multiple selection] in the [Actions] button menu.

Case description

Implement an action, which displays account names of several selected list orders in the message window for the [Orders] section list.

ATTENTION

The [Orders] section is available in bpm’online sales products.

NOTE

The primary column values of the selected records are stored in the *SelectedRows* property of the section view model. These values can further be used for getting the values, downloaded into the selected object field list, for instance, from a regular list data collection, which is stored in the *GridData* list view model property.

Source code

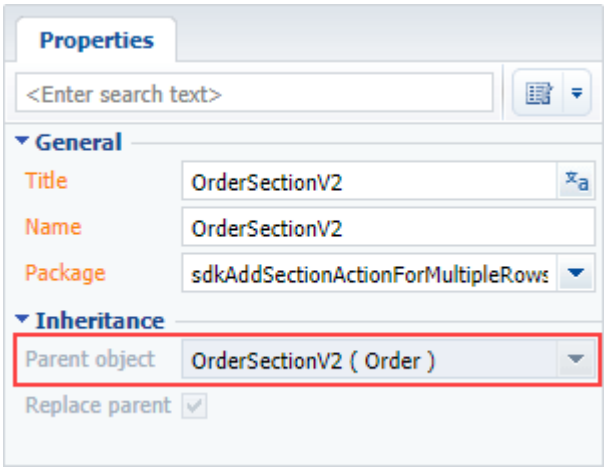
Use this link to download the case implementation package.

Case implementation algorithm

1. Create a replacing page of the [Orders] section in the custom package

Create a replacing client module and specify the *OrderSectionV2* schema as parent object (Fig. 1). The procedure for creating a replacing page is described in the “**Creating a custom client module schema**” article.

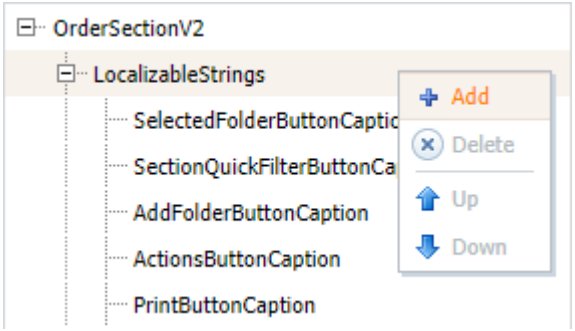
Fig. 1. Properties of the [Orders] section replacing page



2. Add a string with the [Actions] menu title to the localized string collection of the section replacing schema

Create a new localized string (Fig.2).

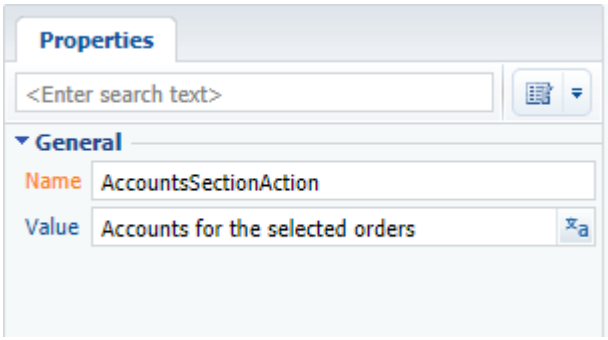
Fig. 2 – Adding the localized string to the schema



Populate the following values for the created string (Fig.3):

- [Name] – “AccountSectionAction”
- [Value] – “Accounts for the selected orders”

Fig. 3. Custom localized string properties



3. Add method implementation to the section view model method collection

- *isCustomActionEnabled()* – determines if the added menu option is enabled.
- *showOrderInfo()* – the action handler method that displays the selected order account list in the message window.
- *getSectionActions()* – an overridden parent schema method that gets the section action collection.

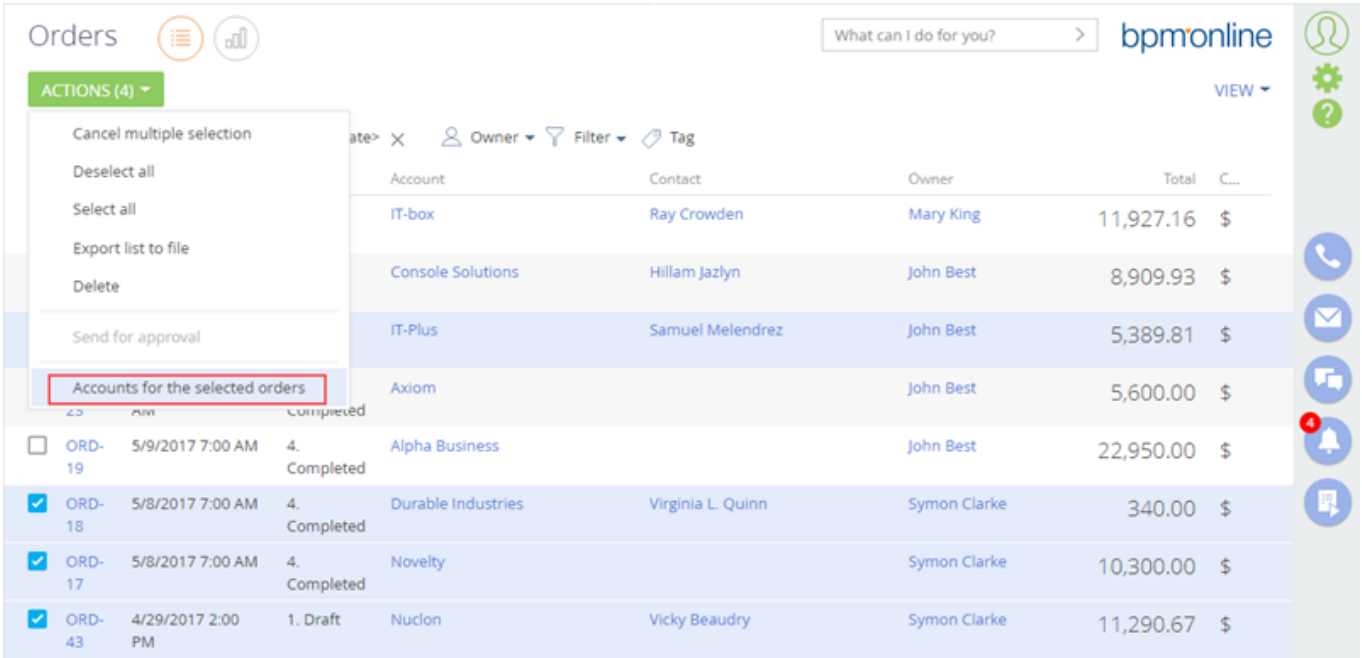
The replacing schema source code is as follows:

```
define("OrderSectionV2", ["OrderConfigurationConstants"],
    function(OrderConfigurationConstants) {
    return {
        // Section schema name.
        entitySchemaName: "Order",
        // Section view model methods.
        methods: {
            // Determines if the menu option is enabled.
            isCustomActionEnabled: function() {
                // Attempt of getting the selected record identifier array.
                var selectedRows = this.get("SelectedRows");
                // If the array contains any elements (at least one list record is
selected),
                // it gets true, otherwise - it gets false.
                return selectedRows ? (selectedRows.length > 0) : false;
            },
            // Action handler method. Displays the account list in the message
window.
            showOrdersInfo: function() {
                // Getting the selected record identifier array.
                var selectedRows = this.get("SelectedRows");
                // Getting the list record data collection.
                var gridData = this.get("GridData");
                // Variable for stoarge of the selected order object model.
                var selectedOrder = null;
                // Variable for stoarge of the selected order account name.
                var selectedOrderAccount = "";
                // Variable for the message window text.
                var infoText = "";
                // Handling of the selected section record identifier array.
                selectedRows.forEach(function(selectedRowId) {
                    // Getting the selected order object model.
                    selectedOrder = gridData.get(selectedRowId);
                    // Getting the selected order account name. The column must be
added to the list.
                    selectedOrderAccount = selectedOrder.get("Account").displayValue;
                    // Adding the account name to the message window text.
                    infoText += "\n" + selectedOrderAccount;
                });
                // Message window display.
                this.showInformationDialog(infoText);
```

```
            },
            // Overriding the base virtual method that gets the section action
collection.
            getSectionActions: function() {
                // Calling of the method parent implementation for getting the
                // initiated action collection of the section.
                var actionMenuItems = this.callParent(arguments);
                // Adding a separator line.
                actionMenuItems.addItem(this.getButtonMenuItem({
                    Type: "Terrasoft.MenuSeparator",
                    Caption: ""
                }));
                // Adding a menu option to the section action list.
                actionMenuItems.addItem(this.getButtonMenuItem({
                    // Linking the menu option title to the schema localized string.
                    "Caption": {bindTo: "Resources.Strings.AccountsSectionAction"},
                    // Action handler method linking.
                    "Click": {bindTo: "showOrdersInfo"},
                    // Linking of the menu option enabling property to the value that
gets the isCustomActionEnabled method.
                    "Enabled": {bindTo: "isCustomActionEnabled"}
                }));
                // Getting the appended section action collection.
                return actionMenuItems;
            }
        }
    };
});
```

After you save the schema and update the application page with clearing the browser cache, a new action appears in the [Orders] section. It will be active when you select orders in the multiple record selection mode (Fig.4).

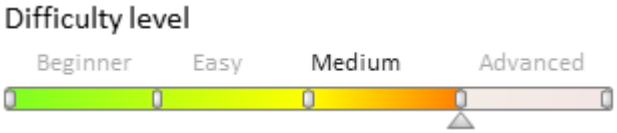
Fig. 4. Case result



See also

- **Adding an action to the list**
- **How to add a section action: handling the selection of a single record**
- **Handling the selection of several records. Examples**

Handling the selection of several records. Examples



Introduction

Before you start implementing cases it is recommended to study the "**How to add a section action: handling the selection of several records**” article.

Example

Case description

Implement action for the [Activities] section list that will set the [Completed] status for several selected list activities.

Source code

You can download the package with case implementation using the following link.

Case implementation algorithm

1. Create a replacing page of the [Activities] section in the custom package

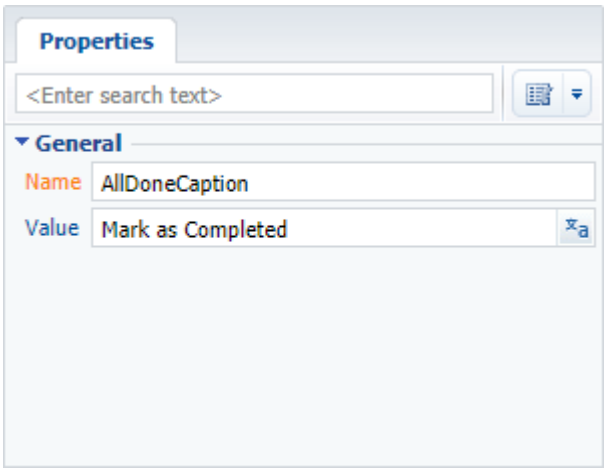
The procedure for creating a replacing page is described in the“**Creating a custom client module schema**” article.

2. Add a string with the [Actions] menu title to the localized string collection of the section replacing schema

Populate the following values for the created string (Fig.1):

- [Name] – "AllDoneCaption";
- [Value] – "Mark as "Completed"".

Fig. 1. Properties of the custom localizable string



3. Add the implementation of the following methods to the method collection of the section view model

- *isCustomActionEnabled()* – the method that determines if the added menu option is enabled.
- *setAllDone()* – the action handler method that sets the [Completed] status for several selected list

activities.

- *getSectionActions()* – an overridden parent schema method that gets the section action collection.

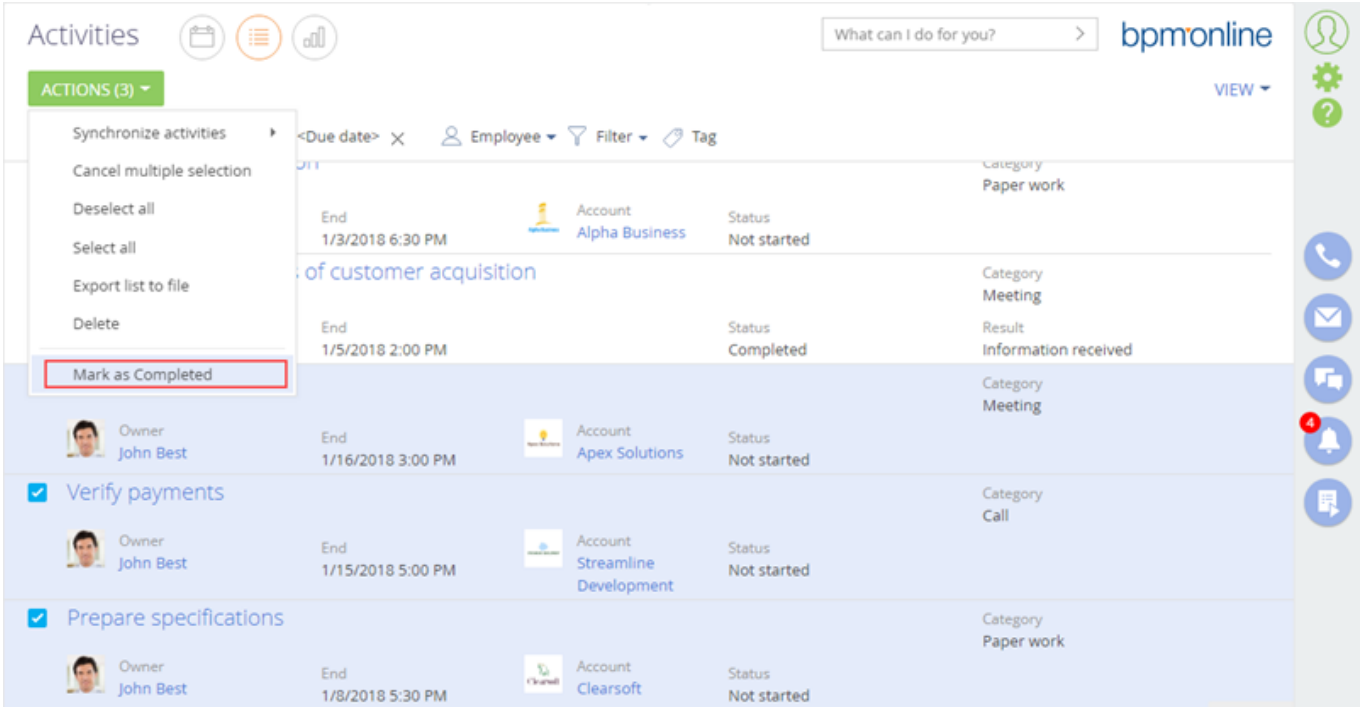
The replacing schema source code is as follows:

```
define("ActivitySectionV2", ["ConfigurationConstants"],
    function(ConfigurationConstants) {
        return {
            // Section schema name.
            entitySchemaName: "Activity",
            // Section view model methods.
            methods: {
                // Defines if the menu option is enabled.
                isCustomActionEnabled: function() {
                    // Attempt to receive the selected record indentifier array.
                    var selectedRows = this.get("SelectedRows");
                    // If the array contains some elements (at least one of the
records is selected from the list),
                    // it returns true, otherwise – false.
                    return selectedRows ? (selectedRows.length > 0) : false;
                },
                // Action handler method. Sets the [Completed] status for the
selected records.
                setAllDone: function() {
                    // Receiving the selected record indentifier array.
                    var selectedRows = this.get("SelectedRows");
                    // The procession starts if at least one record is selected.
                    if (selectedRows.length > 0) {
                        // Creation of the batch query class instance.
                        var batchQuery = this.Ext.create("Terrasoft.BatchQuery");
                        // Update of each selected record.
                        selectedRows.forEach(function(selectedRowId) {
                            // Creation of the UpdateQuery class instance with the
Activity root schema.
                            var update = this.Ext.create("Terrasoft.UpdateQuery", {
                                rootSchemaName: "Activity"
                            });
                            // Applying filter to determine the record for update.
                            update.enablePrimaryColumnFilter(selectedRowId);
                            // The "Success" value is set to the Status column via
                            // the ConfigurationConstants.Activity.Status.Done.
                            update.setParameterValue("Status",
ConfigurationConstants.Activity.Status.Done,

this.Terrasoft.DataValueType.GUID);
                            // Adding a record update query to the batch query.
                            batchQuery.add(update);
                        }, this);
                        // Batch query to the server.
                        batchQuery.execute(function() {
                            // Record list update.
                            this.reloadGridData();
                        }, this);
                    }
                },
                // Overriding the base virtual method, returning the section action
collection.
                getSectionActions: function() {
                    // Calling of the parent method implementation,
                    // returning the initialized section action collection.
                    var actionMenuItems = this.callParent(arguments);
                    // Adding separator line.
```

```
                        actionMenuItems.addItem(this.getButtonMenuItem({
                            Type: "Terrasoft.MenuSeparator",
                            Caption: ""
                        }));
                        // Adding a menu option to the section action list.
                        actionMenuItems.addItem(this.getButtonMenuItem({
                            // Binding the menu option title to the localized schema
string.
                            "Caption": { bindTo: "Resources.Strings.AllDoneCaption" },
                            // Binding of the action handler method.
                            "Click": { bindTo: "setAllDone" },
                            // Binding the menu option enable property to the value that
returns the isCustomActionEnabled method.
                            "Enabled": { bindTo: "isCustomActionEnabled" }
                        }));
                        // Returning of the added section action collection.
                        return actionMenuItems;
                    }
                }
            };
    });
```

After saving the schema and updating the app page with clearing the cache you will be able to apply the [Completed] status to several selected activities in the [Activities] section by using the new [Mark as Completed] action.

Fig. 2. Case result demonstration



Example 2

Case description

Implement action for the [Activities] section list that will call the record owner selection window and set the selected value for several selected list activities.

Source code

You can download the package with case implementation using the following link.

Case implementation algorithm

1. Create a replacing page of the [Activities] section in the custom package

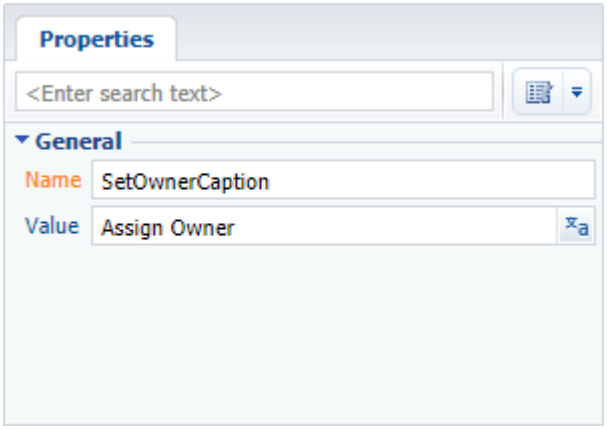
The procedure for creating a replacing page is described in the"**Creating a custom client module schema**" article.

2. Add a string with the [Actions] menu title to the localized string collection of the section replacing schema

Populate the following values for the created string (Fig.3):

- [Name] – "SetOwnerCaption";
- [Value] – "Assign Owner".

Fig. 3. Properties of the custom localizable string



3. Add the implementation of the following methods to the method collection of the section view model

- *isCustomActionEnabled()* – the method that determines if the added menu option is enabled.
- *setOwner()* – the action handler method that calls opening of the [Contacts] lookup.
- *lookupCallback()* – the callback-method that sets the lookup selected contact as the owner for the selected list records.
- *getSectionActions()* – an overridden parent schema method that gets the section action collection.

The replacing schema source code is as follows:

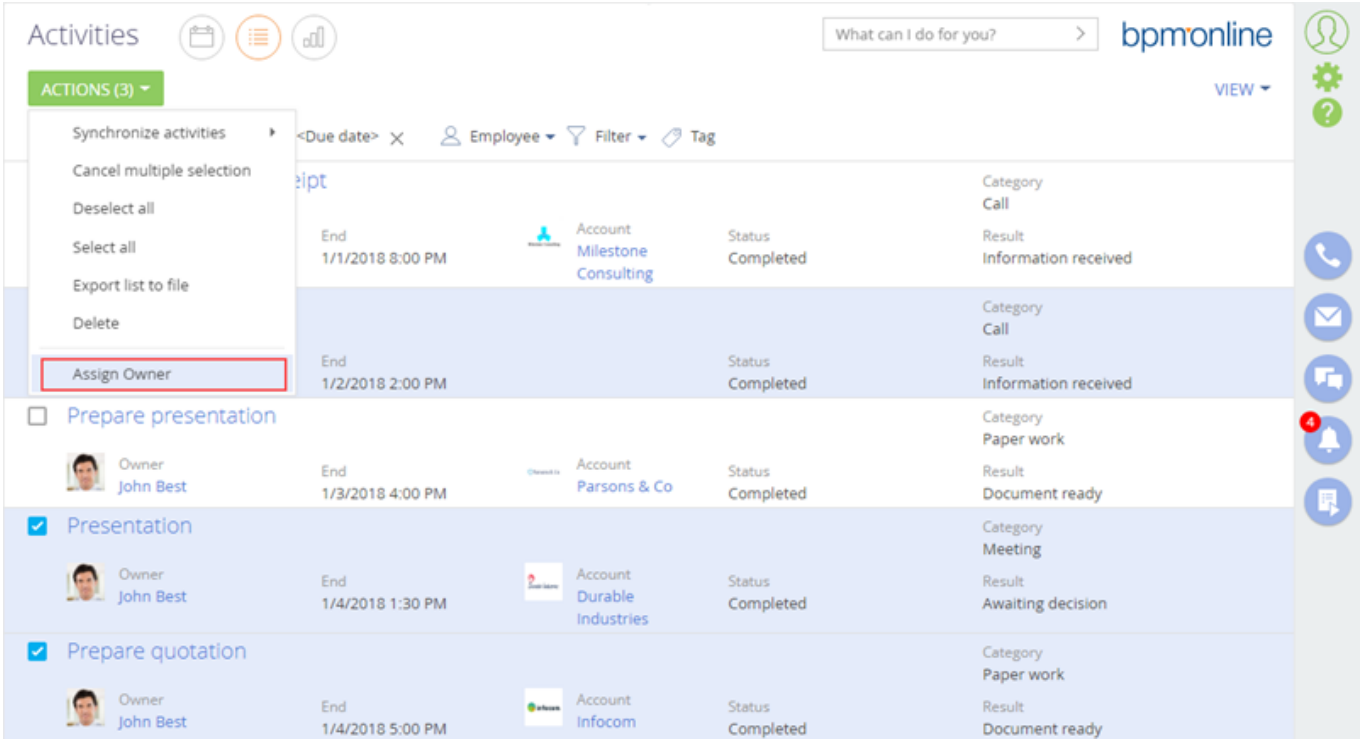
```
define("ActivitySectionV2", ["ConfigurationConstants"],
    function(ConfigurationConstants) {
        return {
            // Section schema name.
            entitySchemaName: "Activity",
            // Section view model methods.
            methods: {
                // Defines if the menu option is enabled.
                isCustomActionEnabled: function() {
                    // Attempt to receive the selected record indentifier array
                    var selectedRows = this.get("SelectedRows");
                    // If the array contains some elements (at least one of the
records is selected from the list),
                    // it returns true, otherwise – false.
                    return selectedRows ? (selectedRows.length > 0) : false;
                },
                // Action handler method. Opens the [Contacts] lookup.
                setOwner: function() {
                    // Defining the lookup configuration.
                    var config = {
```

```
                        // The [Contact] Schema.
                        entitySchemaName: "Contact",
                        // Multiple selection is disabled.
                        multiSelect: false,
                        // The displayed column — [Name].
                        columns: ["Name"]
                    };
                    // Opening of the lookup with certain configuration and call-back
function that is triggered
                    // after you click [Select].
                    this.openLookup(config, this.lookupCallback, this);
                },
                //  Sets the lookup selected contact as the owner
                // for the selected list records.
                lookupCallback: function(args) {
                    // The selected lookup record identifier.
                    var activeRowId;
                    // Receiving of the lookup selected records.
                    var lookupSelectedRows = args.selectedRows.getItems();
                    if (lookupSelectedRows && lookupSelectedRows.length > 0) {
                        // Receiving of the first lookup selected record Id.
                        activeRowId = lookupSelectedRows[0].Id;
                    }
                    // Receiving of the selected record identifier array.
                    var selectedRows = this.get("SelectedRows");
                    // The procession starts if at least one record is selected from
the list and the owner is selected
                    // in the lookup.
                    if ((selectedRows.length > 0) && activeRowId) {
                        // Creation of the batch query class instance.
                        var batchQuery = this.Ext.create("Terrasoft.BatchQuery");
                        // Update of each selected record.
                        selectedRows.forEach(function(selectedRowId) {
                            // Creation of the UpdateQuery class instance with the
Activity root schema.
                            var update = this.Ext.create("Terrasoft.UpdateQuery", {
                                rootSchemaName: "Activity"
                            });
                            // Applying filter to determine the record for update.
                            update.enablePrimaryColumnFilter(selectedRowId);
                            // The [Owner] column is populated with the value that
equals to
                            // the lookup selected contact id.
                            update.setParameterValue("Owner", activeRowId,
this.Terrasoft.DataValueType.GUID);
                            // Adding a record update query to the batch query.
                            batchQuery.add(update);
                        }, this);
                        // Batch query to the server.
                        batchQuery.execute(function() {
                            // Record list update.
                            this.reloadGridData();
                        }, this);
                    }
                },
                // Overriding the base virtual method, returning the section action
collection.
                getSectionActions: function() {
                    // Calling of the parent method implementation,
                    // returning the initialized section action collection.
                    var actionMenuItems = this.callParent(arguments);
                    // Adding separator line.
```

```
                        actionMenuItems.addItem(this.getButtonMenuItem({
                            Type: "Terrasoft.MenuSeparator",
                            Caption: ""
                        }));
                        // Adding a menu option to the section action list.
                        actionMenuItems.addItem(this.getButtonMenuItem({
                            // Binding the menu option title to the localized schema
string.
                            "Caption": { bindTo: "Resources.Strings.SetOwnerCaption" },
                            // Binding of the action handler method.
                            "Click": { bindTo: "setOwner" },
                            // Binding the menu option enable property to the value that
returns the isCustomActionEnabled method.
                            "Enabled": { bindTo: "isCustomActionEnabled" }
                        }));
                        // Returning of the added section action collection.
                        return actionMenuItems;
                    }
                }
            };
    });
```

After saving the schema and updating the app page with clearing the cache you will be able to change the owner of several selected activities in the [Activities] section by using the new [Assign Owner] action.

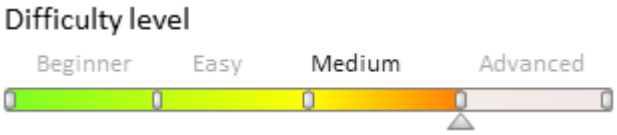
Fig. 4. Case result demonstration



See also

- **Adding an action to the list**
- **How to add a section action: handling the selection of a single record**
- **How to add a section action: handling the selection of several records**

How to add a button to a section



Introduction

During the process of section customization, you may need to create a custom action and add the appropriate button to the section. For this, use the container of action buttons (ActionButtonsContainer) with the the new record button and the button with drop-down action list. More information can be found in the “**Section actions**” article.

To add a custom button to the view model, you need to change the following properties:

- *diff* array of configuration objects. Add a configuration object for setting up location of the component on the edit page.
- *methods* collection. Add an implementation of the handler method, that will be called on button click, and other auxiliary methods necessary for the work of the control. These can be methods that will regulate the visibility or availability of the control, depending on the conditions.

More information about button visualization can be found in the “**Adding a button to the edit page**” article.

Case description

Add a button to the [Accounts] section. The button should open the edit page of the primary contact of the account selected in the list.

NOTE

Access to the selected record is performed through the *ActiveRow* attribute of the section view model, that returns the value of the primary column of selected record. This value can be used to get values loaded into the list of the fields of selected object, for example, from the list data collection that is stored in the *GridData* property of the list view model.

Source code

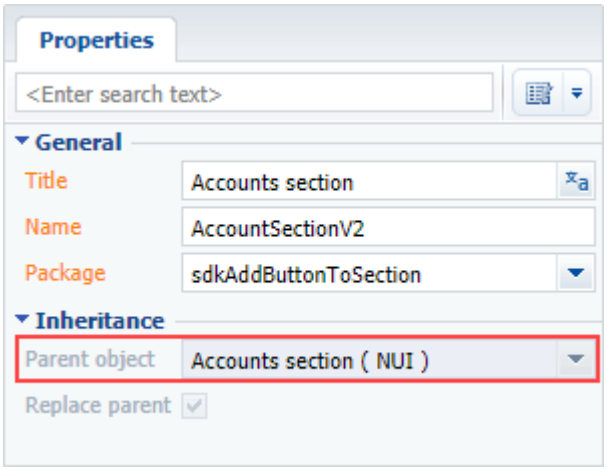
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Create a replacing edit page of the [Accounts] section

Create a replacing client module and specify the [Accounts section] schema as parent object (Fig. 1). The procedure for creating a replacing page is covered in the “**Creating a custom client module schema**” article.

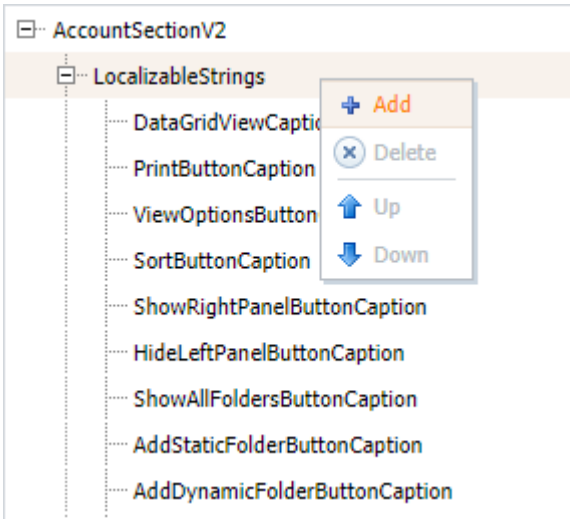
Fig. 1. Properties of the [Accounts] section replacing schema



2. Add a string with the button title to the collection of localizable strings of the replacing schema

Create a new localizable string (Fig. 2).

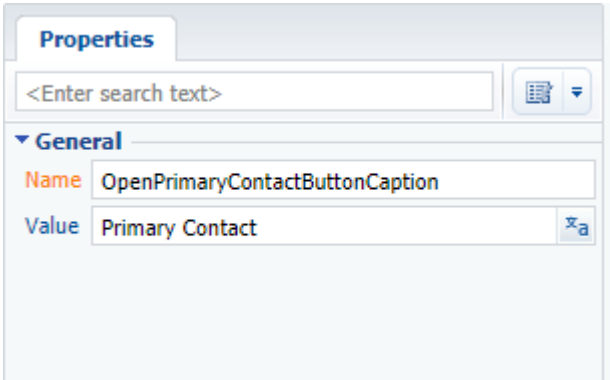
Fig.2. Adding localizable string to the schema



For the created string specify (Fig. 3):

- [Name] – “OpenPrimaryContactButtonCaption"
- [Value] – "Primary Contact".

Fig. 3. Properties of the custom localizable string



3. Add the implementation of the following methods to the method collection of the section view model

- *isAccountPrimaryContactSet()* – checks if the [Primary contact] field is filled.
- *onOpenPrimaryContactClick()* – button click handler method. Opens the edit page of the primary contact.

4. Add a configuration object with the settings determining the button position in the *diff* array

Add an object with the settings determining the button position on the page in the *diff* array.

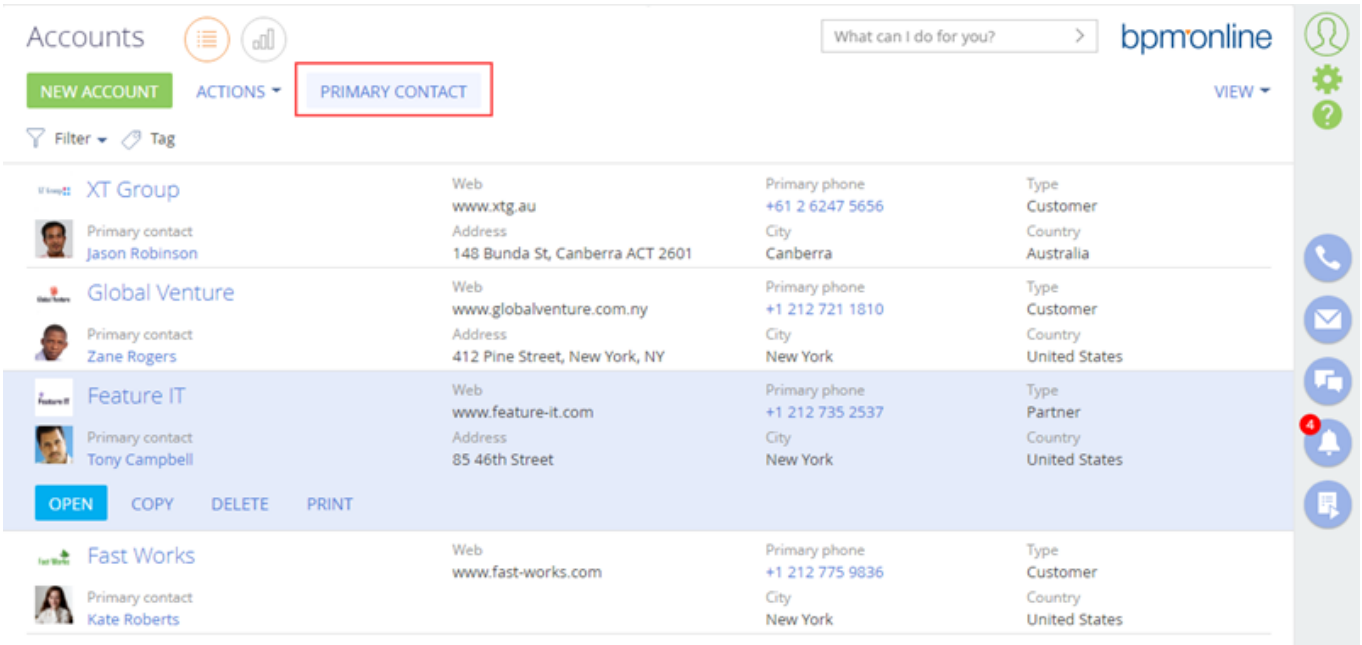
The replacing schema source code is as follows:

```
define("AccountSectionV2", [], function() {
    return {
        // Name of the section object schema.
        entitySchemaName: "Account",
        // Method of the section view model.
        methods: {
            // Button click handler method.
            onOpenPrimaryContactClick: function() {
                // Getting the id of the selected record.
                var activeRow = this.get("ActiveRow");
                if (!activeRow) {
                    return;
                }
                // Defining the id of the primary contact.
                var primaryId =
this.get("GridData").get(activeRow).get("PrimaryContact").value;
                if (!primaryId) {
                    return;
                }
                // Creation of the address string.
                var requestUrl = "CardModuleV2/ContactPageV2/edit/" + primaryId;
                // Publication a message about updating the navigation history of
pages and
                // opening to the primary contact edit page.
                this.sandbox.publish("PushHistoryState", {
                    hash: requestUrl
                });
            },
            // Checks if the [Primary Contact] field of the selected item is filled.
            isAccountPrimaryContactSet: function() {
                var activeRow = this.get("ActiveRow");
                if (!activeRow) {
                    return false;
                }
                var pc = this.get("GridData").get(activeRow).get("PrimaryContact");
                return (pc || pc !== "") ? true : false;
            }
        },
        //Display button in the section.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding a custom button to a section.
            {
                // The operation of adding a component to the page is in progress..
                "operation": "insert",
                // The meta name of the parent container to which the button is
added.
                "parentName": "ActionButtonsContainer",
                // The button is added to the parent component's collection.
                "propertyName": "items",
                // The meta-name of the button to be added.
                "name": "MainContactSectionButton",
                // Properties passed to the component's constructor.
```

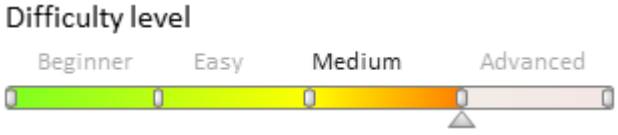
```
                "values": {
                    // The type of the component to add is the button.
                    itemType: Terrasoft.ViewItemType.BUTTON,
                    // Bind the button header to the localizable string of the
schema.
                    caption: { bindTo:
"Resources.Strings.OpenPrimaryContactButtonCaption" },
                    // Bind the button click handler method.
                    click: { bindTo: "onOpenPrimaryContactClick" },
                    // Binding the button availability property.
                    enabled: { bindTo: "isAccountPrimaryContactSet" },
                    // Setting the location of the button.
                    "layout": {
                        "column": 1,
                        "row": 6,
                        "colSpan": 1
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

After saving the schema, clearing the browser cache and updating the application page, the [Primary Contact] button will be displayed in the [Accounts] section. The button will be enabled after selecting the account with the specified primary contact (Fig. 4).

Fig. 4. Case result



How to highlight a record in the list in color



Introduction

Bpm'online enables you to configure the layout of the record list by highlighting some records when a specific condition is met. This setting helps to highlight the records that require special attention.

The display of the list record is controlled by the *customStyle* property of the record list.

The *customStyle* property is an object, which properties are similar to CSS properties and generate style of displaying a list record. Example:

```
item.customStyle = {
    // The text color is white.
    "color": "white",
    // The background color is orange.
    "background": "orange"
};
```

Follow these steps to configure the display of individual list records:

1. Override the *prepareResponseCollectionItem(item),* base method in the replacing schema of the section. This method modifies a data string before loading it to the list.
2. In the *prepareResponseCollectionItem(item)* method, implement the assigning of specific value to the *customStyle* property for the necessary list records.

Case description

For the [Orders] section, implement highlighting the list records with the [In progress] status.

> ATTENTION
>
> The [Orders] section is available in bpm'online sales products.

Source code

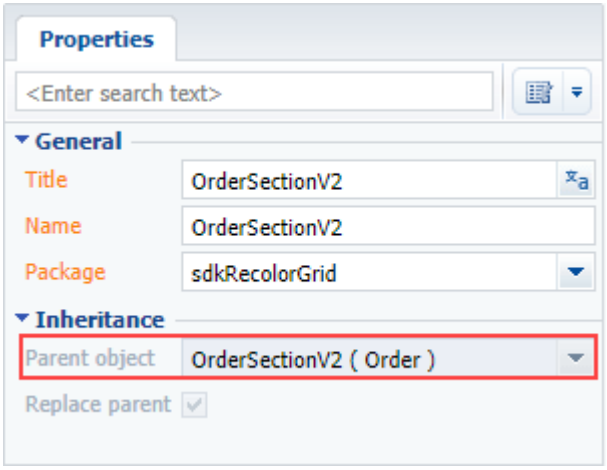
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Create a replacing page of the [Orders] section in the custom package

Create a replacing client module and specify the *OrderSectionV2* schema as parent object (Fig. 1). The procedure for creating a replacing page is covered in the "**Creating a custom client module schema**" article.

Fig. 1. Properties of the [Orders] section replacing page



2. Override the prepareResponseCollectionItem method

Add the *prepareResponseCollectionItem()* to the method collection of the created schema. The method overrides

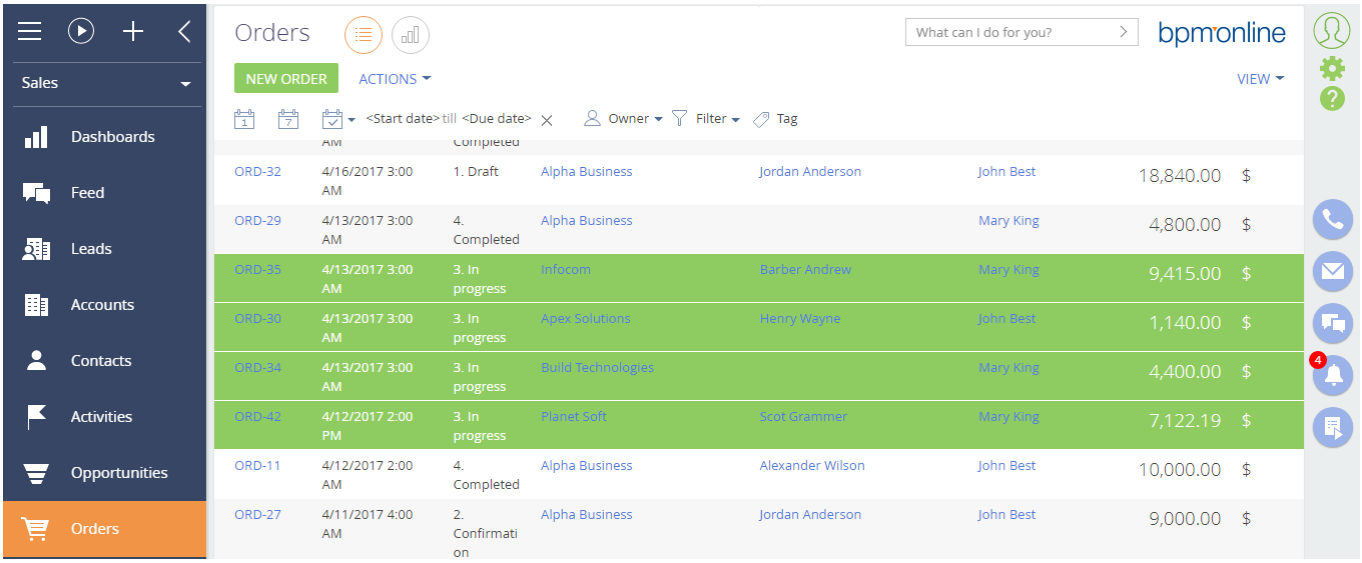
the base method, modifies the data string before uploading it to the list, and adds custom styles to the specific list records.

The replacing schema source code is as follows:

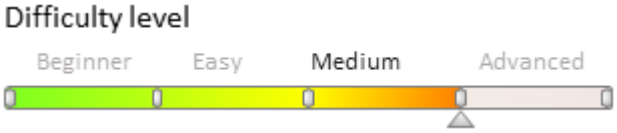
```
define("OrderSectionV2", ["OrderConfigurationConstants"],
function(OrderConfigurationConstants) {
    return {
        // The name of the section scheme.
        entitySchemaName: "Order",
        // Methods of the section representation model.
        methods: {
            // Override the base method, which modifies the data string before it is
loaded into the list.
            prepareResponseCollectionItem: function(item) {
                // Calling the base method.
                this.callParent(arguments);
                item.customStyle = null;
                // Determining the order status.
                var running = item.get("Status");
                //If the status of the order is "In progress", the record style
changes.
                if (running.value ===
OrderConfigurationConstants.Order.OrderStatus.Running) {
                    item.customStyle = {
                        // The text color is white.
                        "color": "white",
                        // The background color is green.
                        "background": "8ecb60"
                    };
                }
            }
        }
    };
});
```

After saving the schema, clearing the browser cache and updating the application page, the orders in the [Orders] section with the [In progress] status will be highlighted in green (Fig. 2).

Fig. 2. Case result



Adding quick filter block to a section



Introduction

Filters are designed to search and filter records in sections. In bpm'online quick, standard and advanced filters and folders are provided. More information can be found in the "**Filters**” article.

To add a block of quick filters to the section, override the *initFixedFiltersConfig()* method in the replacement schema, create the *fixedFilterConfig* configuration object in this method with the following properties:

- *entitySchema* – object schema.
- *filters* – array of added filters.

Assign a reference to the created configuration object to the *fixedFiltersConfig* view model attribute:

```
this.set("FixedFilterConfig", fixedFilterConfig);
```

Case description

Add a block of quick filters to the [Contracts] section. Filter by the contract start date and owner.

Source code

You can download the package with case implementation using the following link.

Case implementation algorithm

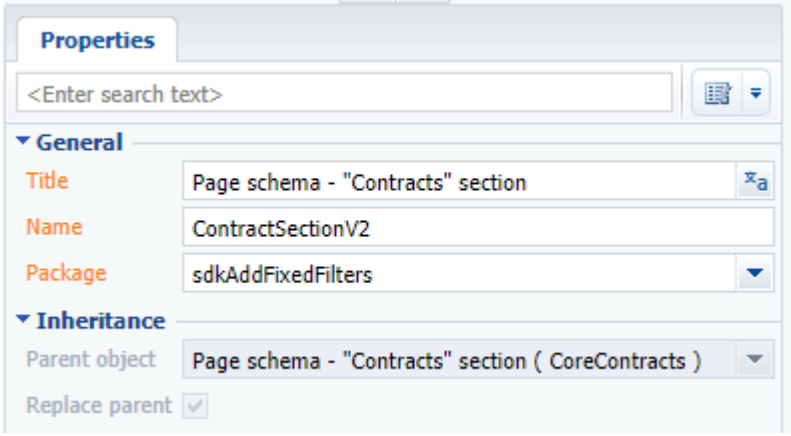
1. Create a replacing schema of the [Contracts] section in the custom package.

Create a replacing custom module and populate its properties with (Fig.1):

- [Parent object] – “Page schema – “Contracts” section”;
- [Name] – *ContractSectionV2*.

The procedure for creating a replacing client schema is covered in the “**Creating a custom client module schema**” article.

Fig. 1. Properties of the [Contracts] section replacing schema

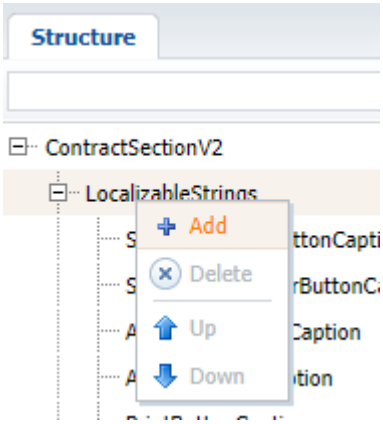


2. Add localizable strings to the schema structure.

Create two new localizable strings (Fig.2) with the following properties:

Name	Value
OwnerFilterCaption	Owner
PeriodFilterCaption	Period

Fig. 2. Adding localizable string to the schema



3. Add the implementation of the initFixedFiltersConfig() method to the method collection of the section view model.

Create a configuration object with the *PeriodFilter* and *OwnerFilter* filter arrays in the *initFixedFiltersConfig()* method, assign a reference to the created configuration object to the *fixedFiltersConfig* view model attribute .

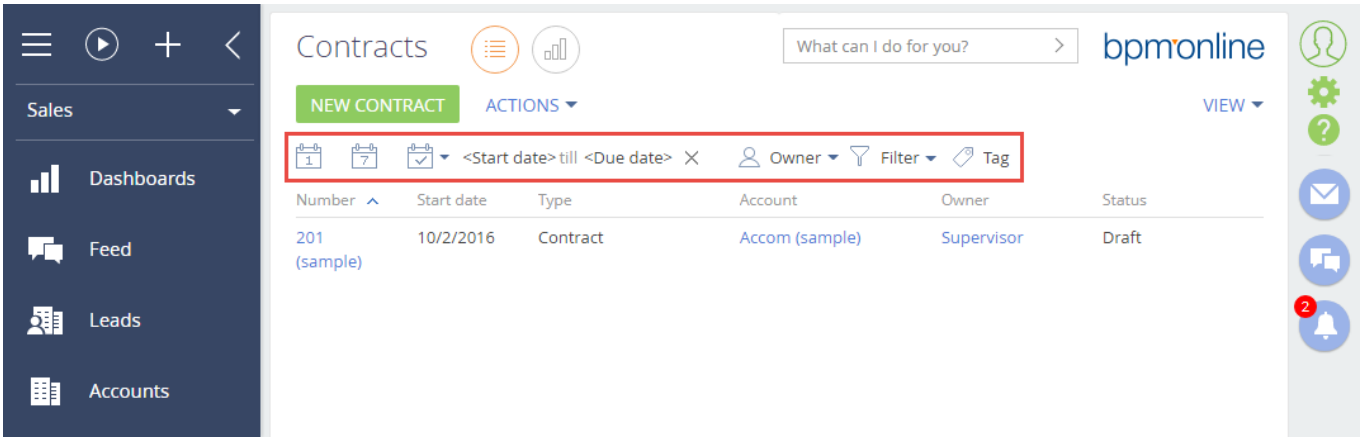
The replacing schema source code is as follows:

```
define("ContractSectionV2", ["BaseFiltersGenerateModule"],
function(BaseFiltersGenerateModule) {
    return {
        // Name of the section schema
        entitySchemaName: "Contract",
        // Method collection of the section view model.
        methods: {
            // Initializes the fixed filters.
            initFixedFiltersConfig: function() {
                // Creating a Configuration Object.
                var fixedFilterConfig = {
                    // The schema of the section object is specified as an object
schema for fixed filters.
                    entitySchema: this.entitySchema,
                    // Array of filters.
                    filters: [
                        // Start period filter.
                        {
                            // The name of the filter.
                            name: "PeriodFilter",
                            // Filter header.
                            caption:
this.get("Resources.Strings.PeriodFilterCaption"),
                            // The data type - date.
                            dataValueType: this.Terrasoft.DataValueType.DATE,
                            // Start date of the filtering period.
                            startDate: {
                                // Filter the data from the [Date] column.
                                columnName: "StartDate",
                                // Default value.
                                defValue: this.Terrasoft.startOfWeek(new Date())
                            },
                            // Date of the filtering period completion.
```

```
                            dueDate: {
                                columnName: "StartDate",
                                defValue: this.Terrasoft.endOfWeek(new Date())
                            }
                        },
                        // Owner filter.
                        {
                            // The name of the filter.
                            name: "Owner",
                            // Filter header.
                            caption:
this.get("Resources.Strings.OwnerFilterCaption"),
                            // Filter the data from the [Owner] column.
                            columnName: "Owner",
                            // Current user contact is specified as default value.
                            // Value is received from the system setting.
                            defValue: this.Terrasoft.SysValue.CURRENT_USER_CONTACT,
                            // The data type - lookup.
                            dataValueType: this.Terrasoft.DataValueType.LOOKUP,
                            // Filter.
                            filter: BaseFiltersGenerateModule.OwnerFilter
                        }
                    ]
                };
                // A link to the configurational object is assigned to the
[FixedFilterConfig] column.
                this.set("FixedFilterConfig", fixedFilterConfig);
            }
        }
    };
});
```

After saving the schema and restarting the system, a block of fixed filters will appear in the [Contracts] section. These filters will enable you to filter contracts by start date and owner (Fig. 3).

Fig. 3. Case result



Page configuration

Introduction

An edit page is a container having a number of fields for entering and changing the columns of section object schema (see **“Section list”**). It opens when you add a new record to the section list, or when you edit the existing record. Every section has one or several edit pages.

Business rules are one of the tools to setup page logic in bpm’online.

Business rules represent a standard bpm’online mechanism that enables you to set up the page field behavior by configuring the view model columns.

Business rules enable you to:

- hide and display fields
- lock and unlock fields
- make fields required or optional
- filter the lookup field value depending on another field value

The **primary control elements** of a page include:

- an input field
- a button
- an image field
- a color button
- a multicurrency field

Bpm’online enables you to add and edit standard control elements on the edit page as well as to create custom control elements.

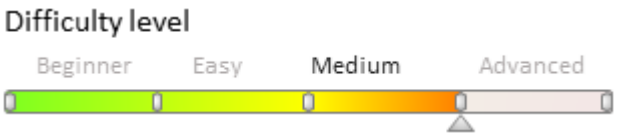
Page configuration options enable setting up the behavior of the existing control elements on the page:

- **add validation**
- **set up calculated fields**
- **apply filtration to lookup fields**
- **setting default values for fields**

Contents

- **Setting the edit page fields using business rules**
- **Adding an action to the edit page**
- **Control elements**
- **Adding an action panel**
- **Adding a new channel to the action panel**
- **Adding calculated fields**
- **How to set a default value for a field**
- **How to add the field validation**
- **Using filtration for lookup fields. Examples**
- **Adding an action panel**
- **Adding a new channel to the action panel**
- **Displaying contact's time zone**
- **How to display the difference between dates on edit page fields**
- **How to block fields of the edit page**

Setting the edit page fields using business rules



Introduction

Business rules are one of the tools to setup page logic in bpm'online.

Business rules are a standard bpm'online mechanism that enables you to set up the page field behavior by configuring the view model columns.

Business rules enable you to:

- hide and display fields
- lock and unlock fields
- make fields required or optional
- filter the lookup field value depending on another field value

You can add business rules in two ways:

1. Via section wizard or detail wizard.

- Wizard generated business rules are added to the *businessRules* client module property.
- The generated business rules have higher priority at execution.
- BusinessRuleModule enumerations are not used when describing the generated business rules.

See the "Setting up business rules" article for more information on business rule setup via wizard. The manual setup of the generated business rules is covered in the "**Business rules. The businessRules property**" article.

2. Via configuring the "rules" property of the client module schema.

The functions of business rules are implemented in the *BusinessRuleModule* client module. To use these functions, add the *BusinessRuleModule* module to the list of user schema dependencies of the view model.

Case of declaring user module with using business rules

```
define("CustomPageModule", ["BusinessRuleModule"], function(BusinessRuleModule) {
    return {
        // View model schema implementation.
    };
});
```

NOTE

Capability of adding business rules by developer tools ensures compatibility with previous versions.

General requirements to business rule declaring in a client module

- All rules are described in the *rules* property of the page view model.
- The rules are applied to view model columns and not to control elements.
- Business rules are not supported on list pages.
- Rules have names.
- Rule parameters are set in its configuration object.

Examples of business rule declaring

```
// List of view model rules.
rules : {
    // Name of the column where the rule is added.
    "FirstColumnName" : {
        // FirstColumnName column list of rules.
        // Rule name.
        FirstRuleName : {
            // FirstRuleName configuration object.
            ruleType: <BusinessRuleModule.enums.RuleType enumeration value>
```

```
            // The rest of rule configuration properties.
        },
        SecondRuleName: {
            // SecondRuleName configuration object.
            ruleType: <BusinessRuleModule.enums.RuleType enumeration value>
            // The rest of rule configuration properties.
        }
    },
    "SecondColumnName" : {
        // SecondColumnName column list of rules.
        ...
    }
}
```

Business rule types

Rule types are defined in the *RuleType* enumeration of the *BusinessRuleModule* module.

Currently two rule types are used – BINDPARAMETER and FILTRATION.

BINDPARAMETER

This rule type is used for linking properties of a column to values of different parameters. For instance, for setting up the visibility of a column or to enable a column depending on the value of another column. The main BINDPARAMETER configuration object properties are described in table 1.

Table 1. – BINDPARAMETER configuring

Property	Value
ruleType	Type of the rule. It is defined by the *BusinessRuleModule.enums.RuleType* enumeration value. In this case BINDPARAMETER type is used.
property	Control element property. Set by the *BusinessRuleModule.enums.Property* enumeration value: • VISIBLE – column visibility • ENABLED – enabling of a column • REQUIRED – column populating is required • READONLY – column for reading only
conditions	Condition array for rule application. Every condition represents a configuration object, whose properties are described in table 2.
logical	Logical operation of combining the conditions from the *conditions* property. Set by the *Terrasoft.LogicalOperatorType* enumeration value.

Table 2. – BINDPARAMETER configuring

Property	Value
leftExpression	Expression of the left side of the condition. Represents a configuration object with the following properties: • *Type* – expression type. Set by the *BusinessRuleModule.enums.ValueType* enumeration value: • CONSTANT – constant value • ATTRIBUTE – the view model column value • SYSSETTING – system setting • SYSVALUE – system value The *Terrasoft.core.enums.SystemValueType* system value list element. • *Attribute* – model column name • *attributePath* – meta-path to the lookup schema column • *Value* – value for comparison
comparisonType	Type of comparison. Set by the *Terrasoft.core.enums.ComparisonType* enumeration value.
rightExpression	Expression of the right side of the condition. Similar to *leftExpression*.

FILTRATION

Use the FILTRATION rule to set up filtering of values in view model columns. For example, you can filter a lookup column depending on the current status of a page.

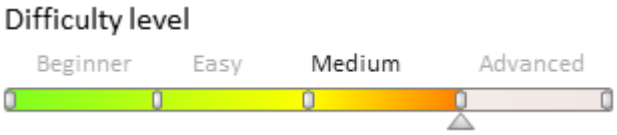
Table 3. – FILTRATION configuring

Property	Value
ruleType	Rule type. Set by the *BusinessRuleModule.enums.RuleType*.enumeration value. In this case FILTRATION type is used.
autocomplete	Reverse filtering checkbox. Can take the *true* or *false* values.
autoClean	The checkbox of automated value cleaning upon changing the column that is used for filtration. Can take the *true* or *false* values.
baseAttributePatch	Meta-path to the lookup schema column that will be used for filtration. The feedback principle is applied when building the column path similar to **EntitySchemaQuery**,. The path is generated in relation to the schema, referred to by the model column.
comparisonType	Type of comparison operation. Set by the *Terrasoft.ComparisonType*.enumeration value.
type	The value type for comparison *baseAttributePatch*. Set by the *BusinessRuleModule.enums.ValueType* enumeration value.
attribute	The view model column name. This property is described if ATTRIBUTE value type (*type*) is indicated.
attributePath	Meta-path to the object schema column. The feedback principle is applied when building the column path similar to **EntitySchemaQuery**,. The path is generated in relation to the schema, referred to by the model column.
value	Filtration value. This property is described if ATTRIBUTE value type (*type*) is indicated.

See also

- **The FILTRATION rule use case**
- **The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to make a field required based on a specific condition**
- **Business rules created via wizards**

The FILTRATION rule use case



Introduction

The *FILTRATION* rule is used to configure filtering of the lookup column of the view model based on the value of another column. For more information on business rules, see the "**Setting the edit page fields using business rules**" article.

NOTE

In bpm'online, you can configure business rules using developer tools the as well as the section wizard. For more information please refer to the "Setting up the business rules" article.

Case description

Add the [Country], [State/Province] and [City] fields to the page. If the [Country] field is populated, the values in the [State/Province] field must include only states and provinces of that country. If the [State/Province] field is populated, the values in the [City] field must include only cities located in that state or province. If the [City] field is populated first, the [Country] and [State/Province] fields must be automatically populated with the corresponding values.

NOTE

The base contact page schema already has a rule for filtering cities by country. Therefore, if the [Country] field is not added, only cities from the country specified for a contact can be selected.

Source code

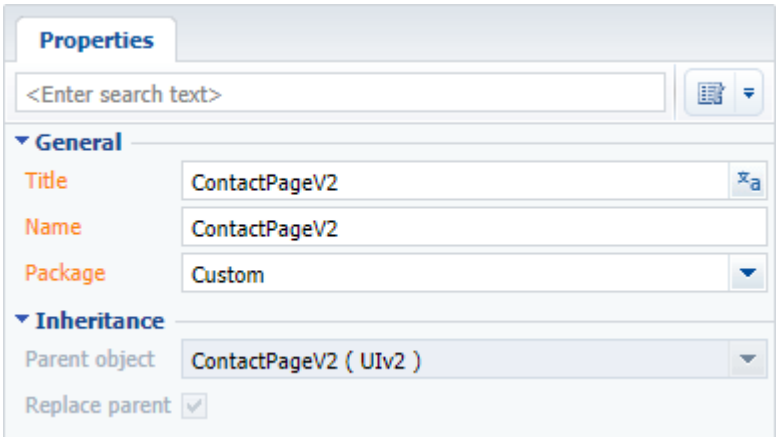
Use this link to download the case implementation package.

Case implementation algorithm

1. Create a replacing contact page

Create a replacing client module and specify the [Display schema — Contact card] schema as parent object (Fig. 1). The procedure for creating a replacing page is covered in the "**Creating a custom client module schema**" article.

Fig. 1. Order edit page replacing schema properties



2. Add the [Country], [State/Province] and [City] fields to the page.

To do this, add three configuration objects with the settings for the corresponding field properties to the *diff* array.

3. Add FILTRATION-type rules to the [City] and [State/Province] columns.

To do this, add two rules of the *BusinessRuleModule.enums.RuleType.FILTRATION* type to the *rules* property for the [City] and [Region] columns. To enable reverse filtering (i.e., to automatically populate the [Country] and [State/Province] fields based on the selected city), set the *autocomplete* property to *true*.

The replacing schema source code is as follows:

```
// Add the module BusinessRuleModul to the dependency list of the module.
define("ContactPageV2", ["BusinessRuleModule"],
    function(BusinessRuleModule) {
        return {
            // Name of the schema of the edit page object.
            entitySchemaName: "Contact",
            // A property that contains a collection of business rules for the schema
of the page view model.
            rules: {
                // A set of rules for the [City] column of the view model..
                "City": {
                    // The rule for filtering the [City] column by the value of the
[Region] column.
                    "FiltrationCityByRegion": {
                        //  FILTRATION rule type.
                        "ruleType": BusinessRuleModule.enums.RuleType.FILTRATION,
                        // Reverse filtering will be performed.
                        "autocomplete": true,
                        // The value will be cleared when the value of the [Region]
column changes.
                        "autoClean": true,
                        // The path to the column for filtering in the [City]
reference schema,
                        // which is referenced by the [City] column of the
                        // edit page view model.
                        "baseAttributePatch": "Region",
                        // The type of the comparison operation in the filter.
                        "comparisonType": Terrasoft.ComparisonType.EQUAL,
                        // The column (attribute) of the view model will be the
comparison value.
                        "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                        // The column name of the view model of the edit page,
                        // the value of which will be filtered.
                        "attribute": "Region"
                    }
```

```
                },
                // A set of rules for the [Region] column of the view model.
                "Region": {
                    "FiltrationRegionByCountry": {
                        "ruleType": BusinessRuleModule.enums.RuleType.FILTRATION,
                        "autocomplete": true,
                        "autoClean": true,
                        "baseAttributePatch": "Country",
                        "comparisonType": Terrasoft.ComparisonType.EQUAL,
                        "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                        "attribute": "Country"
                    }
                }
            },
            // Setting up the visualization of the [Country], [State/Province] and
[City] fields on the edit page.
            diff: [
                // Metadata for adding the [Country] field.
                {
                    "operation": "insert",
                    "parentName": "ProfileContainer",
                    "propertyName": "items",
                    "name": "Country",
                    "values": {
                        "contentType": Terrasoft.ContentType.LOOKUP,
                        "layout": {
                            "column": 0,
                            "row": 6,
                            "colSpan": 24
                        }
                    }
                },
                // Metadata for adding the [State/Province] field.
                {
                    "operation": "insert",
                    "parentName": "ProfileContainer",
                    "propertyName": "items",
                    "name": "Region",
                    "values": {
                        "contentType": Terrasoft.ContentType.LOOKUP,
                        "layout": {
                            "column": 0,
                            "row": 7,
                            "colSpan": 24
                        }
                    }
                },
                // Metadata for adding the [City] field.
                {
                    "operation": "insert",
                    "parentName": "ProfileContainer",
                    "propertyName": "items",
                    "name": "City",
                    "values": {
                        "contentType": Terrasoft.ContentType.LOOKUP,
                        "layout": {
                            "column": 0,
                            "row": 8,
                            "colSpan": 24
                        }
                    }
                }
```

```
            ]
        };
    });
```

4. Save the created replacing page schema

After saving the schema and updating the application page, three new fields will be added to the contact profile (Fig. 2). Their values will be filtered based on the values entered in any of these fields. The filtering also works in the lookup selection window (Fig. 4).

Fig. 2. New fields in the contact profile

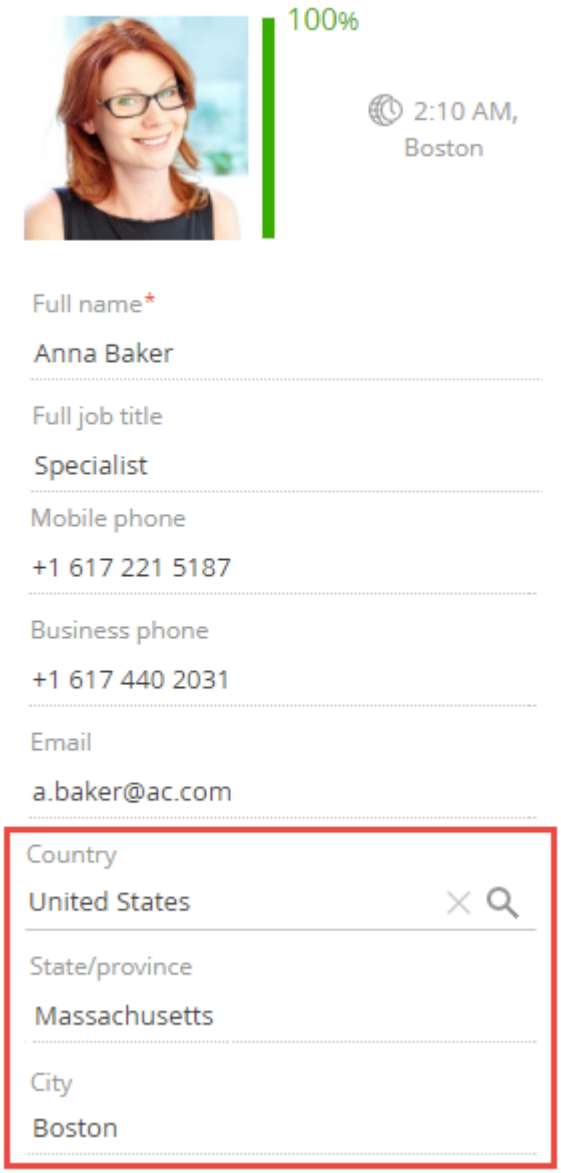


Fig. 3. Filtering

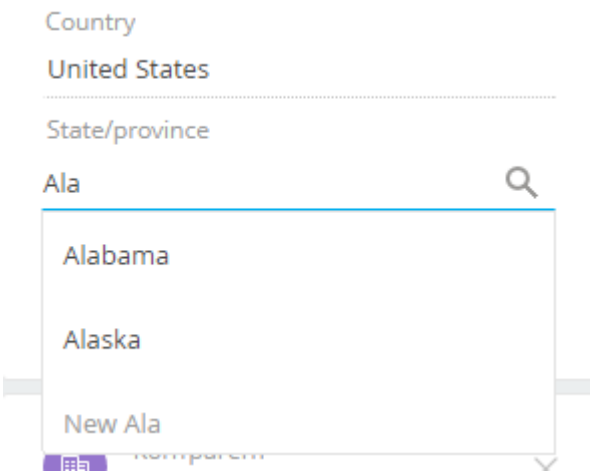
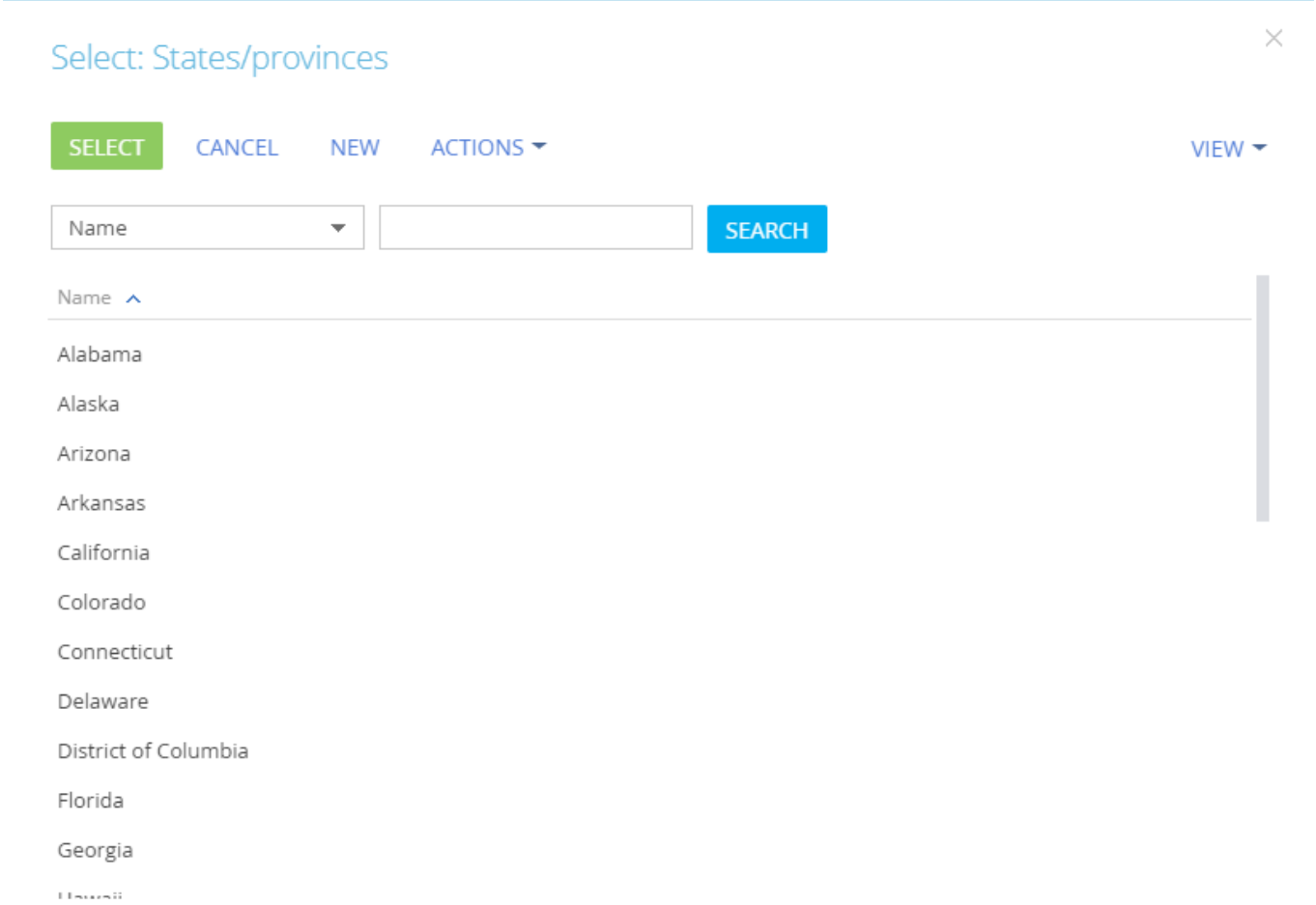


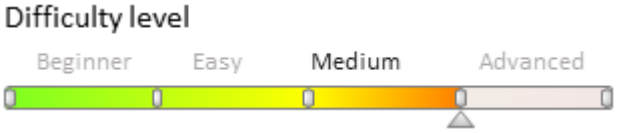
Fig. 4. Filtered values in the lookup selection window



See also

- **Setting the edit page fields using business rules**
- **The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to make a field required based on a specific condition**

The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition



Introduction

BINDPARAMETER rule is used for resolving he following tasks:

- hide and display fields
- lock and unlock fields
- make fields required or optional

For more information on business rules, see the “**Setting the edit page fields using business rules**” article.

> NOTE
>
> In bpm’online, you can configure business rules using developer tools as well as the section wizard. For more information please refer to the “Setting up the business rules”.

Case description

Add a new [Meeting place] field to the activity page. The field will be available only for activities of the [Meeting] type.

> NOTE
>
> You can add fields to the edit page manually or via the section wizard.
>
> For more on adding fields to edit pages see the “**Adding a new field to the edit page**” article.

Source code

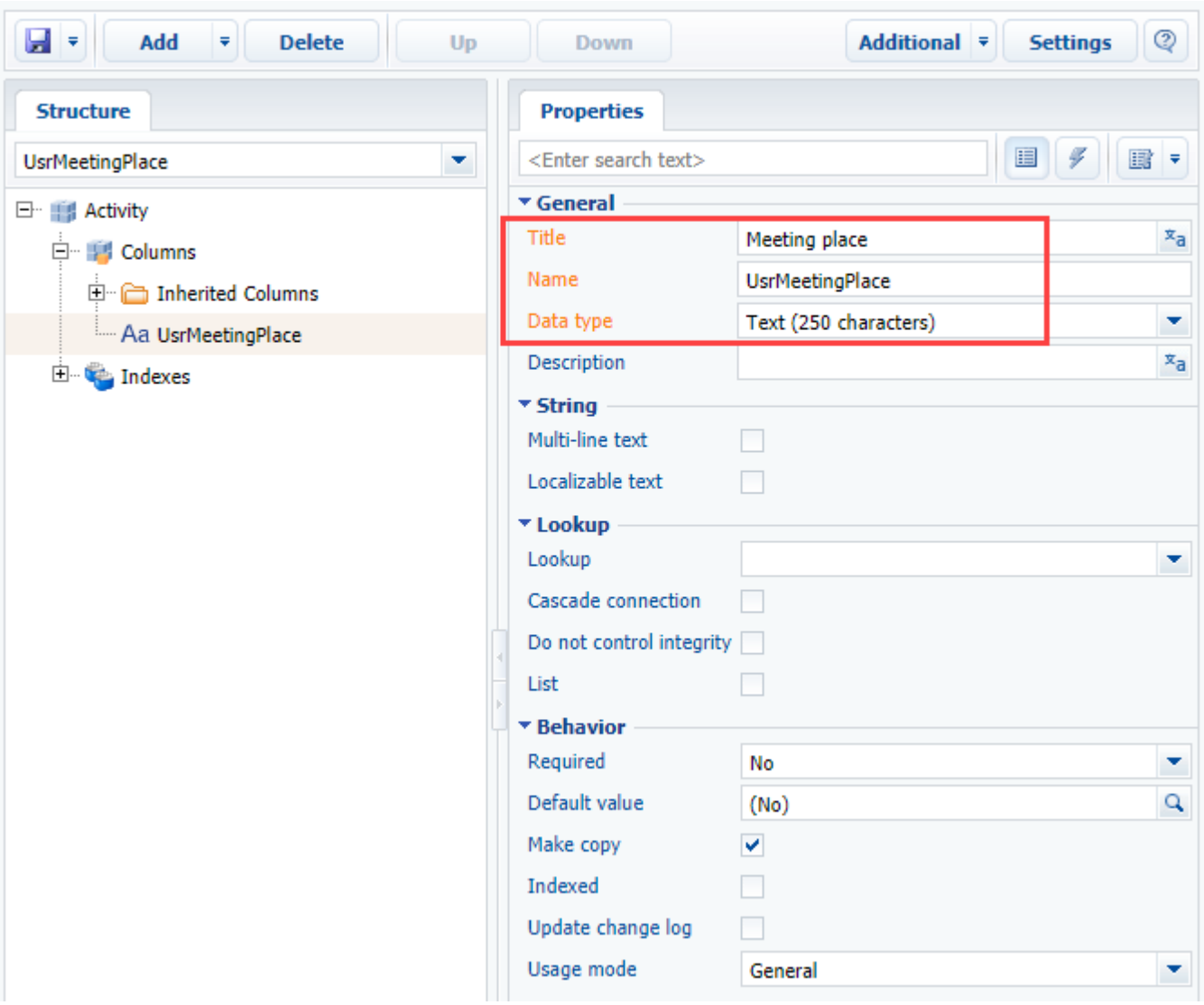
Use this link to download the case implementation package.

Case implementation algorithm

1. Create a replacing object and add a new column to it.

Create an [Activity] replacing object and add a new [Meeting place] column of the “string” type to it (Fig. 1). Learn more about creating a replacing object schema in the “**Creating the entity schema**” article.

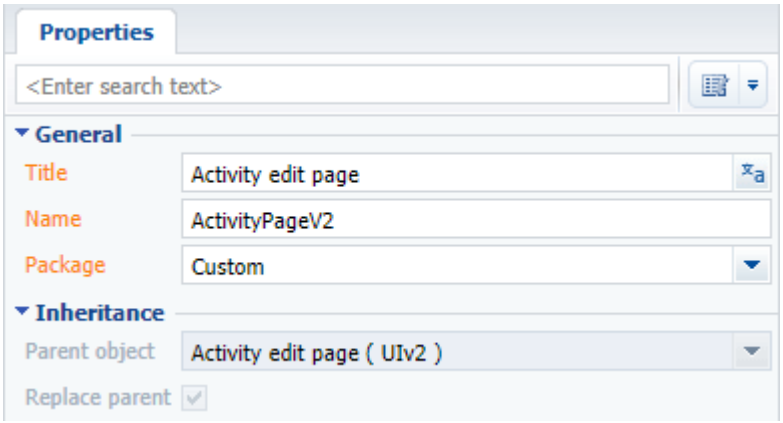
Fig. 1. Adding a custom column to the replacing object



2. Create a replacing client module for the activity page

Create a replacing client module and specify the [Activity edit page] schema as parent object (Fig. 2). The procedure for creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 2. Replacing edit page properties



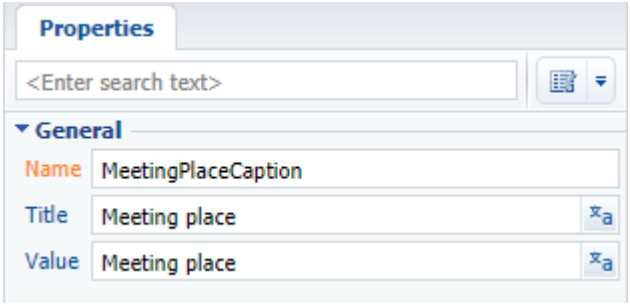
3. Add a new field to the activity edit page.

Add a configuration object with the [Meeting place] field properties on the page to the diff array. The process of

adding fields to pages is covered in the "**Adding a new field to the edit page**" article.

To enable localization of this field, add a localizable string (Fig. 3) and bind it to the field title.

Fig. 3. Localizable string properties



4. Add a rule to the "rules" property of the page view model

For the *UsrMeetingPlace* column, add the rule with the BINDPARAMETER type to *rules* property of the page view model. Set the *BusinessRuleModule.enums.Property.VISIBLE* value for the rule's *property*. Add the following condition for rule execution to the *conditions* array: the value in the *ActivityCategory* column of the model should be equal to the *ConfigurationConstants.Activity.ActivityCategory.Meeting* configuration constant.

> ⚠ NOTE
>
> The *ConfigurationConstants.Activity.ActivityCategory.Meeting* configurational constant contain the id of the "Meeting" record of the [Activity category] lookup.

The replacing schema source code is as follows:

```
// Add the module BusinessRuleModule and ConfigurationConstants to the dependency
list of the module.
define("ActivityPageV2", ["BusinessRuleModule", "ConfigurationConstants"],
    function(BusinessRuleModule, ConfigurationConstants) {
        return {
            // Name of the page schema of the edit page.
            entitySchemaName: "Activity",
            // Displaying a new field on the edit page.
            diff: /**SCHEMA_DIFF*/[
                // Metadata for adding a field [Meeting place].
                {
                    // The operation of adding a component to a page.
                    "operation": "insert",
                    // The meta name of the parent container to which the field is
added.
                    "parentName": "Header",
                    // The field is added to the parent
                    // component's collection.
                    "propertyName": "items",
                    // The name of the column of the schema to which the component is
bound.
                    "name": "UsrMeetingPlace",
                    "values": {
                        // Field title.
                        "caption": {"bindTo":
"Resources.Strings.MeetingPlaceCaption"},
                        // Location of the field.
                        "layout": { "column": 0, "row": 5, "colSpan": 12 }
                    }
                }
            ]/**SCHEMA_DIFF*/,
            // Rules of the edit page view model  Правила модели представления
страницы редактирования.
```

```
            rules: {
                // A set of rules for the [Meeting place] column of the view model.
                "UsrMeetingPlace": {
                    // The dependence of visibility of the [Meeting Place] field from
the value in [Category] field.
                    "BindParametrVisibilePlaceByType": {
                        // The type of the BINDPARAMETER rule.
                        "ruleType": BusinessRuleModule.enums.RuleType.BINDPARAMETER,
                        // Rule regulates the VISIBLE property.
                        "property": BusinessRuleModule.enums.Property.VISIBLE,
                        // An array of conditions in which the rule is triggered.
                        // Determines whether the value in the [Category] column is
equal to the value "Meeting".
                        "conditions": [{
                            // Expression of the left side of the condition.
                            "leftExpression": {
                                //The type of the expression is the attribute
(column) of the view model.
                                "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                                // Name of the view model column which value is
compared in the expression.
                                "attribute": "ActivityCategory"
                            },
                            // The type of comparison operation.
                            "comparisonType": Terrasoft.ComparisonType.EQUAL,
                            // Expression of the right side of the condition.
                            "rightExpression": {
                                // Type of expression is a constant value.
                                "type": BusinessRuleModule.enums.ValueType.CONSTANT,
                                // The value with which the left side expression is
compared.
                                "value":
ConfigurationConstants.Activity.ActivityCategory.Meeting
                            }
                        }]
                    }
                }
            }
        };
    });
```

After saving the schema and refreshing the application page, an additional [Meeting place] field will appear on the activity page if the activity category is “Meeting” (Fig. 5, 5).

Fig. 4. Case result. Activity type is “To do”, the [Meeting place] field is not visible

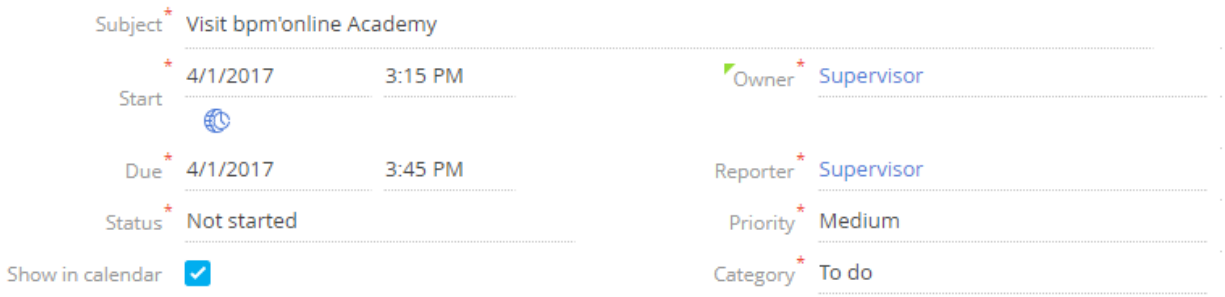
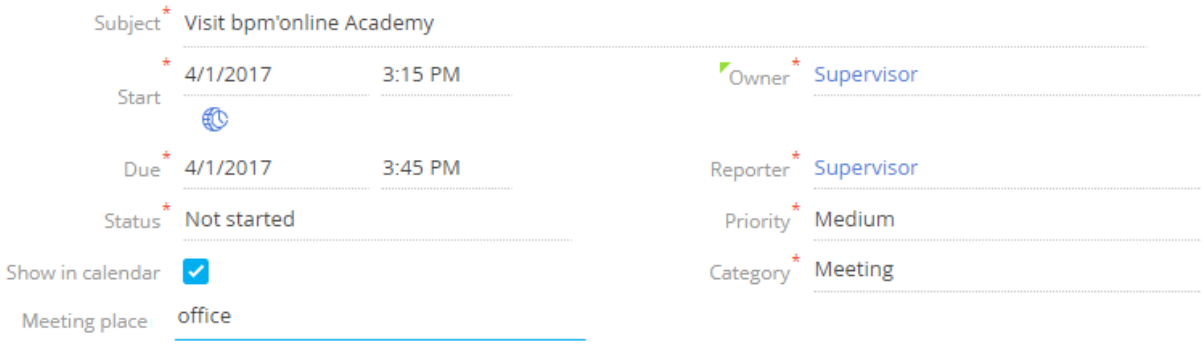


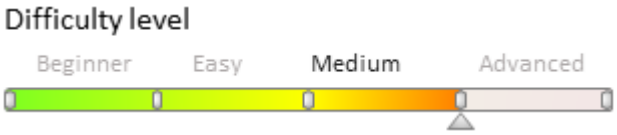
Fig. 5. Case result. Activity type is “To do”, the [Meeting place] field is visible



See also

- **Setting the edit page fields using business rules**
- **The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to make a field required based on a specific condition**
- **The FILTRATION rule use case**
- **Business rules created via wizards**

The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition



Introduction

BINDPARAMETER rule is used for resolving he following tasks:

- hide and display fields
- lock and unlock fields
- make fields required or optional

For more information on business rules, see the "**Setting the edit page fields using business rules**" article.

> NOTE
>
> In bpm'online, you can configure business rules using developer tools the as well as the section wizard. For more information please refer to the "Setting up the business rules".

Case description

Configure fields on the contact edit page to make the [Business phone] field editable only if the [Mobile phone] field is filled.

Source code

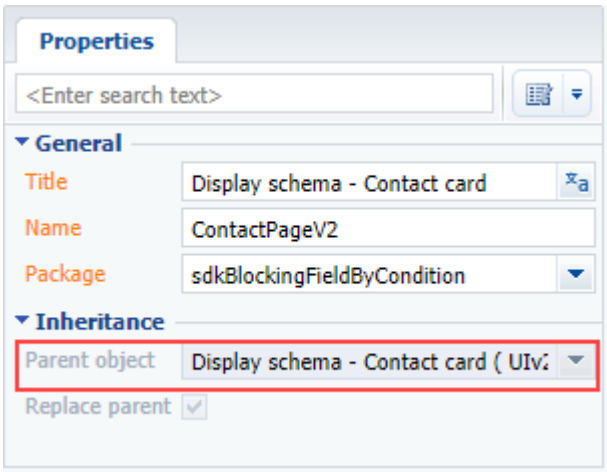
Use this link to download the case implementation package.

Case implementation algorithm

1. Create a replacing client module for the contact page

Create a replacing client module and specify the [Display schema — Contact card] schema as parent object (Fig. 1). The procedure for creating a replacing page is covered in the "**Creating a custom client module schema**" article.

Fig. 1. Order edit page replacing schema properties



2. In the rules property of the page view model, add the rule

For the *Phone* column, add the rule with the BINDPARAMETER type to *rules* property of the page view model. Set the BusinessRuleModule.enums.Property.ENABLED. Value for the rule's *property*. Add the following condition for rule execution to the *conditions* array: the value in the *MobilePhone* column should be filled.

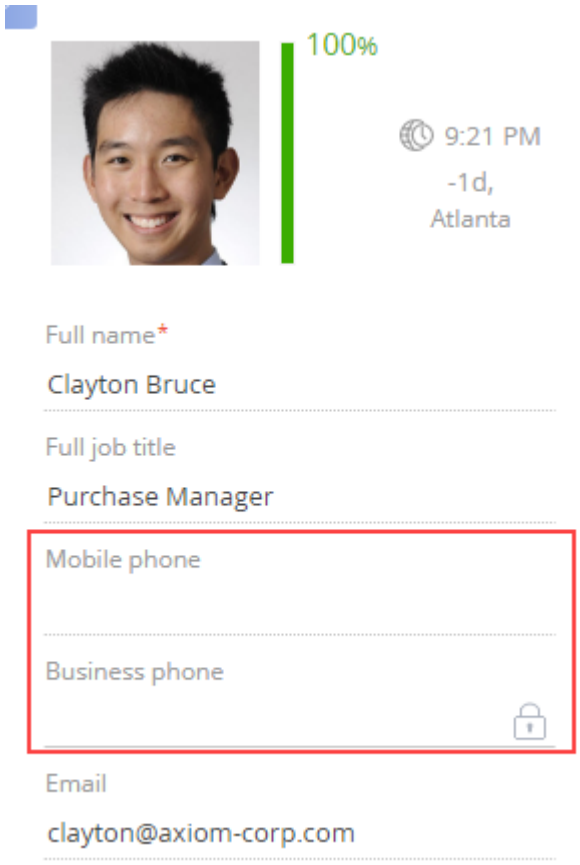
The replacing schema source code is as follows:

```
// Add the module BusinessRuleModule to the list of dependent modules.
define("ContactPageV2", ["BusinessRuleModule"], function(BusinessRuleModule) {
    return {
        // Name of the page schema of the edit page.
        entitySchemaName: "Contact",
        // Rules of the edit page view model.
        rules: {
            // A set of rules for the [Business phone] column of the view model.
            "Phone": {
                // Dependence of the availability of the [Business phone] field from
the value of the [Mobile phone] field.
                "BindParameterEnabledPhoneByMobile": {
                    // The type of the BINDPARAMETER rule.
                    "ruleType": BusinessRuleModule.enums.RuleType.BINDPARAMETER,
                    // The rule regulates the ENABLED property.
                    "property": BusinessRuleModule.enums.Property.ENABLED,
                    // An array of conditions in which the rule is triggered.
                    // Determines whether the [Mobile Phone] field is populated.
                    "conditions": [{
                        // Expression of the left side of the condition.
                        "leftExpression": {
                            // The type of the expression is the attribute (column)
of the view model.
                            "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                            // The name of the column in the view model, whose value
is compared in the expression.
                            "attribute": "MobilePhone"
                        },
                        // The type of comparison operation is "not equal to".
                        "comparisonType": Terrasoft.ComparisonType.NOT_EQUAL,
                        // Expression of the right side of the condition.
```

```
                        "rightExpression": {
                            // The expression type is a constant value.
                            "type": BusinessRuleModule.enums.ValueType.CONSTANT,
                            // The value with which the left side expression is
compared.
                            "value": ""
                        }
                    }]
                }
            }
        }
    };
});
```

After saving the schema and refreshing the application page, the [Business phone] field will be non-editable until the [Mobile phone] field is empty (Fig. 2).

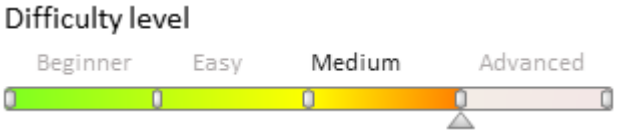
Fig. 2. Example result demonstration



See also

- **Setting the edit page fields using business rules**
- **The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to make a field required based on a specific condition**
- **The FILTRATION rule use case**
- **Business rules created via wizards**

The BINDPARAMETER rule. How to make a field required based on a specific condition



Introduction

BINDPARAMETER rule is used for resolving he following tasks:

- hide and display fields
- lock and unlock fields
- make fields required or optional

For more information on business rules, see the "**Setting the edit page fields using business rules**" article.

NOTE

In bpm'online, you can configure business rules using developer tools as well as the section wizard. For more information please refer to the "Setting up business rules" article.

Case description

Set up the contact edit page fields so that the [Business phone] field is required on condition that the [Contact type] field is populated with the "Customer" value.

Source code

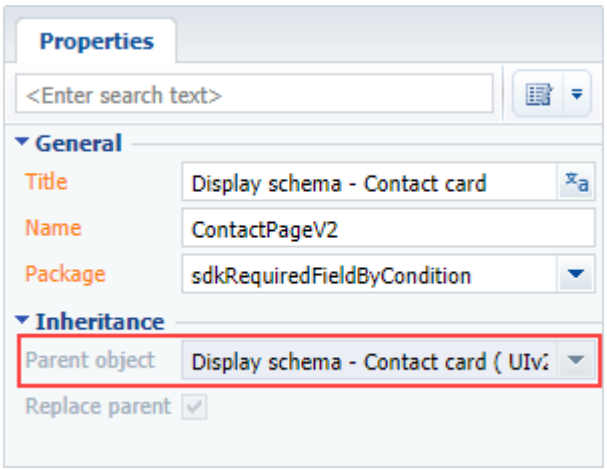
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Create a replacing client module for the contact edit page.

Create a replacing client module and specify the [Display schema — Contact card] schema as parent object (Fig. 1). The procedure for creating a replacing page is covered in the "**Creating a custom client module schema**" article.

Fig. 1. Replacing edit page properties



2. Add a rule to the "rules" property of the page view model

Add the BINDPARAMETER type rule for the *Phone* column to the *rules* property of the page view model. Set the *property* rule value to BusinessRuleModule.enums.Property.REQUIRED. Add a rule execution condition to the *conditions* array – the *Type* column value of the model should be equal to the *ConfigurationConstants.ContactType.Client* configuration constant.

> **NOTE**
>
> The *ConfigurationConstants.ContactType.Client* configuration constant contains the "Client" record identifier of the [Contact type] lookup.

The replacing schema source code is as follows:

```
// Add the BusinessRuleModule and ConfigurationConstants modules to the module
dependency list.
define("ContactPageV2", ["BusinessRuleModule", "ConfigurationConstants"],
    function(BusinessRuleModule, ConfigurationConstants) {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Contact",
            // Rules of the edit page view model.
            rules: {
                // Set of rules of the [Business rule] view model column.
                "Phone": {
                    // Dependency of the [Business phone] field "required" property
on the [Type] field value.
                    "BindParameterRequiredAccountByType": {
                        // BINDPARAMETER rule type.
                        "ruleType": BusinessRuleModule.enums.RuleType.BINDPARAMETER,
                        // The rule regulates the REQUIRED property.
                        "property": BusinessRuleModule.enums.Property.REQUIRED,
                        // Condition array, whose performance triggers the rule
execution.
                        // Defines if the [Type] column value is equal to the
"Client" value.
                        "conditions": [{
                            // Expression of the left side of the condition.
                            "leftExpression": {
                                // Expression type – view model attribute(column).
                                "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                                // Name of the view model column whose value is
compared in the expression.
                                "attribute": "Type"
                            },
                            // Comparison operation type.
                            "comparisonType": Terrasoft.ComparisonType.EQUAL,
                            // Expression of the right side of the condition.
                            "rightExpression": {
                                // Expression type - constant value.
                                "type": BusinessRuleModule.enums.ValueType.CONSTANT,
                                // The comparison value for the left side of the
expression.
                                "value": ConfigurationConstants.ContactType.Client
                            }
                        }]
                    }
                }
            }
        };
    });
```

After you save the schema and update the application web page, the [Business phone] filed of the contact edit page

will be required on condition the contact type is the “Customer”.

Fig. 2. Case result The [Business phone] field – optional

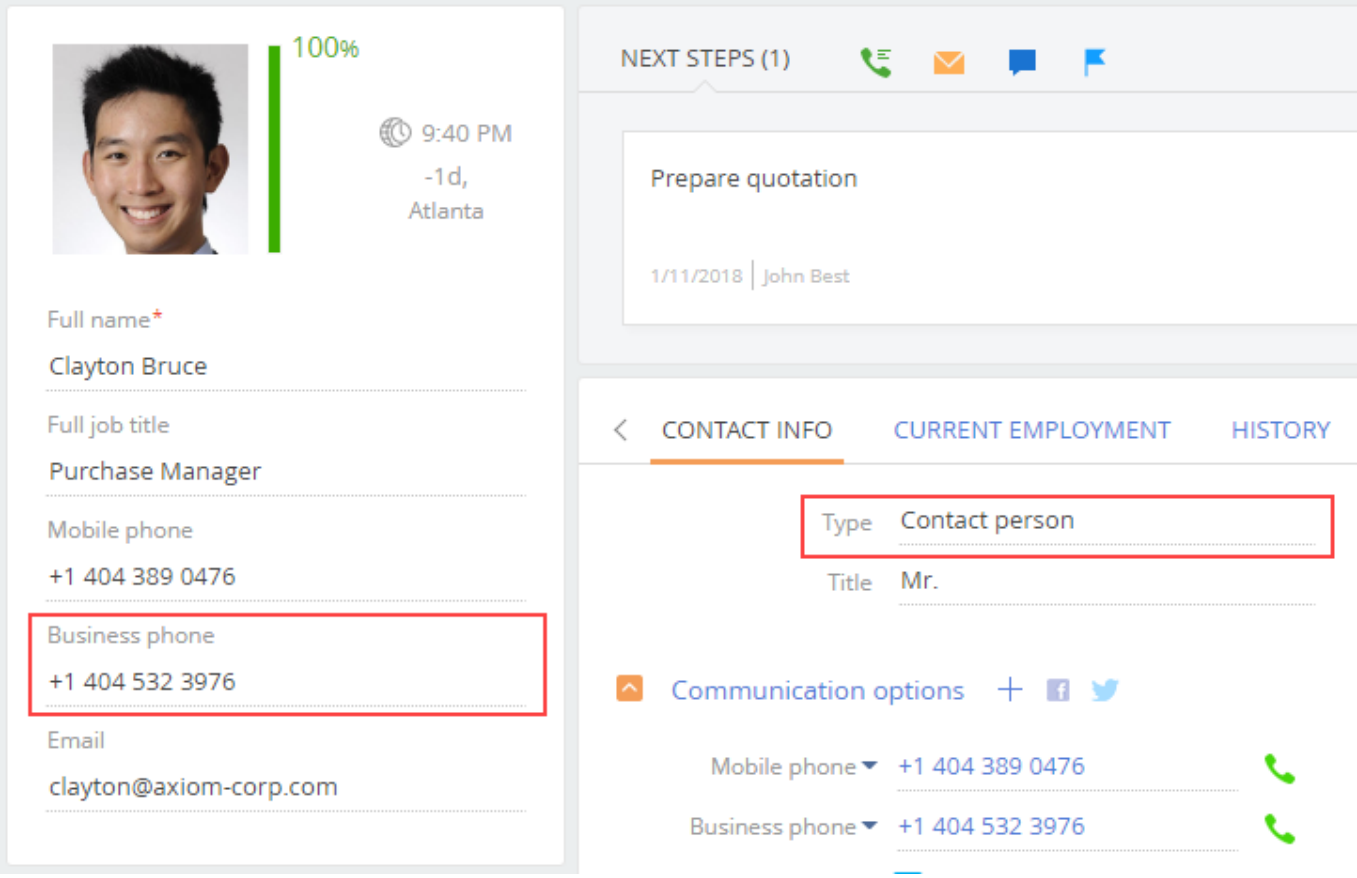
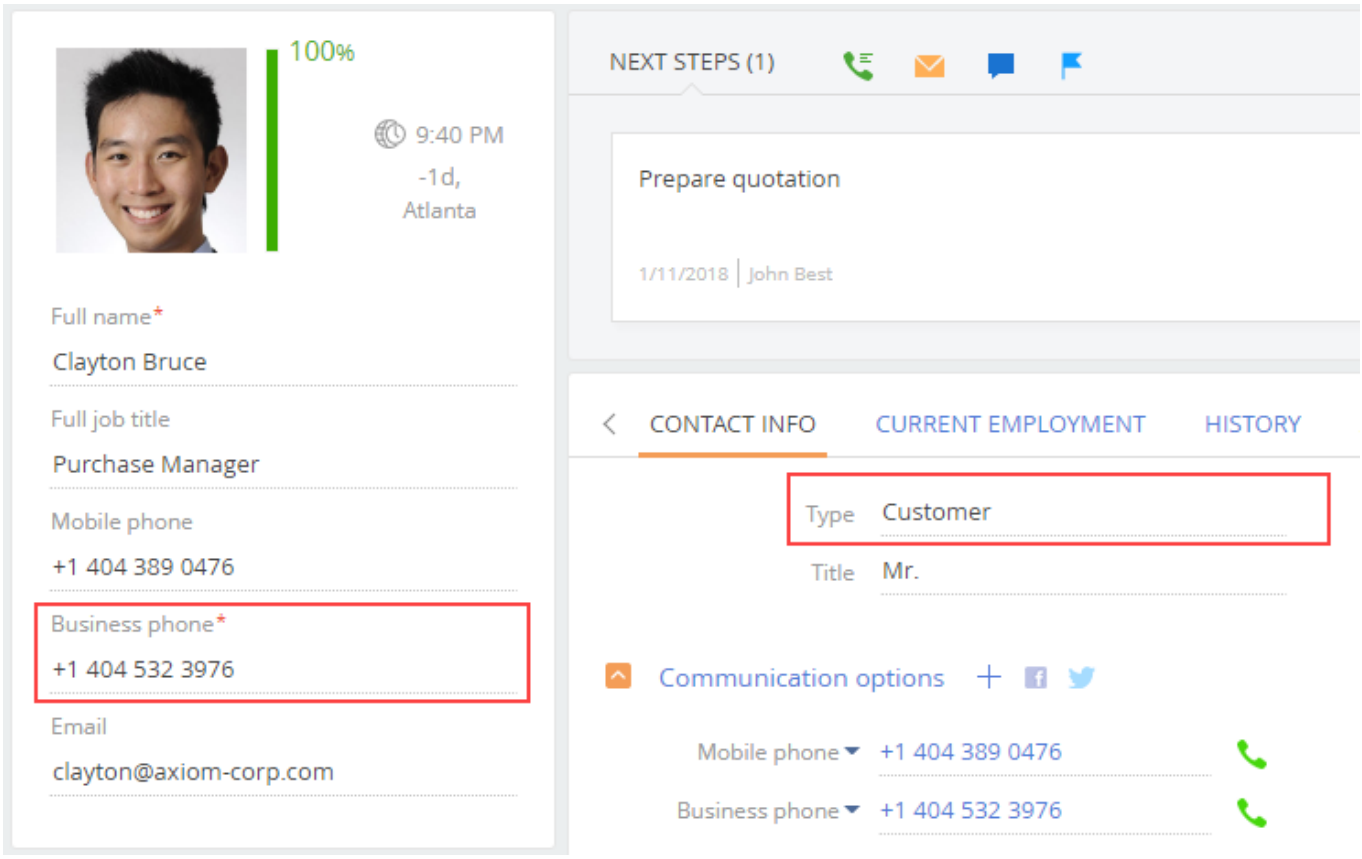


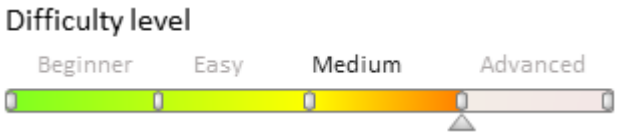
Fig. 3. Case result The [Business phone] field – required



See also

- **Setting the edit page fields using business rules**
- **The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition**
- **The FILTRATION rule use case**
- **Business rules created via wizards**

Business rules created via wizards



Introduction

Starting from version 7.10.0 besides the **business rules created by developer tools**, there exist the business rules generated by section or detail wizards.

- Wizard generated business rules are added to the *businessRules* client module property.
- The generated business rules have higher priority at execution.
- The *BusinessRuleModule* enumerations are not used when describing the generated business rules.

See the "Setting up business rules" article for more information on business rule setup via wizards. The manual setup of the generated business rules is covered in the "**Business rules. The businessRules property**" article.

Additional properties

You can find additional properties of the wizard generated business rules in table 1.

Table 1. Additional properties

Property	Details
uId.	Unique rule identifier. The "GUID” type value.
enabled	Enabling checkbox. Can take the *true* or *false* values.
removed	The checkbox indicating if the rule is removed. Can take the *true* or *false* values.
invalid	The checkbox indicating if the rule is valid. Can take the *true* or *false* values.

Cases

Case of a master generated business rule connected with a field property (whether it is visible, enabled or required):

```
define("SomePage", [], function() {
  return {
     // ...
     businessRules: /**SCHEMA_BUSINESS_RULES*/{
        // Set of rules for the [Type] column of the view model.
        "Type": {
           // Wizard generated rule code.
           "ca246daa-6634-4416-ae8b-2c24ea61d1f0": {
              // Unique rule identifier.
              "uId": "ca246daa-6634-4416-ae8b-2c24ea61d1f0",
              // Enabling checkbox.
              "enabled": true,
              // Checkbox indicating if the rule is removed.
              "removed": false,
              // Checkbox indicating if the rule is valid.
              "invalid": false,
              // Rule type.
              "ruleType": 0,
              // The property code, regulating the rule.
              "property": 0,
              // Logical connection between several rule conditions.
              "logical": 0,
              // Condition array, whose performance triggers the rule implementation.
              // Compares the [Account.PrimaryContact.Type] value with the [Type]
column value.
              "conditions": [
                 {
                    // Comparison operation type.
                    "comparisonType": 3,
                    // Expression of the left side of the condition.
                    "leftExpression": {
                       // Expression type – the view model column (attribute).
                       "type": 1,
                       // The view model column name.
                       "attribute": "Account",
                       // The path to the [Account] lookup schema, whose value
                       // is compared in the expression.
                       "attributePath": "PrimaryContact.Type"
                    },
                    // Expression of the right side of the condition.
                    "rightExpression": {
                       // Expression type – the view model column (attribute).
```

```
                        "type": 1,
                        // The view model column name.
                        "attribute": "Type"
                    }
                }
            ]
        }
    }
  }/**SCHEMA_BUSINESS_RULES*/
  // ..
};
});
```

Case of a master generated business rule for field filtration:

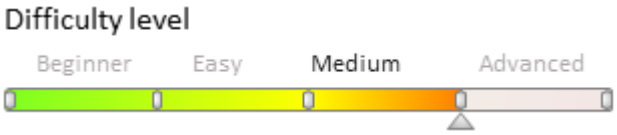
```
define("SomePage", [], function() {
  return {
    // ...
    businessRules: /**SCHEMA_BUSINESS_RULES*/{
      // Set of rules for the [Type] column of the view model.
      "Account": {
        // Master generated rule code.
        "a78b898c-c999-437f-9102-34c85779340d": {
          // Unique rule identifier.
          "uId": "a78b898c-c999-437f-9102-34c85779340d",
          // Enabling checkbox.
          "enabled": true,
          // Checkbox indicating if the rule is removed.
          "removed": false,
          // Checkbox indicating if the rule is valid.
          "invalid": false,
          // Rule type.
          "ruleType": 1,
          // Path to the filtration column of the [Account] lookup schema,
          // that the [Type] column of the edit page view model
          // refers to.
          "baseAttributePatch": "PrimaryContact.Type",
          // Filter comparison operation type.
          "comparisonType": 3,
          // Expression type – the view model column (attribute).
          "type": 1,
          // The view model column name,
          // whose value will be used for filtration.
          "attribute": "Type"
        }
      }
    }/**SCHEMA_BUSINESS_RULES*/
    // ..
  };
});
```

See also

- **Setting the edit page fields using business rules**
- **The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to make a field required based on a specific condition**

- **The FILTRATION rule use case**

Adding an action to the edit page



Introduction

Bpm’online has the possibility to set up a list of actions from the standard [Actions] menu on the edit page.

The list of page actions is an instance of the *Terrasoft.BaseViewModelCollection class*. Each item of the actions list is a view model.

An action is set up in the configuration object where both properties of the actions view model may be set explicitly and the base binding mechanism may be used.

The base content of the [Actions] menu for the edit page is implemented in the base class of the *BasePageV2* pages. The list of section actions returns the *getActions()* protected virtual method from the *BasePageV2* schema.

A separate action is added to the collection by calling the *addItem()* method. The *getButtonMenuItem()* callback method is passed to it as a parameter. The method creates an instance of the actions view model by the configuration object passed to it as the parameter.

Base implementation of addig the action

```
/**
* Returns the collection of edit page actions
* @protected
* @virtual
* @return {Terrasoft.BaseViewModelCollection} Returns the collection of page actions
*/
getActions: function() {
    // List of actions - Terrasoft.BaseViewModelCollection instance
    var actionMenuItems = this.Ext.create("Terrasoft.BaseViewModelCollection");
    // Adding an action to the collection. The method instanting the action model
    // instance by the passed configuration object is passed as callback.
    actionMenuItems.addItem(this.getButtonMenuItem({
        // Configuration object for action setting.
        ...
    }));
    // Returns a new colection of actions.
    return actionMenuItems;
}
```

Below are the properties of the configuration object of the section action to be passed as a parameter to the *getButtonMenuItem()* method:

Table 1. Property of the configuration object

Property	Details
Type.	a type of the [Actions] menu item A horizontal line for separating the menu blocks may be added to the action menu using this property. For this purpose, the *Terrasoft.MenuSeparator* string must be specified as the property value. If no property value is specified, the menu item will be added by default.
Caption	the title of the [Actions] menu item. To set titles, the use of localizable schema strings is recommended.
Tag	the name of the action handler method is set in this property
Enabled	a logical property controlling the menu item availability

Property	Details
Visible	a logical property controlling the menu item visibility

Procedure for adding a custom action

1. Create replacing schema of existing page or a new page.
2. Override the *getActions()* method.
3. Add an action to the actions collection using the *addItem()* method.
4. Pass a configuration object with the added action settings to the *getButtonMenuItem()* callback method.

ATTENTION

When base sections are replaced in the *getActions()* method of the replacing module, the parent implementation of this method must be called first to initialize actions of the parent section. For this, execute the *this.callParent(arguments)* method that returns collection of base page actions.

Case description

The [Show execution date] which will display the scheduled order execution date in the data window must be added to the edit page. The action will be available only for orders at the [In progress] stage.

ATTENTION

The [Orders] section is available in bpm’online sales products.

NOTE

The edit page action is used to edit a specific object opened on the page. To have access to values of the edit page object fields in the action handler method, the following view model methods must be used: *get()* – to receive a value and *set()* - to set a value.

Source code

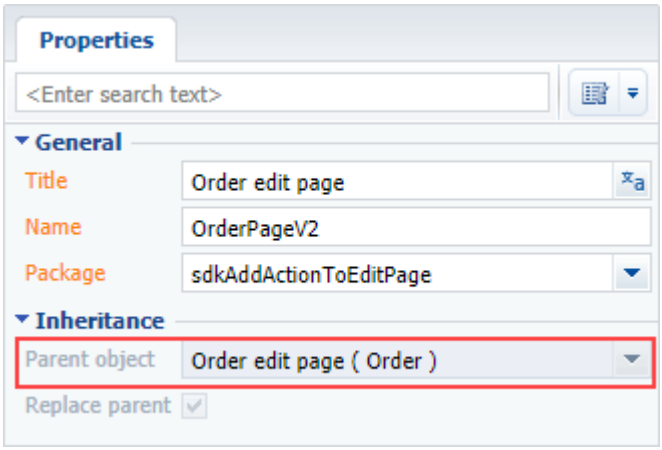
Use this link to download the case implementation package.

Case implementation algorithm

1. Create a replacing edit page for an order in a custom package

A replacing client module must be created and [Order edit page] (*OrderPageV2*) must be specified as the parent object in it (Fig. 1). The procedure of creating a replacing page is covered in the“**Creating a custom client module schema**” article.

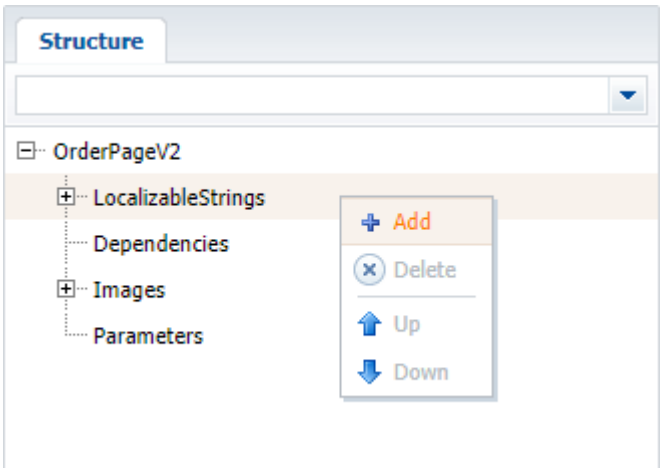
Fig. 1. Properties of the replacing edit page



2. Add a string with the [Actions] menu title to the localized string collection of the page replacing schema

Create a new localizable string (Fig. 2).

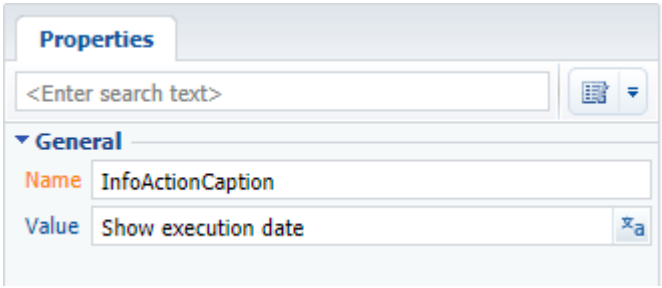
Fig. 2 – Adding the localized string to the schema



Populate the following values for the created string (Fig.3):

- [Name] – “InfoActionCaption".
- [Value] – “Show execution date”.

Fig. 3. Properties of the custom localizable string



3. Add the implementation of the following methods to the method collection of the page view model

- *isRunning()* – checks whether the order is at the [In progress] stage and defines availability of the added menu item.
- *showOrderInfo()* – the action handler method that displays the scheduled end date of the order in the message window.
- *getActions()* – an overridden parent schema method that gets the page action collection.

The replacing schema source code is as follows:

```
define("OrderPageV2", ["OrderConfigurationConstants"],
function(OrderConfigurationConstants) {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Order",
        // Methods of the edit page view model.
        methods: {
            // Method which checks the stage of the order execution to define
availability of menu item.
            isRunning: function() {
                // The method returns true if the order status is [In progress],
otherwise it returns false.
                if (this.get("Status")) {
                    return this.get("Status").value ===
OrderConfigurationConstants.Order.OrderStatus.Running;
                }
                return false;
            },
            // Action handler method which displays the order end date in the data
window.
            showOrderInfo: function() {
                // Receiving the order end date.
                var dueDate = this.get("DueDate");
                // Calling the standard system method for the data window display.
                this.showInformationDialog(dueDate);
            },
            // Override the base virtual method which returns the actions collection
of the edit page.
            getActions: function() {
                 // The parent implementation of the method is called to receive
                 // the initiated actions collection of the base page.
                var actionMenuItems = this.callParent(arguments);
                // Adding the separator line.
                actionMenuItems.addItem(this.getButtonMenuItem({
                    Type: "Terrasoft.MenuSeparator",
                    Caption: ""
                }));
                // Adding a menu item to the actions list of the edit page.
                actionMenuItems.addItem(this.getButtonMenuItem({
                    // Binding the title of the menu item to the localizable string
of the schema.
                    "Caption": {bindTo: "Resources.Strings.InfoActionCaption"},
                    // Binding the action handler method.
                    "Tag": "showOrderInfo",
                    // Binding the visibility property of the menu item to the value
returning the isRunning() method.
                    "Enabled": {bindTo: "isRunning"}
                }));
                return actionMenuItems;
            }
        }
    };
});
```

ATTENTION

The previous steps are enough to add an action to the edit page. But the custom page action will not be displayed in the vertical view of the list.

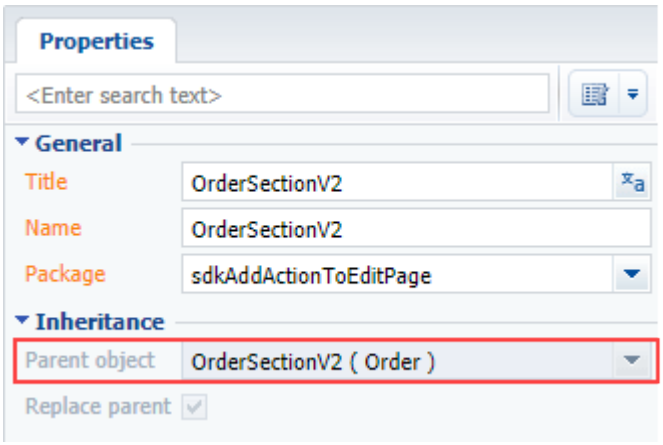
For correct displaying of the page action, add the localizable string of the title and a method that defines menu

item availability to the section view model schema.

4. Create a replacing schema of the [Orders] section

Create a replacing client module and specify the *OrderSectionV2* schema as parent object (Fig. 4). The procedure for creating a replacing page is described in the "**Creating a custom client module schema**" article.

Fig. 4. Properties of the [Orders] section replacing page



5. Add a localizable string with the [Actions] menu item title

For this purpose, repeat the actions in par. 2

6. Add the method implementation

Add the implementation of the *isRunning()* method to the view model collection of methods. The method will check whether the selected order is at the [In progress] stage and defines availability of the added menu item.

The replacing schema source code is as follows:

```
define("OrderSectionV2", ["OrderConfigurationConstants"],
function(OrderConfigurationConstants) {
    return {
        // Name of the section schema.
        entitySchemaName: "Order",
        // Methods collection of the section view model.
        methods: {
            // Method checking the stage of the selected order to determine the menu
item visibility.
            isRunning: function(activeRowId) {
                activeRowId = this.get("ActiveRow");
                //  Receiving the data collection of the section list view.
                var gridData = this.get("GridData");
                // Receiving the model of the selected order by a value in the
primary column.
                var selectedOrder = gridData.get(activeRowId);
                // Receiving the model property - selected order status.
                var selectedOrderStatus = selectedOrder.get("Status");
                // Value of the selected order status is compared to the value of the
[In-progress] type and
                    // returns true or false depending on the comparison result.
                return selectedOrderStatus.value ===
OrderConfigurationConstants.Order.OrderStatus.Running;
            }
        }
    };
});
```

After you save the schema and update the application page with clearing the cache, a new action will appear on the order edit page when selecting an order at the [In progress] stage (Fig. 5,6).

Fig. 5. Case result. The order is at the [Completed] stage and the action is inactive

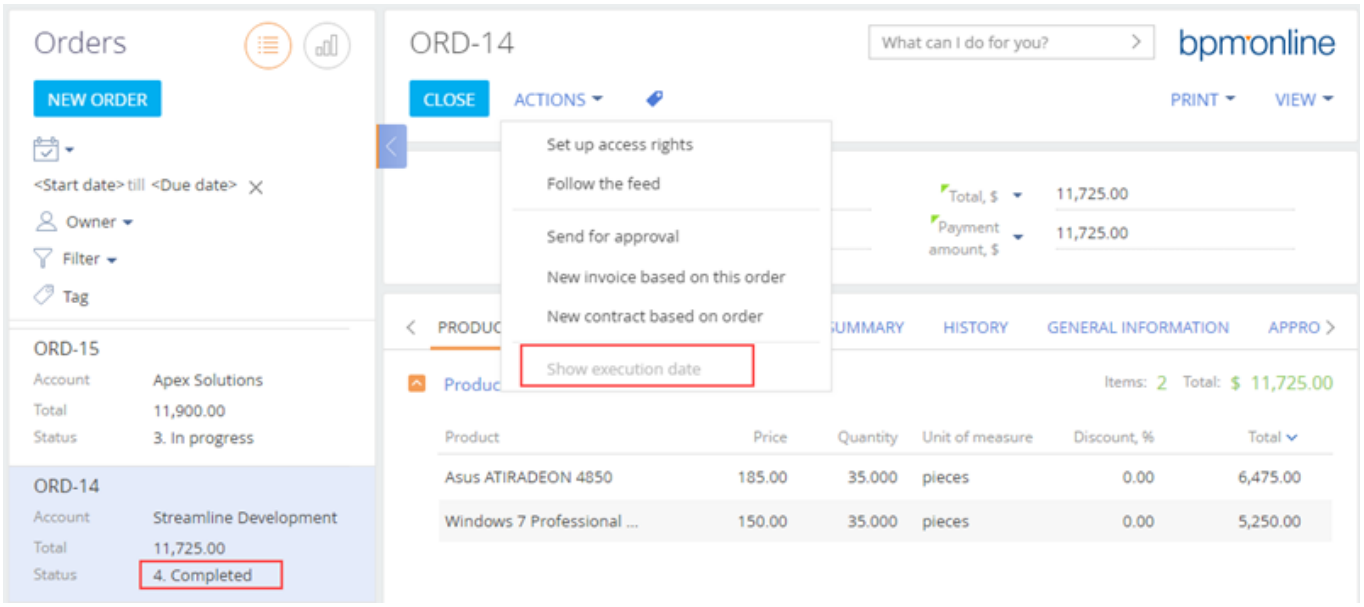
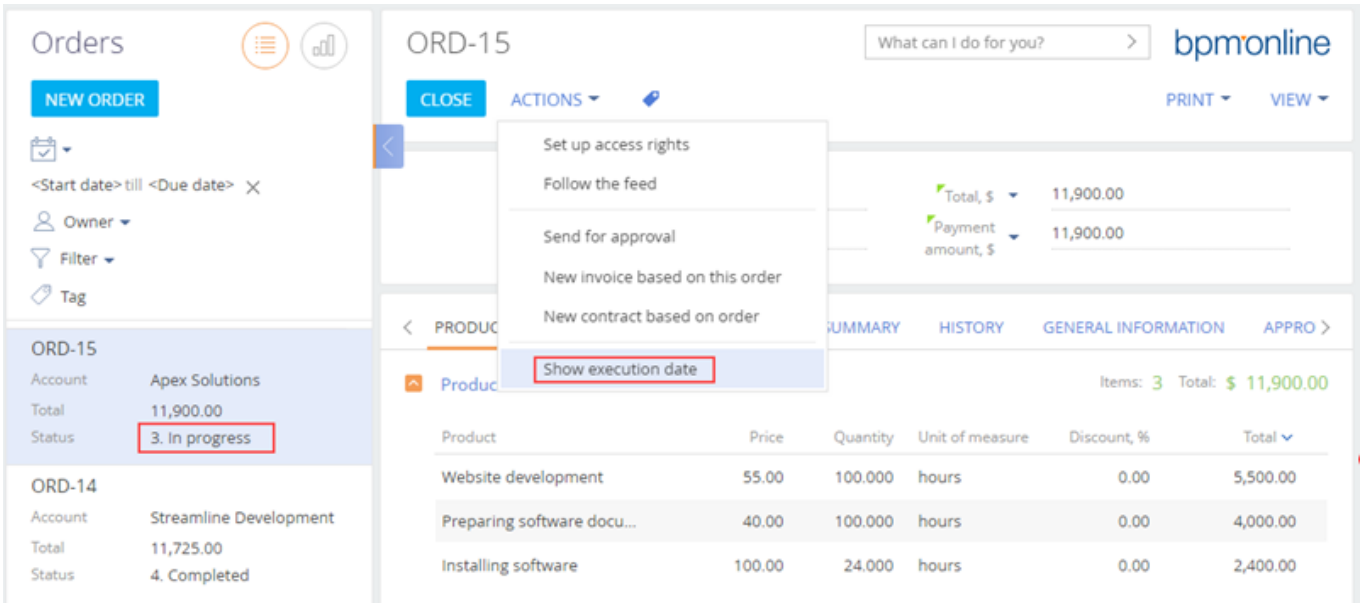


Fig. 6. Case result. The order is at the [In progress] stage and the action is active

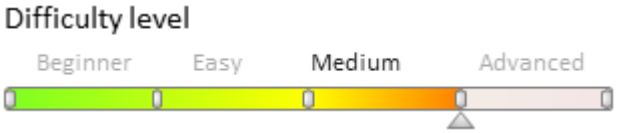


Control elements

Contents

- **Adding a new field to the edit page**
- **Adding a button to the edit page**
- **How to add a field with an image to the edit page**
- **How to add the color select button to the edit page**
- **How to add multi-currency field**
- **How to add custom logic to the existing controls**

Adding a new field to the edit page



Introduction

You can add fields to the edit page in two ways:

1. Via section wizard (see the "Section wizard", "How to set up page fields” articles).

A base object (for example, the [Activity] object) replacing schema and a base edit page (for example, the *ActivityPageV2*) replacing schema will be created in the custom package as a result of the section wizard operation. A new column will be added in the object replacing schema. A configuration object with the settings of the new field location on a page will be added to the *diff* array in the edit page replacing schema.

> NOTE
>
> When creating new sections via the wizard, it will create new schemas instead of replacing schemas in your current custom package.

You should implement additional edit page business logic or develop new client interface elements in the created edit page replacing schema.

2. Via creating replacing base object and replacing base page by developer tools.

Source code

You can download the package with case implementation using the following link.

Example 1

Case description

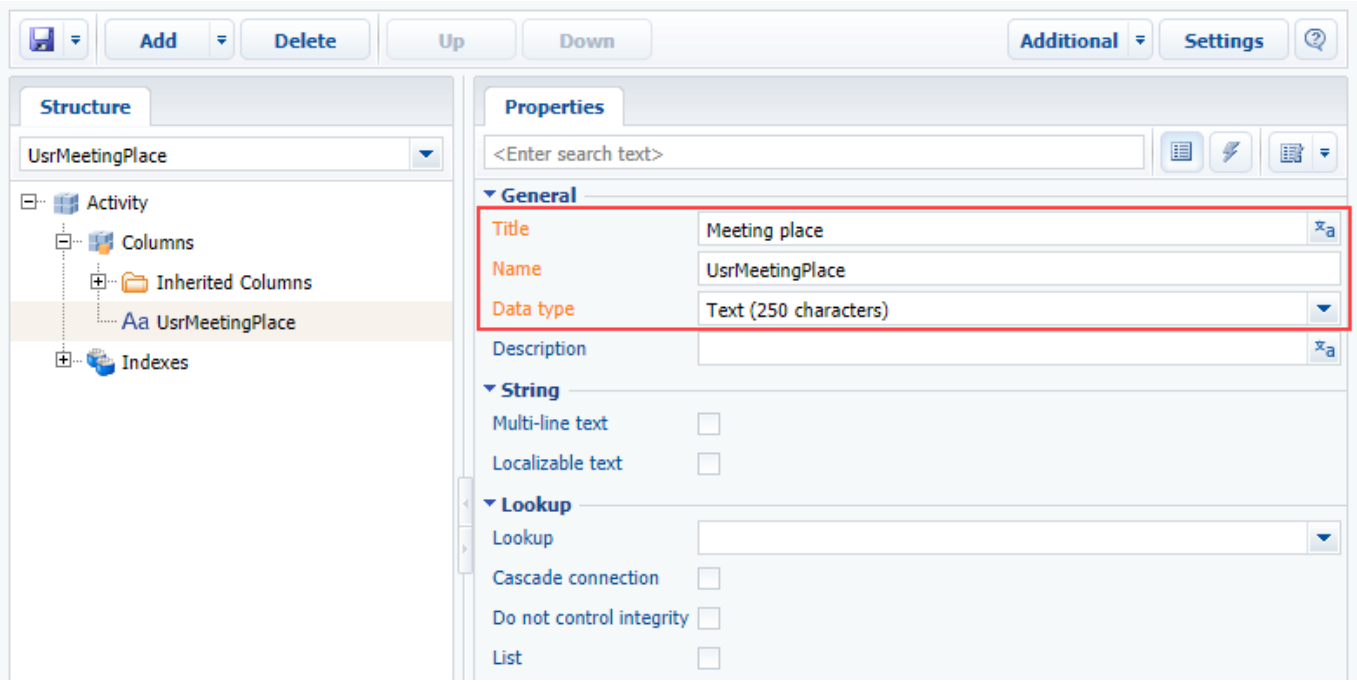
Manually add a new [Meeting place] field to the activity edit page.

Case implementation algorithm

1. Create a replacing object and add a new column to it.

Create an [Activity] replacing object and add the new [Meeting place] column of the “string” type to it (Fig. 1). Learn more about creating a replacing object schema in the “**Creating the entity schema**” article.

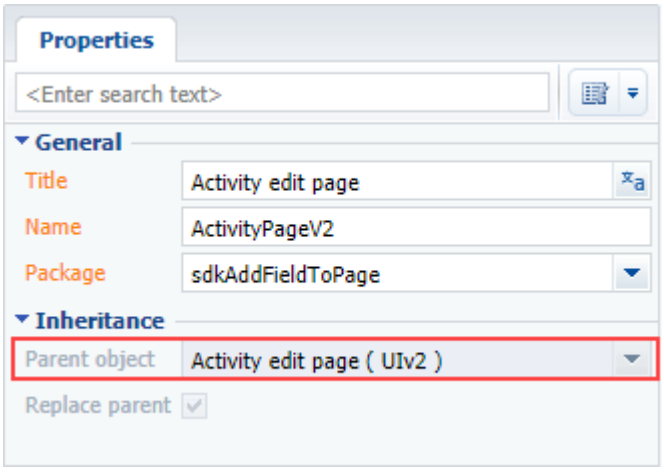
Fig. 1. Adding a custom column to the replacing object



2. Create a replacing client module for the activity page

Create a replacing client module and specify the [Activity edit page] schema as parent object (Fig. 2). The procedure of creating a replacing page is covered in the “**Creating a custom client module schema**” article.

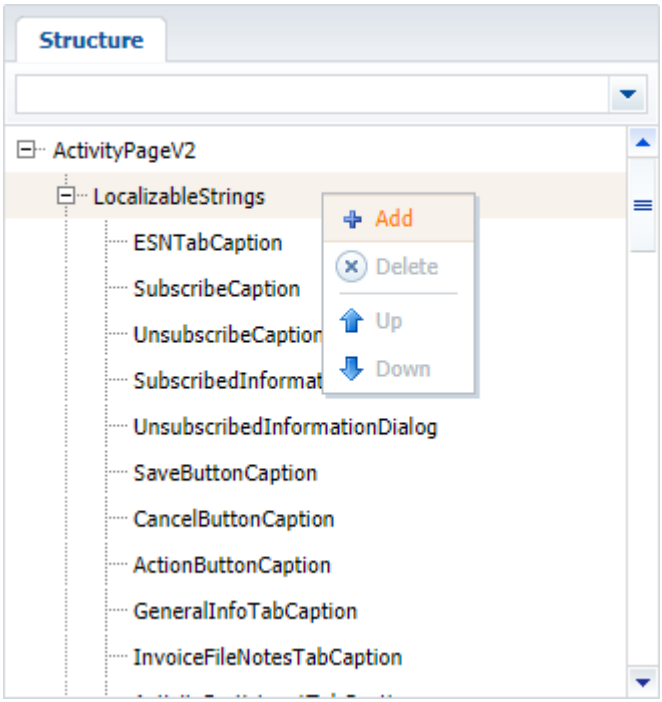
Fig. 2. Replacing edit page properties



3. Add a localized string with the field caption.

Add a string containing the added field caption to the localized string collection of the replacing page schema (fig.3).

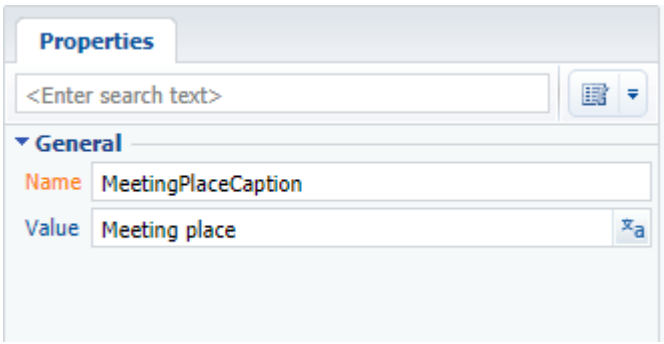
Fig. 3. Adding localized string to the schema



For the created string specify (Fig. 4):

- [Name] – "MeetingPlaceCaption";
- [Value] – “Meeting place”.

Fig. 4. Properties of the custom localized string



4. Add a new field to the activity edit page.

Add a configuration object containing the settings of the [Meeting place] field location on the page to the diff array.

More information about the *diff* array properties is available in the "**The "diff" array**” article.

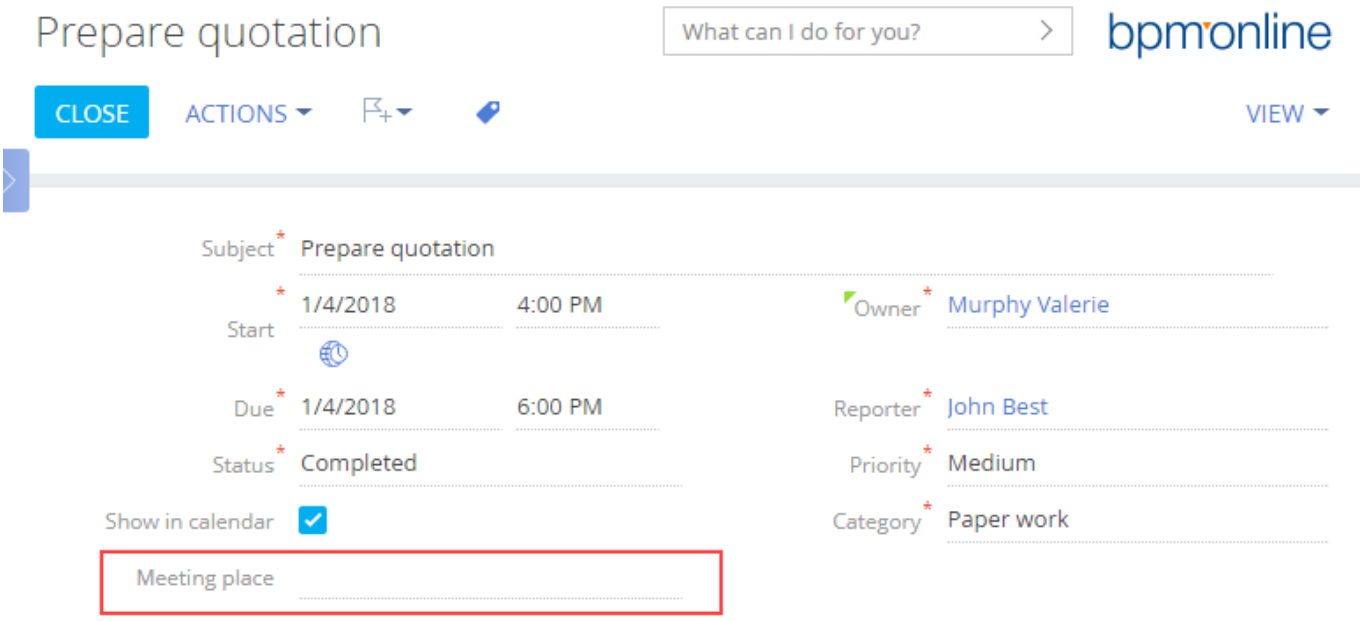
The replacing schema source code is as follows:

```
define("ActivityPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Activity",
        // Displaying of a new field on the edit page.
        diff: /**SCHEMA_DIFF*/[
            // Meta data for adding the [Meeting place] field.
            {
                // Operation of adding a component to the page.
                "operation": "insert",
                // Meta name of a parent container where a field is added.
                "parentName": "Header",
                // The field is added to the component collection
```

```
                // of a parent element.
                "propertyName": "items",
                // The name of a schema column that the component is linked to.
                "name": "UsrMeetingPlace",
                "values": {
                    // Field caption.
                    "caption": {"bindTo": "Resources.Strings.MeetingPlaceCaption"},
                    // Field location.
                    "layout": {
                        // Column number.
                        "column": 0,
                        // String number.
                        "row": 5,
                        // Span of the occupied columns.
                        "colSpan": 12
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

After you save the schema and update the application page with clearing the cache, you will see a new field appear on the activity edit page (fig.5).

Fig. 5. Case result demonstration



Example 2

Case description

Manually add a [Country] field to the contact profile edit page. The difference of this case is that you already have the [Country] column in your object schema.

Case implementation algorithm

1. Create a replacing contact page

Create a replacing client module and specify the [Display schema — Contact card], *ContactPageV2* schema as parent object. The procedure of creating a replacing page is covered in the"**Creating a custom client module schema**"

article.

2. Add the [Country] field to the contact profile.

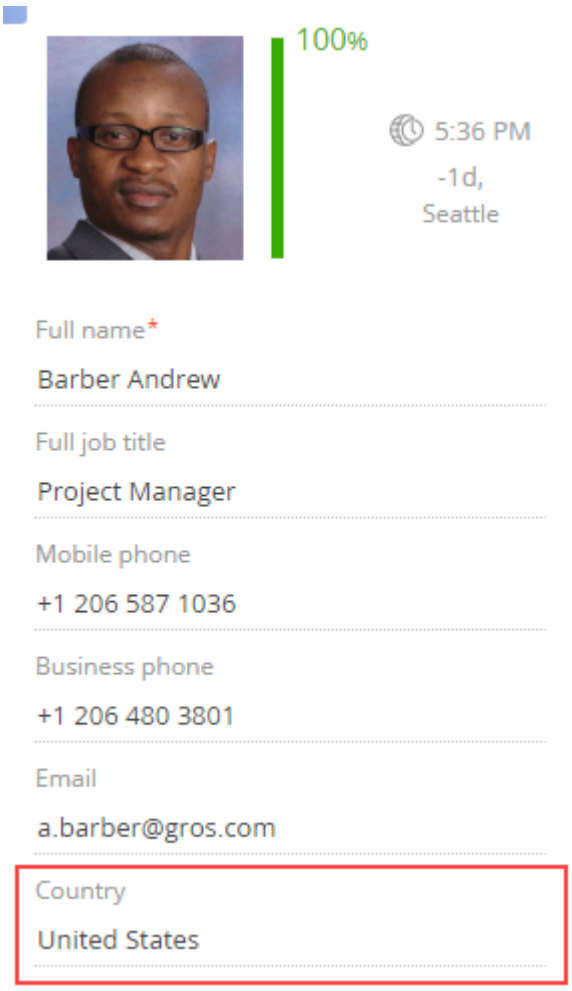
Add a configuration object containing the field property settings to the diff array. Indicate the *ProfileContainer*. element as a parent schema element where the field will be located.

The replacing schema source code is as follows:

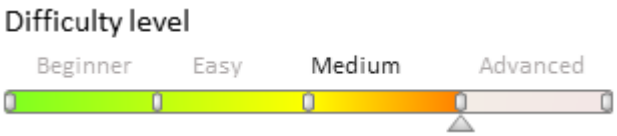
```
define("ContactPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Contact",
        diff: [
            // Meta data for adding the [Country] field.
            {
                // Operation of adding a component to the page.
                "operation": "insert",
                // Meta name of a parent container where a field is added.
                "parentName": "ProfileContainer",
                // The field is added to the component collection
                // of a parent element.
                "propertyName": "items",
                // The name of a schema column that the component is linked to.
                "name": "Country",
                "values": {
                    // Field type — lookup.
                    "contentType": Terrasoft.ContentType.LOOKUP,
                    // Field location.
                    "layout": {
                        // Column number.
                        "column": 0,
                        // String number.
                        "row": 6,
                        // Span of the occupied columns.
                        "colSpan": 24
                    }
                }
            }
        ]
    };
});
```

After you save the schema and update the application page with clearing the cache, you will see a new field appear on the contact edit page (fig.6).

Fig. 6. Case result demonstration



Adding a button to the edit page



To add a custom button to the view model on the edit page, two properties must be modified:

- Methods collection. The implementation of the handler method to be called when the button is clicked must be added to the methods collection. Other auxiliary methods required for the control item functioning must be also added. These may be methods for regulating the visibility or availability of the control item depending on conditions.
- diff configuration objects array. Add a configuration object in the diff configuration objects array to set up the visual location of the control item on the edit page.

> ⚠ NOTE
>
> To display the button on the page in the existing record edit mode, the section schema must be modified.
>
> To display the button in the new record addition mode, similar modifications are made in the schema of the page view model itself.

DOM model of standard page buttons

The html-container hierarchical structure is used to locate standard functional buttons of the bpm'online edit page.

CombinedModeActionButtonsCardContainer is a top level container on the existing record edit page. Two more containers are inside it:

- CombinedModeActionButtonsCardLeftContainer - where the Save, Cancel and Actions standard buttons are located;
- CombinedModeActionButtonsCardRightContainer - where the Print and View buttons are located.

Similarly, for the new record edit page: ActionButtonsContainer — a top level container.

Two more containers are inside it:

- Leftcontainer - where the Save, Cancel and standard Actions buttons are located;
- RightContainer - where the Print and View buttons are located.

Depending on the exact position required for a button, the corresponding container is specified when setting the button visualization in the diff array.

NOTE

Meta-names of html-containers are used here.

These names are specified when setting the control item visualization in the configuration object of the diff array.

The actual ID of corresponding html-items of the page are formed by the system automatically, based on such meta-names.

Setting the button visualization properties

To set the visual location of a custom button on the edit page, a configuration object with the following properties must be added to the *diff* array of the view model:

Property	Description
operation	A type of operation with a control item (*insert, move, remove, merge, set*). This is set by a string with the corresponding operation name. The *insert* value is specified to add a new control item.
parentName	The meta-name of the parent control item where a custom item is to be placed. If this is a functional button, *LeftContainer* and *RightContainer* may act as the parent containers.
propertyName	The *items* value is specified for the custom control item.
name	The meta-name of the added control item.
values	A configuration object with settings of supplementary properties for a control item.

Setting properties of the *values* object:

Property	Description
itemType	A type of an item. This is set by a value of the *Terrasoft.ViewItemType* list. The *BUTTON* value is used for the button.
caption	The button title. It is recommended to set title values by binding to a localizable string of the schema.
click	Binding of the button handler method.
layout	The object of setting a control item location on the grid.
enabled	Controls the the button availability (activity).
visible	Controls the button visibility.
style	Component style. The property should contain the value of the

Property	Description
	Terrasoft.controls.ButtonEnums.style enumeration.

The *Terrasoft.controls.ButtonEnums.style* enumeration contains following values:

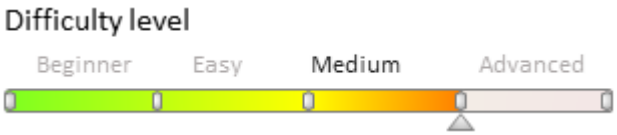
Property	Description
Terrasoft.controls.ButtonEnums.style.DEFAULT	Default style.
Terrasoft.controls.ButtonEnums.style.GREEN	Green color button.
Terrasoft.controls.ButtonEnums.style.RED	Red color button.
Terrasoft.controls.ButtonEnums.style.BLUE	Blue color button.
Terrasoft.controls.ButtonEnums.style.GREY	Transparent button. This value is from previous versions of bpm'online.
Terrasoft.controls.ButtonEnums.style.TRANSPARENT	Transparent button.

You can read more about the diff array in the "**The "diff" array**" article.

Examples of implementing a button adding to the edit page

- **How to add the button on the edit page in the combined mode**
- **How to add a button to an edit page in the new record add mode**
- **How to add a button to a section**

How to add a button to an edit page in the new record add mode



Case description

The button opening the primary contact edit page must be added to the new account add page.

ATTENTION

By default a pop-up summary is used to add new account. To use the page to add an account select the [Default value] checkbox in the [*Enable account mini page add mode*] system setting . In order for the system setting change to take effect, you must log off and log on to the user.

To complete these case, you need to use the page to add the account.

NOTE

The edit page mode can be accessed at the creation of the record and when the page is refreshed in the combined mode by pressing F5 key. The vertical list should be disabled.

Source code

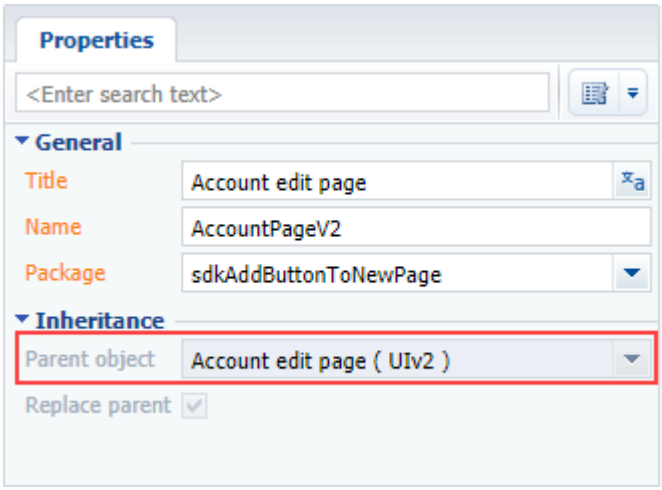
Use this link to download the case implementation package.

Case implementation algorithm

1. Create a replacing account edit page

A replacing client module must be created and [AccountPageV2] must be specified as the parent object in it (Fig. 1). The procedure of creating a replacing page is covered in the"**Creating a custom client module schema**" article.

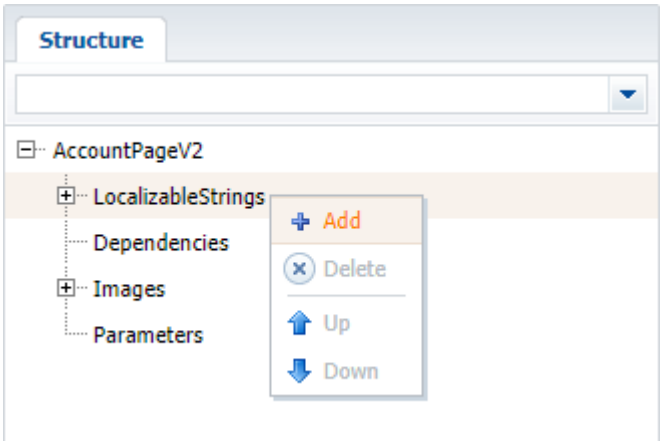
Fig. 1. Properties of the replacing edit page



2. Add a string with the button title to the collection of localizable strings of the page replacing schema

Create a new localizable string (Fig. 2).

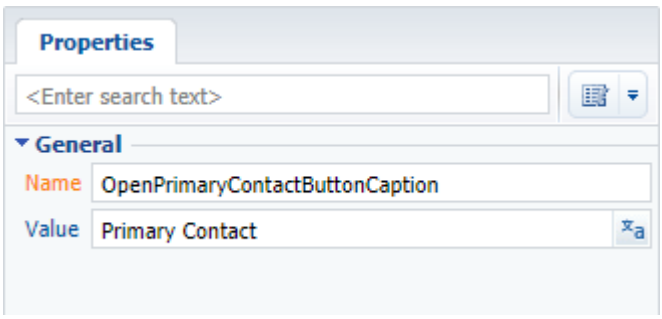
Fig. 2. Adding localized string to the schema



For the created string specify (Fig. 3):

- [Name] – "OpenPrimaryContactButtonCaption".
- [Value] – "Primary Contact".

Fig. 3. Properties of the custom localizable string



3. Add the implementation of the following methods to the method collection of the

page view model

- *isAccountPrimaryContactSet()* – checks if the [Primary contact] field is filled.
- *onOpenPrimaryContactClick()* – button pressing handler method which performs passing to the base contact edit page.

4. Add a button on the edit page

Add an object with the settings determining the button position on the account edit page in the *diff* array.

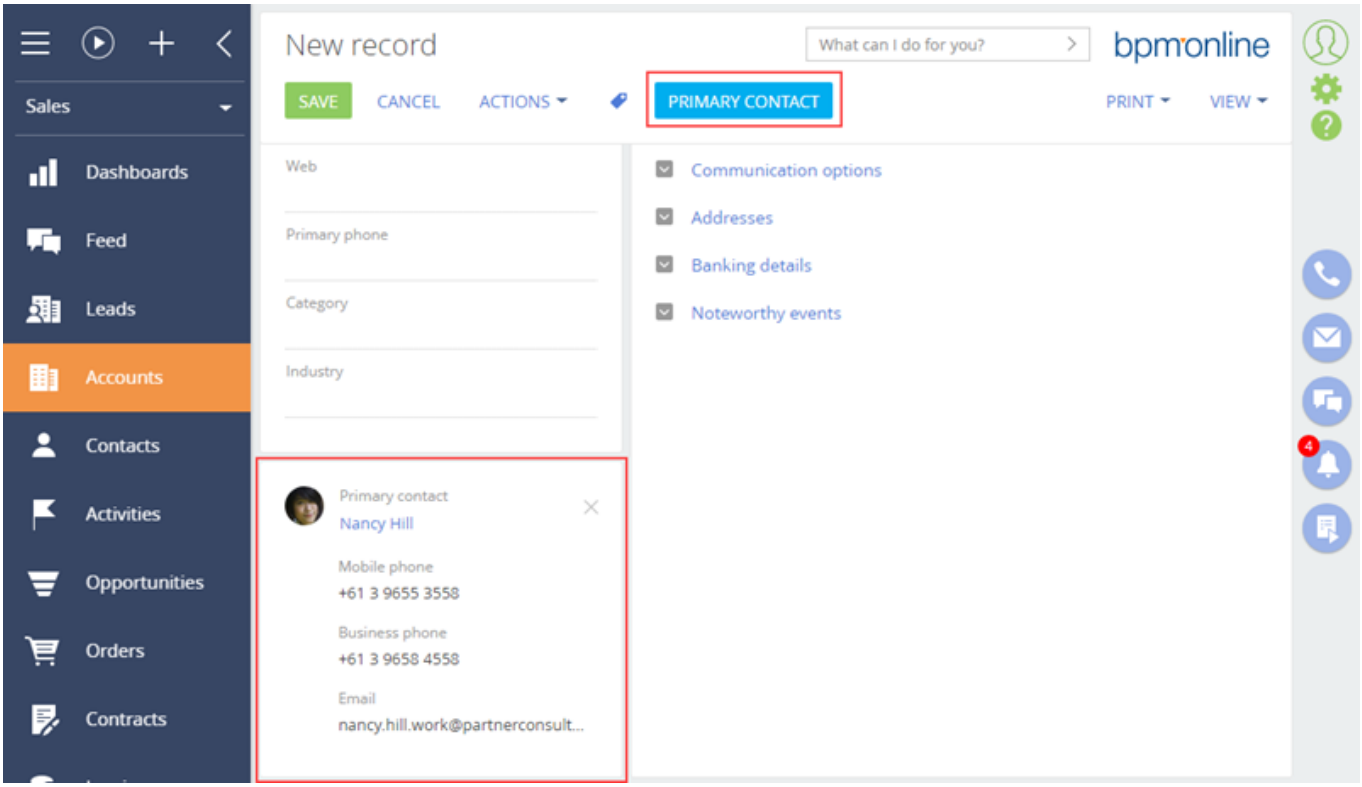
The replacing schema source code is as follows:

```
define("AccountPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Account",
        // Methods collection of the edit page view model.
        methods: {
            // The method checks whether the [Primary contact] field is completed.
            isAccountPrimaryContactSet: function() {
                return this.get("PrimaryContact") ? true : false;
            },
            // Button press handler method.
            onOpenPrimaryContactClick: function() {
                var primaryContactObject = this.get("PrimaryContact");
                if (primaryContactObject) {
                    // Determining the base contact Id.
                    var primaryContactId = primaryContactObject.value;
                    // Forming the address string.
                    var requestUrl = "CardModuleV2/ContactPageV2/edit/" +
primaryContactId;
                      // Publishing a message and going to the
                      // base contact edit page.
                    this.sandbox.publish("PushHistoryState", {
                        hash: requestUrl
                    });
                }
            }
        },
        // Displaying a button on the edit page.
        diff: [
            // Metadata for adding a new control item - custom button to the page.
            {
                // Indicates that an operation of adding an item to the page is being
executed.
                "operation": "insert",
                // MMetadata of the parent control item the button is added.
                "parentName": "LeftContainer",
                 // Indicates that the button is added to the control items
collection
                 // of the parent item (which meta-name is specified in the
parentName).
                "propertyName": "items",
                // Meta-name of the added button.
                "name": "PrimaryContactButton",
                // Supplementary properties of the item.
                "values": {
                    // Type of the added item is button.
                    itemType: Terrasoft.ViewItemType.BUTTON,
                    //  Binding the button title to a localizable string of the
schema..
                    caption: {bindTo:
```

```
"Resources.Strings.OpenPrimaryContactButtonCaption"},
                        // Binding the button press handler method.
                        click: {bindTo: "onOpenPrimaryContactClick"},
                        // Binding the property of the button availability.
                        enabled: {bindTo: "isAccountPrimaryContactSet"},
                        // Setting the button style.
                        "style": Terrasoft.controls.ButtonEnums.style.BLUE
                    }
                }
            ]
        };
});
```

When the schema is saved and the system web-page is updated, the [Primary contact] button will appear on the new account create page. The button will be activated after filling the [Primary contact] field in the account (Fig. 4).

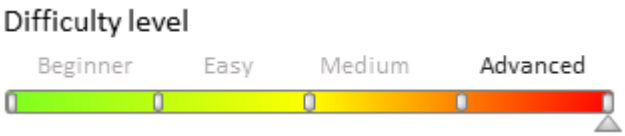
Fig. 4. Demonstrating the case implementation result



See also

- **Adding a button to the edit page**
- **How to add the button on the edit page in the combined mode**

How to add the button on the edit page in the combined mode



Example of implementing the button adding to an edit page

Case description

The button opening the base contact edit page must be added to the account edit page.

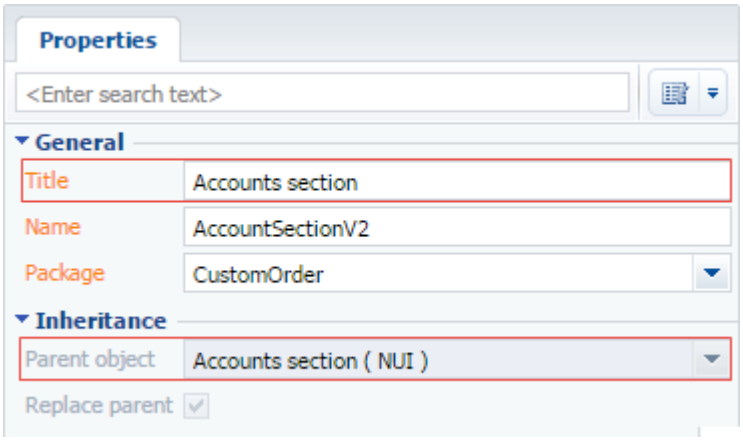
Case implementation algorithm

1. Create the [Accounts] section replacing schema

A replacing client module must be created and [Accounts section] must be specified as the parent object (Fig. 1).

The procedure for creating the replacing page is described in the article **Creating a custom client module schema**.

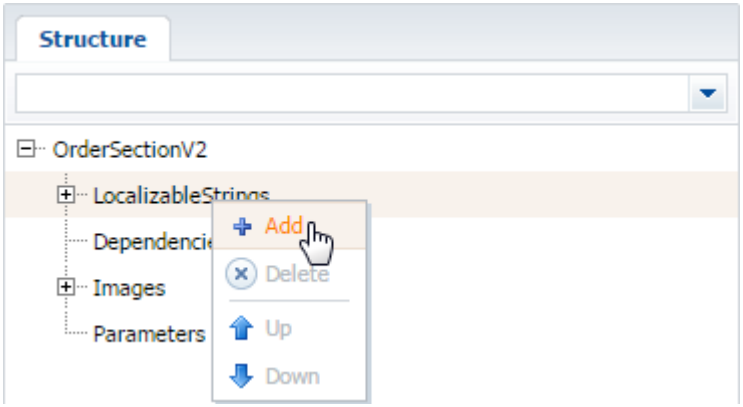
Fig. 1. — Properties of the section replacing page



2. Add a string with the button name to the localizable strings collection of the section replacing schema

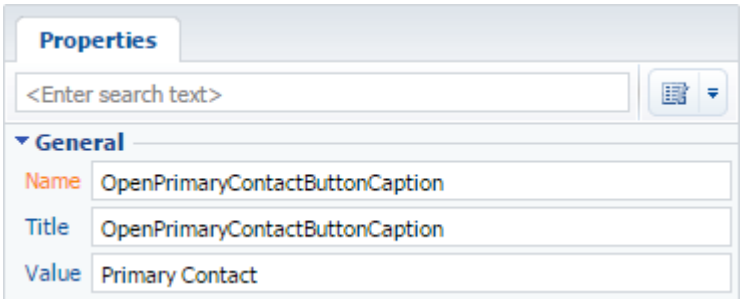
For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure node (Fig. 2).

Fig. 2. — Adding a localizable string to the schema



Fill properties for the created line as shown in Fig. 3.

Fig. 3. — Properties of a custom localized string



3. Add the method implementation to the methods collection of the view model

- isAccountPrimaryContactSet - checks whether the [Primary contact] field of the page is filled;
- onOpenPrimaryContactClick - button pressing handler method which goes to the base contact edit page.

For this purpose, add the program code of the section replacingmodule to the source code tab. Required methods are added to the methods collection of the view model.

```
define("AccountSectionV2", [],
    function() {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Account",
            // Methods collection of the edit page view model.
            methods: {
                // Button press handler method.
                onOpenPrimaryContactClick: function() {
                    var activeRow = this.get("ActiveRow");
                    if (activeRow) {
                        // Determining the base contact Id.
                        var primaryId =
this.get("GridData").get(activeRow).get("PrimaryContact").value;
                        if (primaryId) {
                            // Forming the address string.
                            var requestUrl = "CardModuleV2/ContactPageV2/edit/" +
primaryId;
                            // Publishing the message and going to the
                            // base contact edit page.
                            this.sandbox.publish("PushHistoryState", {
                                hash: requestUrl
                            });
                        }
                    }
                },
                // The method checks whether the [Base Contact] field is filled.
                isAccountPrimaryContactSet: function() {
                    debugger;
                    var activeRow = this.get("ActiveRow");
                    if (activeRow)
                    {
                        var pc =
this.get("GridData").get(activeRow).get("PrimaryContact");
                        return (pc || pc !== "") ? true : false;
                    }
                    return false;
                }
            }
        };
    });
```

4. Add a configuration object with button location settings on the edit page to the diff array

```
define("AccountSectionV2", [],
    function() {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Account",
            // Methods collection of the edit page view model.
            methods: {
                // onOpenPrimaryContactClick, isAccountPrimaryContactSet method
implementation
            },
            // Setting a button visualization on the edit page.
            diff: [
```

```
                // Metadata for adding a cutom button to the page.
                {
                    // Indicates that an operation of adding an item to the page is
being executed.
                    "operation": "insert",
                    // Meta-name of the parent control item where the button is
added.
                    "parentName": "CombinedModeActionButtonsCardLeftContainer",
                    // Indicates that the button is added to the control items
collection
                    // of the parent item (which name is specified in the
parentName).
                    "propertyName": "items",
                    // Meta-name of the added button. .
                    "name": "MainContactButton",
                    // Supplementary properties of the item.
                    "values": {
                        // Type of the added item is button.
                        itemType: Terrasoft.ViewItemType.BUTTON,
                        // Binding the button title to a localizable string of the
schema.
                        caption: {bindTo:
"Resources.Strings.OpenPrimaryContactButtonCaption"},
                        // Binding the button press handler method.
                        click: {bindTo: "onOpenPrimaryContactClick"},
                        // Binding the property of the button availability.
                        enabled: {bindTo: "isAccountPrimaryContactSet"},
                        // Setting the field location.
                        "layout": {
                            "column": 1,
                            "row": 6,
                            "colSpan": 1
                        }
                    }
                }
            ]
        };
    });
```

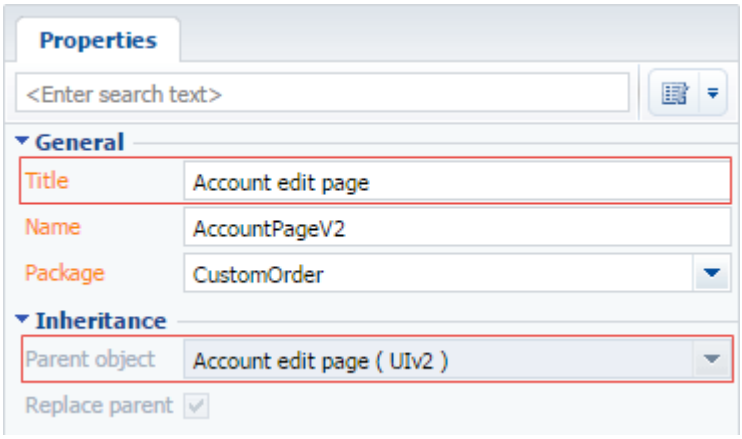
5. Save the created replacingpage schema

6. Create a replacingaccount edit page

A replacing client module must be created and [Account edit page] must be specified as the parent object (Fig. 4).

The procedure for creating the replacingpage is described in the article **Creating a custom client module schema**.

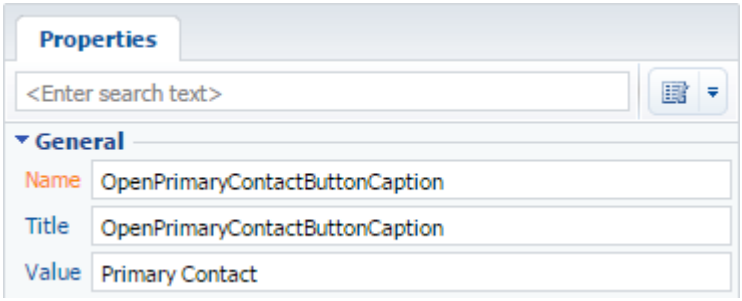
Fig. 4. — Properties of the replacing edit page



7. Add a string with the button title to the localizable strings collection of the replacing page schema

Fill in properties for the created string as shown on Fig. 5.

Fig. 5. — Properties of a custom localized string



8. Add the method implementation to the methods collection of the page view model

- isAccountPrimaryContactSet – checks whether the [Base Contact] field of the page is filled;
- onOpenPrimaryContactClick – button pressing handler method which goes to the base contact edit page.

For this purpose, add the program code of the replacingmodule of the page to the source code tab. Required methods are added to the methods collection of the view model.

```
define("AccountPageV2", [],
    function() {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Account",
            // Methods collection of the edit page view model.
            methods: {
                // The method checks whether the [Base Contact] field is filled.
                isAccountPrimaryContactSet: function() {
                    return this.get("PrimaryContact") ? true : false;
                },
                // Button press handler method.
                onOpenPrimaryContactClick: function() {
                    // Determining the base contact Id.
                    var primaryId = this.get("PrimaryContact").value;
                    if (primaryId) {
                        // Forming the address string.
                        var requestUrl = "CardModuleV2/ContactPageV2/edit/" +
primaryId;
                        // Publishing a message and going to the
                        // base contact edit page.
                        this.sandbox.publish("PushHistoryState", {
```

```
                            hash: requestUrl
                        });
                    }
                }
            }
        };
    });
```

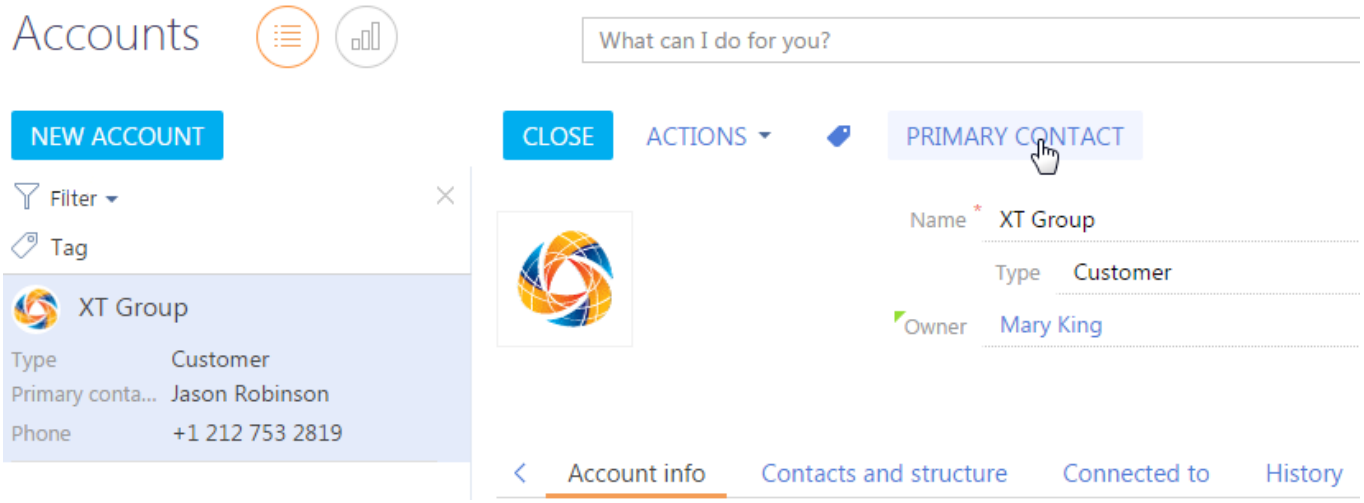
9. Add a configuration object with settings of a button location on a page to the diff array

```
define("AccountPageV2", [],
    function() {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Account",
            // Methods collection of the edit page view model.
            methods: {
                // onOpenPrimaryContactClick, isAccountPrimaryContactSet method
implementation
            },
            // Setting a button visualization on the edit page.
            diff: [
                // Metadata for adding a new control item - custom button - to the
page.
                {
                    // Indicates that an operation of adding an item to the page is
being executed.
                    "operation": "insert",
                    // Meta-name of the parent control item where the button is
added.
                    "parentName": "LeftContainer",
                    // Indicates that the button is added to the control items
collection
                    // of the parent item (whose name is specified in the
parentName).
                    "propertyName": "items",
                    // Meta-name of the added button.
                    "name": "MainContactButton",
                    // Supplementary properties of the item.
                    "values": {
                        // Type of the added item is button.
                        itemType: Terrasoft.ViewItemType.BUTTON,
                        // Binding the button title to a localizable string of the
schema.
                        caption: {bindTo:
"Resources.Strings.OpenPrimaryContactButtonCaption"},
                        // Binding the button press handler method.
                        click: {bindTo: "onOpenPrimaryContactClick"},
                        // Binding the property of the button availability.
                        enabled: {bindTo: "isAccountPrimaryContactSet"},
                        // Setting the field location.
                        "layout": {
                            "column": 1,
                            "row": 6,
                            "colSpan": 1
                        }
                    }
                }
            ]
        };
    });
```

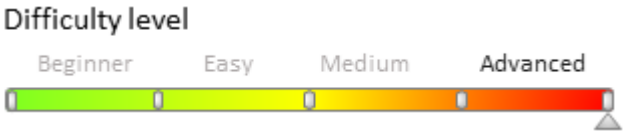
10. Save the created replacing page schema

When the schema is saved and the system web-page is updated, the [Base Contact] button will appear on the account edit page. The button will be activated if the [Base Contact] field of the account is filled (Fig. 6).

Fig. 6. – Demonstrating the case implementation result



How to add a field with an image to the edit page



Introduction

There are certain peculiarities of adding a field with image (contact's photo, product picture, account logo, etc.) to an edit page:

1. The object column used for a field with image should have the "Link to image" type. Specify it as the [Image] system column of the object.
2. Add the default image to the edit page image schema collection.
3. A field with image is added to the *diff* edit page schema array with usage of the additional *image-edit-container* CSS-class container-wrapper.
4. The *values* property of the configuration object containing settings of a field with image must include the following properties:
   - *getSrcMethod* – method receiving image by link
   - *onPhotoChange* – method called upon image modification
   - *Readonly* – property defining the capability of image editing (changing, deleting)
   - *Generator* – control element generator. Indicate the *ImageCustomGeneratorV2.generateCustomImageControl* for the field with image
   - *beforeFileSelected* – method called before opening the image selection dialog box
5. Add the following to the edit page schema collection:
   - method receiving image by link
   - method called upon image modification
   - method saving the link to a modified image in the object column
   - method called before opening the image selection dialog box

Case description

Adding a field with logo to the knowledge base article edit page.

Source code

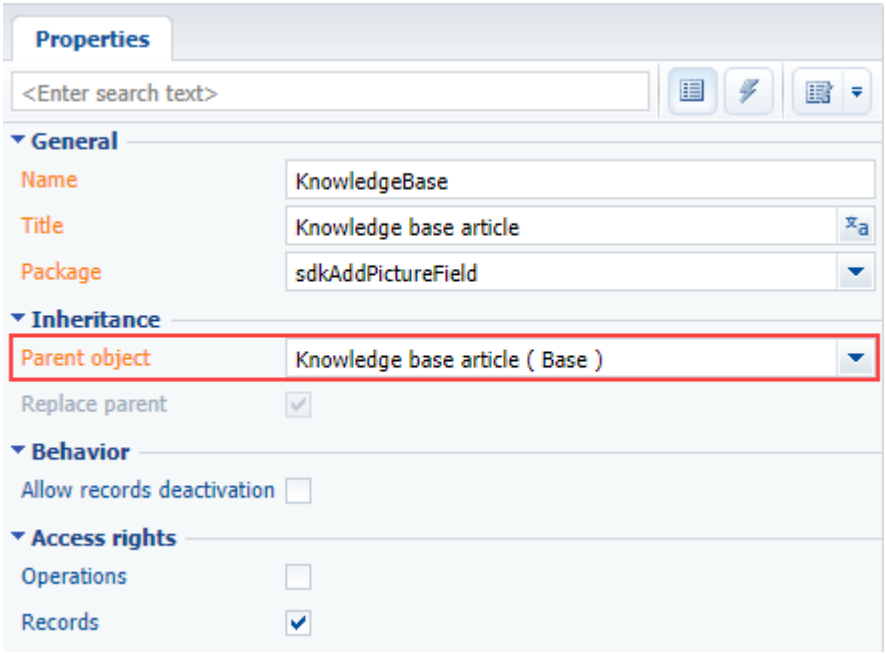
You can download the package with case implementation using the following link..

Case implementation algorithm

1. Creating the [Knowledge base] replacing object.

Create the [Knowledge base article] replacing object (Fig.1). Learn more about creating a replacing object in the “**Creating the entity schema**” article.

Fig. 1. Properties of the object replacing schema

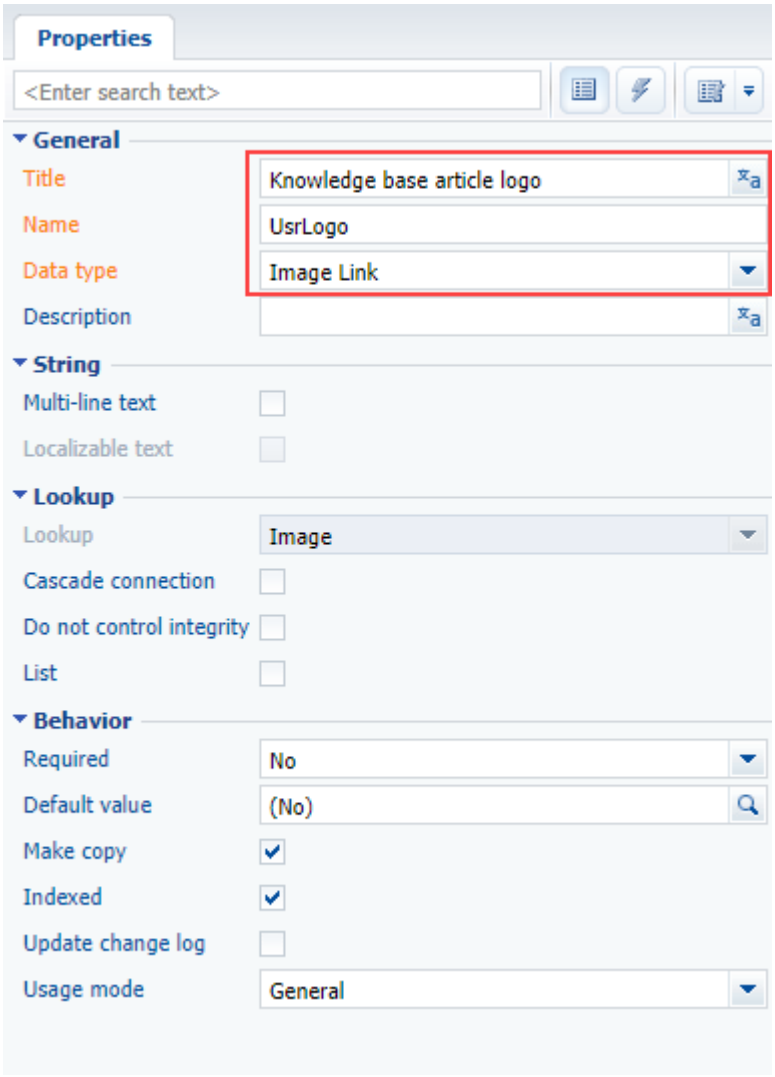


2. Adding a new column to the replacing object.

For the created column specify (Fig. 2):

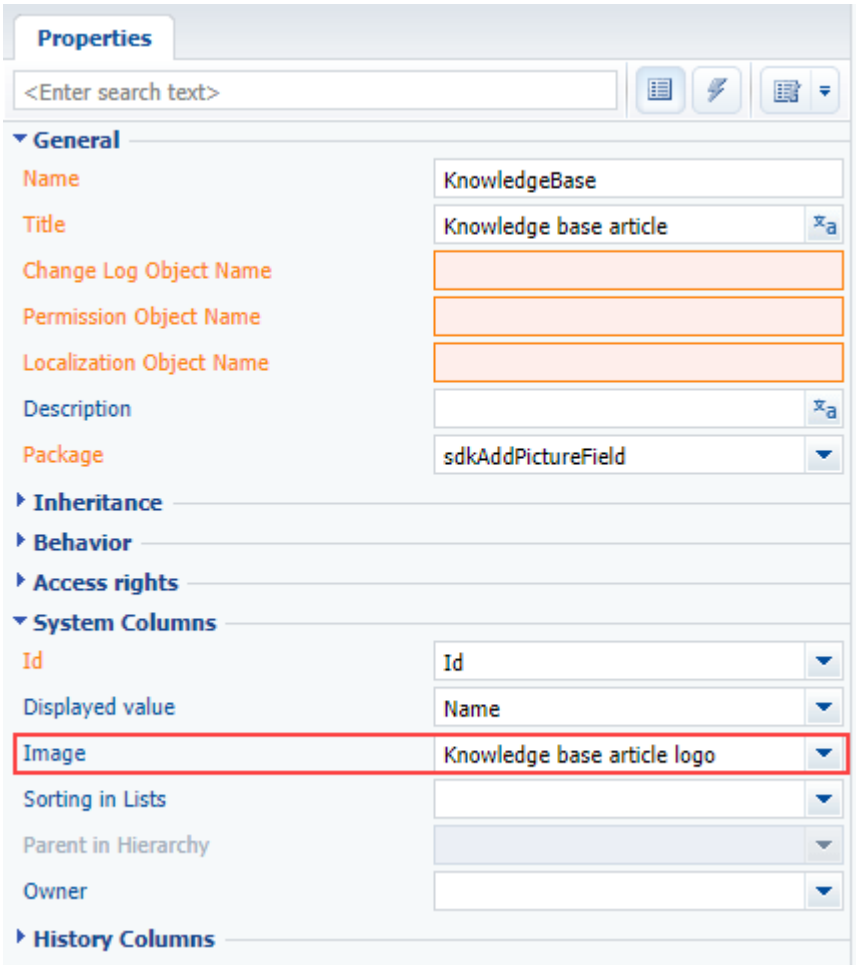
- [Title] – "Knowledge base article logo”
- [Name] – "UsrLogo”
- [Data type] – "Image Link"

Fig. 2. Adding a custom column to the replacing object



Indicate the created column as the [Image] object system column (Fig.3).

Fig. 3. Setting up the created column as the system column



NOTE

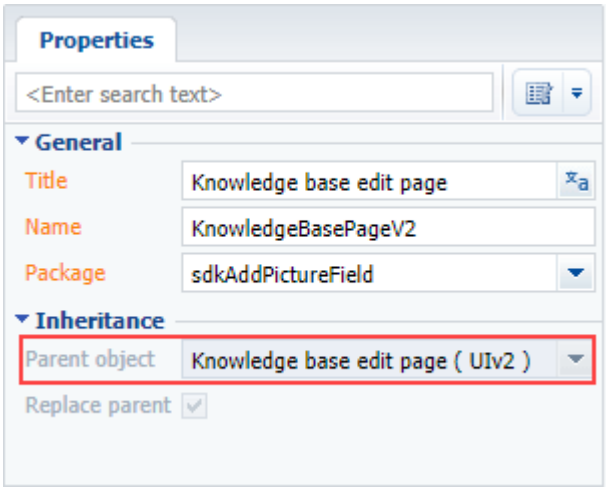
To view all object properties, switch to the object property advanced view mode. You can learn more about object designer capabilities in the "**Workspace of the Object Designer**” article.

Save and publish the object schema after you set up all properties.

3. Creating a replacing client module for the edit page.

Create a replacing client module and specify the [Knowledge base edit page] (*KnowledgeBasePageV2*) as the parent object in it (Fig. 4). The procedure of creating a replacing page is covered in the“**Creating a custom client module schema**” article.

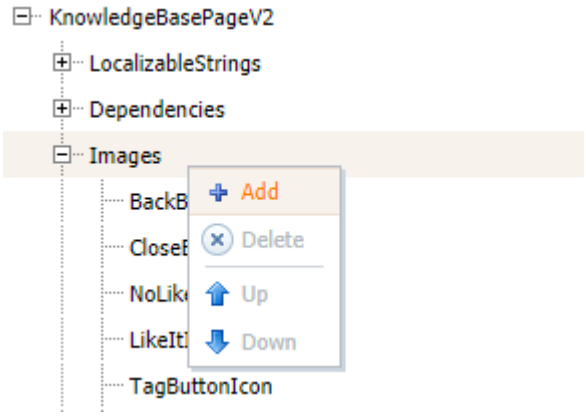
Fig. 4. Properties of the replacing edit page



4. Adding a default image to the [Images] resources of the edit page schema.

Add the default image to the page replacing schema image collection (Fig.5).

Fig. 5. Adding default image to the image schema resources



For the created image specify (Fig. 6):

- [Name] – "DefaultLogo"
- [Image] – file containing the default image (Fig.7)

Fig. 6. Schema resource properties

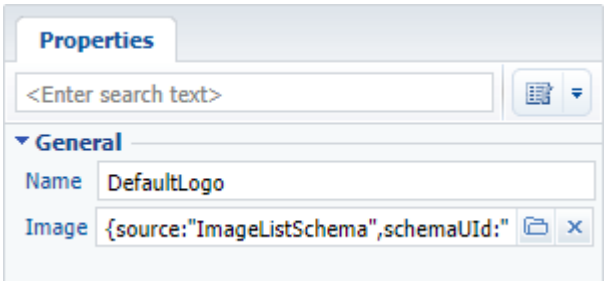


Fig. 7. Default image for the knowledge base article

KB

5. Setting up displaying of a field with logo on the edit page

The field with logo should be placed in the upper part of the account edit page. In the base implementation the fields are placed in such a way that adding a logo can violate the page interface. That is why you need to rearrange the location of existing fields when locating a new field.

Add the configuration object of the filed with logo with the necessary parameters to the *diff* array property of the view model (see the source code below) and describe the modifications of the fields located in the upper part of the page: [Name], [ModifiedBy] and [Type].

The field with image is added to the page by using the additional *PhotoContainer* container-wrapper with the ["image-edit-container"] class.

6. Adding implementation of the following methods to the page view model method collection:

- *getPhotoSrcMethod()* – receives image by link
- *beforePhotoFileSelected()* – is called before opening the image selection dialog box
- *onPhotoChange* – is called upon image modification
- *onPhotoUploaded()* – saves the link to a modified image in the object column

The replacing schema source code is as follows:

```
define("KnowledgeBasePageV2", ["KnowledgeBasePageV2Resources",
"ConfigurationConstants"],
    function(resources, ConfigurationConstants) {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "KnowledgeBase",
            // Edit page view model methods.
            methods: {
                // Called before opening the image selection dialog box.
                beforePhotoFileSelected: function() {
                    return true;
                },
                // Receives image by link.
                getPhotoSrcMethod: function() {
                    // Receiving a link to the image in the object column.
                    var imageColumnValue = this.get("UsrLogo");
                    // If the link is set, the method returns the url of the image
file.
                    if (imageColumnValue) {
                        return this.getSchemaImageUrl(imageColumnValue);
                    }
                    // If the link is not set, it returns the default image.
                    return
this.Terrasoft.ImageUrlBuilder.getUrl(this.get("Resources.Images.DefaultLogo"));
                },
                // Processes the image modification.
                // photo — image file.
                onPhotoChange: function(photo) {
                    if (!photo) {
```

```
                        this.set("UsrLogo", null);
                        return;
                    }
                    // The file is uploaded to the database. onPhotoUploaded is
called when uploading is finished.
                    this.Terrasoft.ImageApi.upload({
                        file: photo,
                        onComplete: this.onPhotoUploaded,
                        onError: this.Terrasoft.emptyFn,
                        scope: this
                    });
                },
                // Saves the link to a modified image.
                // imageId — Id of the saved file from the database.
                onPhotoUploaded: function(imageId) {
                    var imageData = {
                        value: imageId,
                        displayValue: "Image"
                    };
                    // The image column is assigned a link to the image.
                    this.set("UsrLogo", imageData);
                }
            },
            //
            diff: /**SCHEMA_DIFF*/[
                // Container-wrapper that the component will be located in.
                {
                    // Adding operation.
                    "operation": "insert",
                    // Parent container meta-name, where the component is added.
                    "parentName": "Header",
                    // The image is added to the component collection of the
                    // parent container.
                    "propertyName": "items",
                    // Schema component meta-name, involved in the action.
                    "name": "PhotoContainer",
                    // Properties passed to the component structure.
                    "values": {
                        // Element type — container.
                        "itemType": Terrasoft.ViewItemType.CONTAINER,
                        // CSS-class name.
                        "wrapClass": ["image-edit-container"],
                        // Locating in the parent container.
                        "layout": { "column": 0, "row": 0, "rowSpan": 3, "colSpan": 3
},
                        // Child element array.
                        "items": []
                    }
                },
                // The [UsrLogo] field — the field with account logo.
                {
                    "operation": "insert",
                    "parentName": "PhotoContainer",
                    "propertyName": "items",
                    "name": "UsrLogo",
                    "values": {
                        // Method receiving image by link.
                        "getSrcMethod": "getPhotoSrcMethod",
                        // Method called upon image modification.
                        "onPhotoChange": "onPhotoChange",
                        // Method called before opening the image selection dialog
box.
```

```
                        "beforeFileSelected": "beforePhotoFileSelected",
                        // Property defining the capability of image editing
(changing, deleting).
                        "readonly": false,
                        // Control element view-generator.
                        "generator":
"ImageCustomGeneratorV2.generateCustomImageControl"
                    }
                },
                // Rearranging the location of the [Name] field.
                {
                    // Merge operation.
                    "operation": "merge",
                    "name": "Name",
                    "parentName": "Header",
                    "propertyName": "items",
                    "values": {
                        "bindTo": "Name",
                        "layout": {
                            "column": 3,
                            "row": 0,
                            "colSpan": 20
                        }
                    }
                },
                // Rearranging the location of the [ModifiedBy] field.
                {
                    "operation": "merge",
                    "name": "ModifiedBy",
                    "parentName": "Header",
                    "propertyName": "items",
                    "values": {
                        "bindTo": "ModifiedBy",
                        "layout": {
                            "column": 3,
                            "row": 2,
                            "colSpan": 20
                        }
                    }
                },
                // Rearranging the location of the [Type] field.
                {
                    "operation": "merge",
                    "name": "Type",
                    "parentName": "Header",
                    "propertyName": "items",
                    "values": {
                        "bindTo": "Type",
                        "layout": {
                            "column": 3,
                            "row": 1,
                            "colSpan": 20
                        },
                        "contentType": Terrasoft.ContentType.ENUM
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
    });
```

The default logo will be displayed on the knowledge base article edit page after you save the schema and update the application page. When you hover over the image, you will see an action menu appear. You can use it to delete the

image or set up a new one for a specific knowledge base article (Fig.9).

Fig. 8. Default logo

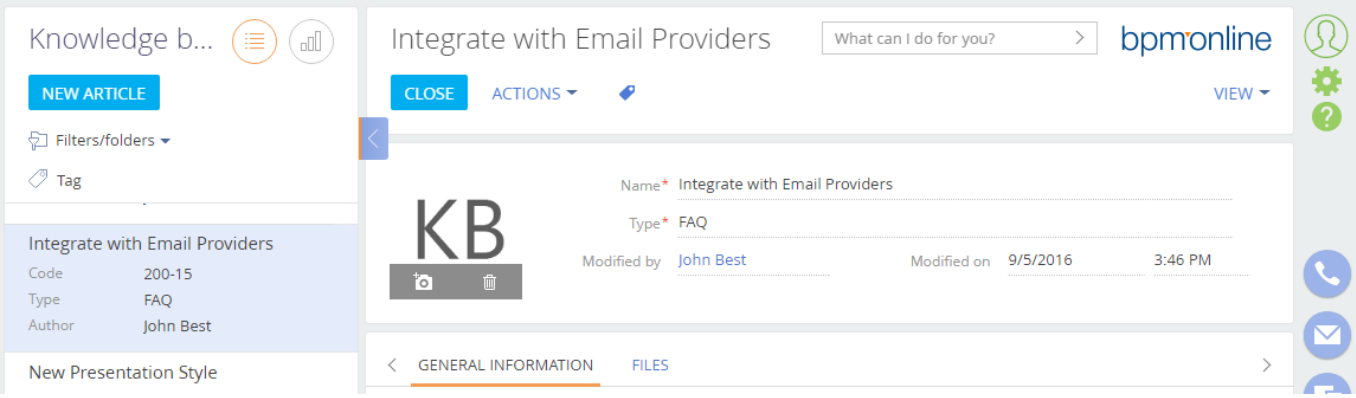
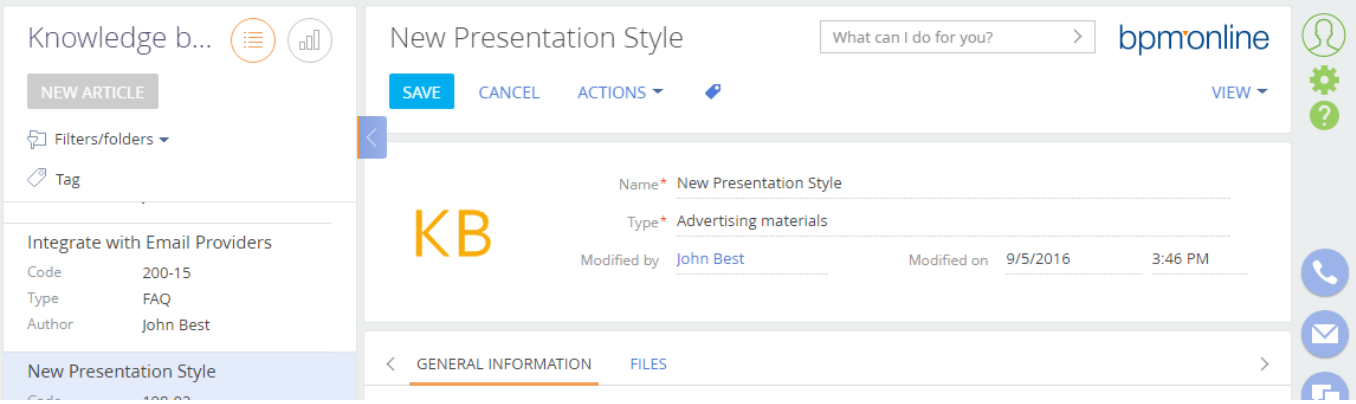
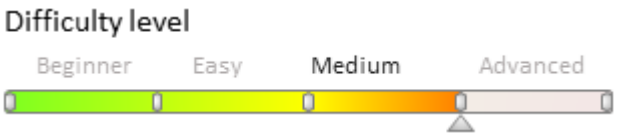


Fig. 9. Custom logo



How to add the color select button to the edit page



Introduction

One of bpm’online control elements is the color button (Fig.1).

Fig. 1. Color button

Algorithm of adding the color button to object edit page:

1. Add the [Text (50 characters)] data type column to the object that will store information about the selected color.
2. Add the [COLOR_BUTTON] type element description to the *diff* array. Set up binding to the column added on previous step for the *value* property of this element.
3. Add a button label via the [LABEL] control element if needed.

Case description

Adding color button to product edit page.

Source code

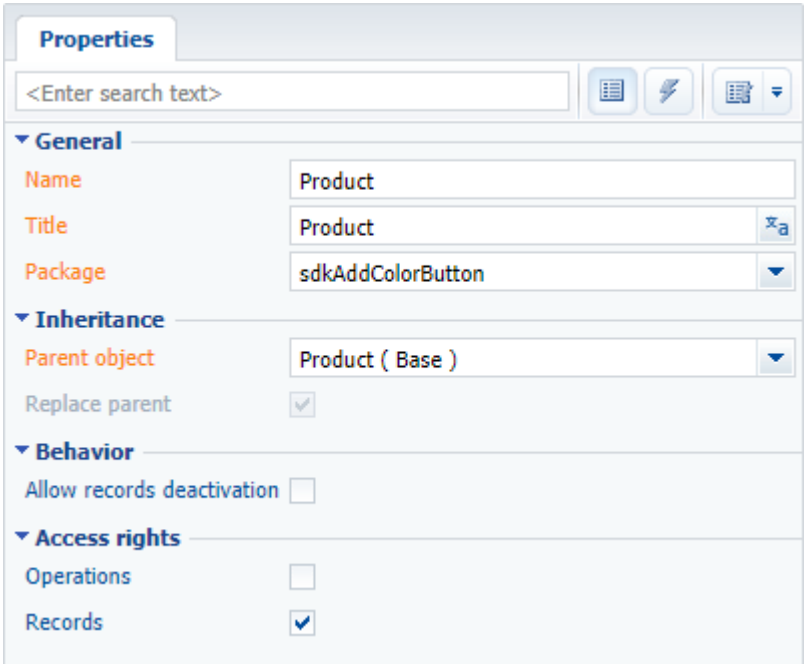
You can download the package with case implementation using the following link..

Case implementation algorithm

1. Creating a "Product" replacing object and adding the [UsrColor] column to it.

Create the [Product] replacing object (Fig.2). Learn more about creating a replacing object in the "**Creating the entity schema**" article.

Fig. 2. Configuration object properties



Add a new column (Fig.3) and indicate the following properties (Fig.4):

- [Title] – "Color"
- [Name] – "UsrColor"
- [Data type] – "Text (50 characters)"

Fig. 3. Adding a new column

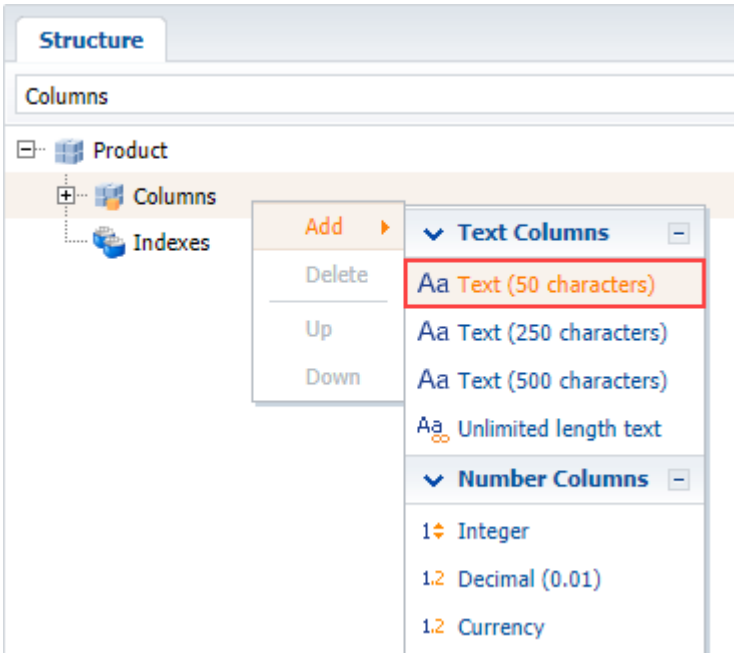
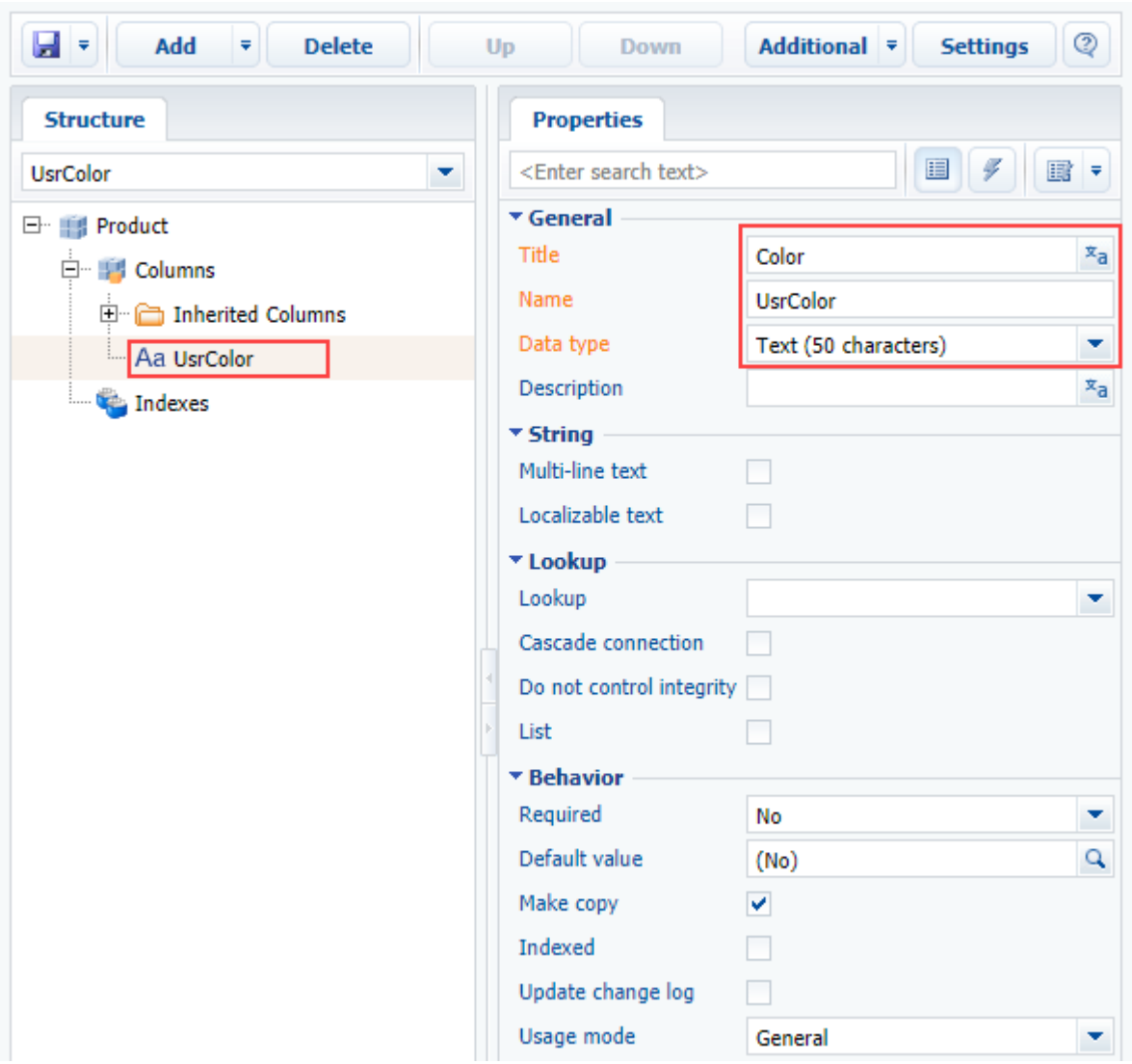


Fig. 4. Properties of the added column

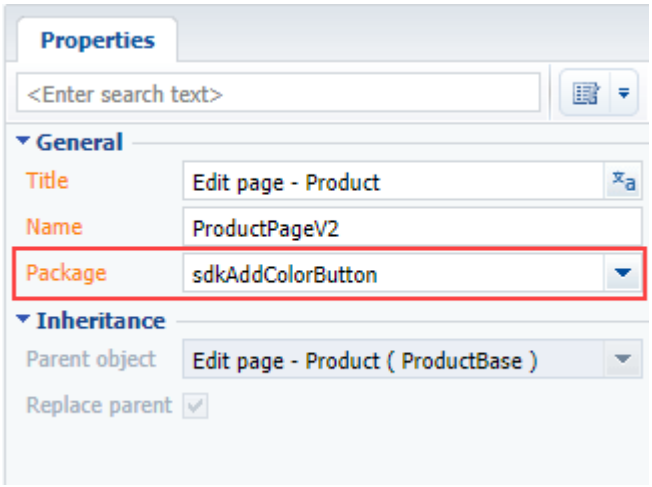


Save and publish the object schema after you set up all properties.

2. Creating a product replacing edit page in custom package

Create a replacing client module and specify the [Edit page – Product], *ProductPageV2* schema as the parent object in it (Fig. 5). The procedure of creating a replacing page is covered in the "**Creating a client schema**" article.

Fig. 5. Properties of the product edit page replacing schema

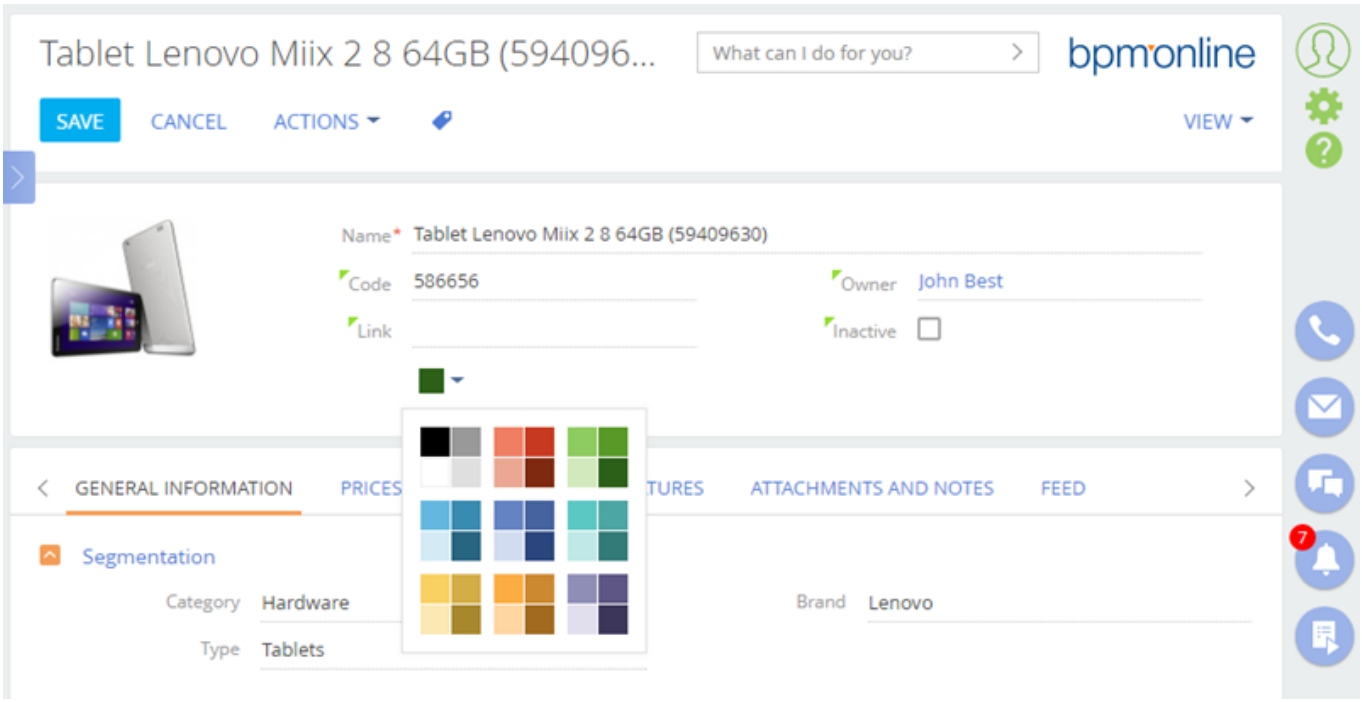


The replacing schema source code is as follows:

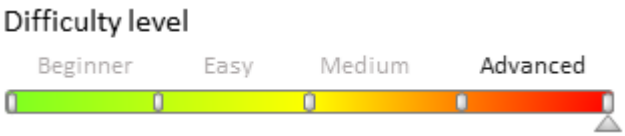
```
define("ProductPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Product",
        diff: /**SCHEMA_DIFF*/[
            // Color button.
            {
                // Operation of adding.
                "operation": "insert",
                //  Meta-name of the parent container where the compponent is added.
                "parentName": "ProductGeneralInfoBlock",
                // The button is added to component collection
                // of the parent container.
                "propertyName": "items",
                // The name of schema component involved in action.
                "name": "ColorButton",
                // Properties transferred to the component constructor.
                "values": {
                    // Element type — color button.
                    "itemType": this.Terrasoft.ViewItemType.COLOR_BUTTON,
                    // Binding of the control element value to the view model column.
                    "value": { "bindTo": "UsrColor" },
                    // Button location.
                    "layout": { "column": 5, "row": 6, "colSpan": 12 }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

After saving the schema and refreshing the application page the color button will be displayed on the product edit page (Fig .6).

Fig. 6. Case result



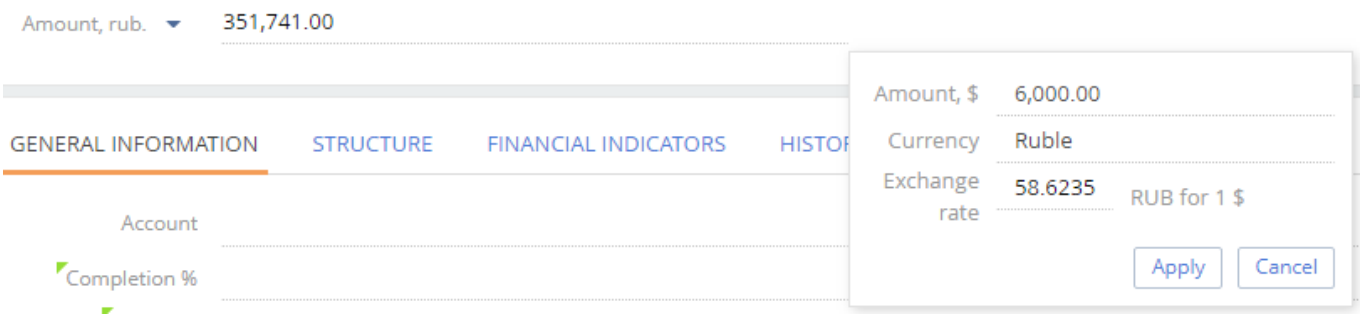
How to add multi-currency field



Introduction

One of the common configuration tasks is adding a [multi-currency field] control element on a page. The multi-currency field enables users to enter monetary sums, specify currencies and exchange rates. If a user changes the currency of a multi-currency field, the amount in it will be automatically re-calculated according to the current exchange rate. Fig. 1 shows an example of a multi-currency field in the system interface.

Fig. 1. Multi-currency field



To add a multi-currency field on an edit page:

1. Add 4 fields to the object schema:

- [Currency] lookup column
- [Exchange rate]
- [Amount]
- [Amount in base currency]

NOTE

> The object itself must contain only the [Amount] field. The rest of the fields can be virtual, unless the business task requires their values to be stored in the database. They can be determined as attributes in the view model schema.

2. Specify 3 modules in the view model class declaration as dependencies:

- *MoneyModule*,
- *MultiCurrencyEdit*,
- *MultiCurrencyEditUtilities*.

3. Connect *Terrasoft.MultiCurrencyEditUtilities* mixin to the view model and initialize it in the overridden *init()* method.

4. Add a configuration object with the multi-currency field settings to the *diff* array of the edit page schema. In addition to common control element properties, the *values* property must contain:

- *primaryAmount* – name of the column that contains the amount in the base currency.
- *currency* – name of the column that references the currency lookup.
- *rate* – name of the column that contains the currency exchange rate.
- *Generator* – control element generator. Specify "*MultiCurrencyEditViewGenerator.generate*".

5. Add recalculation logic. Apply the calculated field mechanism, as described in the **Adding calculated fields** article.

Case description

Add a multi-currency [Amount] field to the project edit page.

Source code

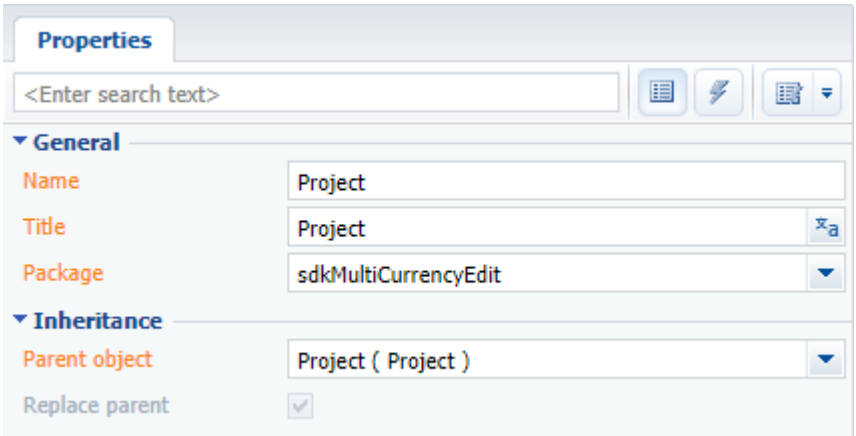
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Add object replacing schema necessary columns

Create the replacing schema of the [Project] object in the custom package (Fig. 2). More information about creating a replacing object and adding columns is described in the **Adding a new field to the edit page** article.

Fig. 2. Properties of the object replacing schema



Add 4 columns with properties given on the Fig. 3 – Fig. 6 to the replacing schema. Column properties in the **object designer** are displayed in the extende mode.

Fig. 3. The [UsrCurrency] column properties

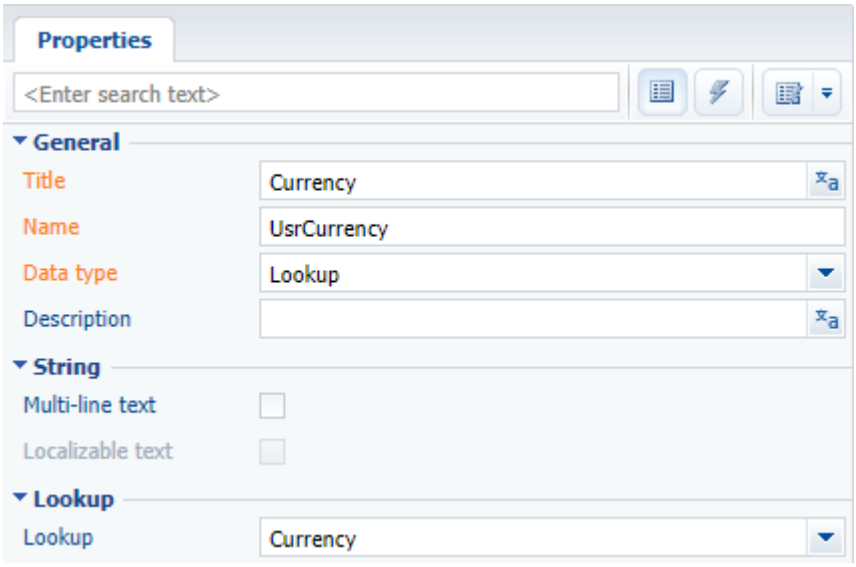


Fig. 4. The [UsrAmount] column properties

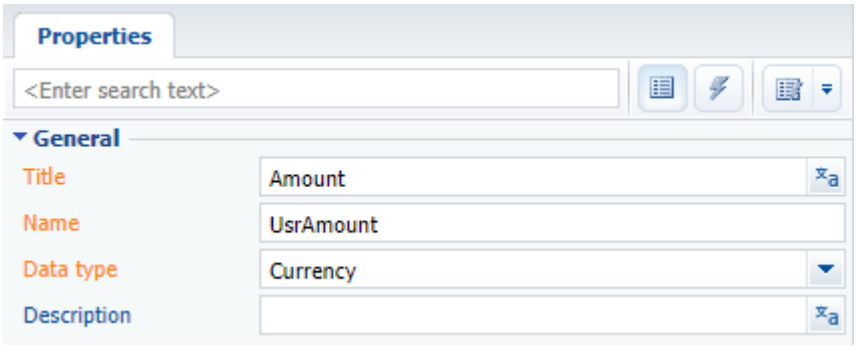


Fig. 5. The [UsrPrimaryAmount] column properties

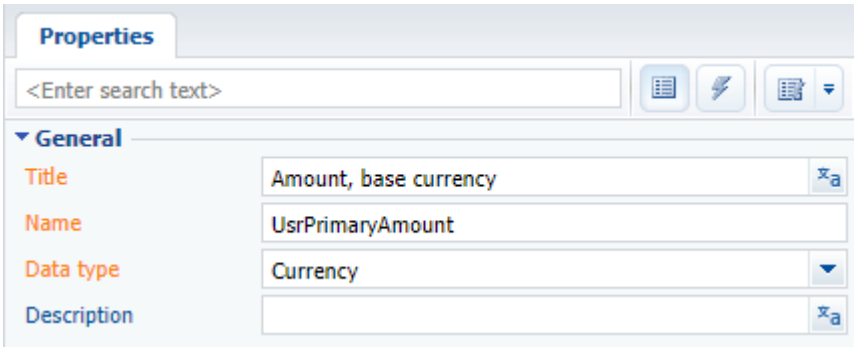
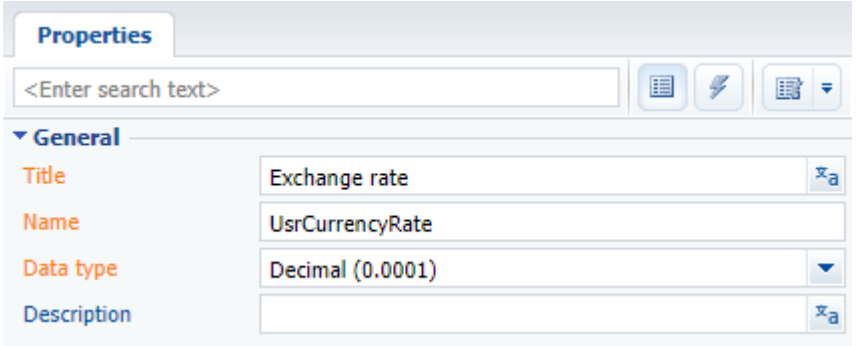


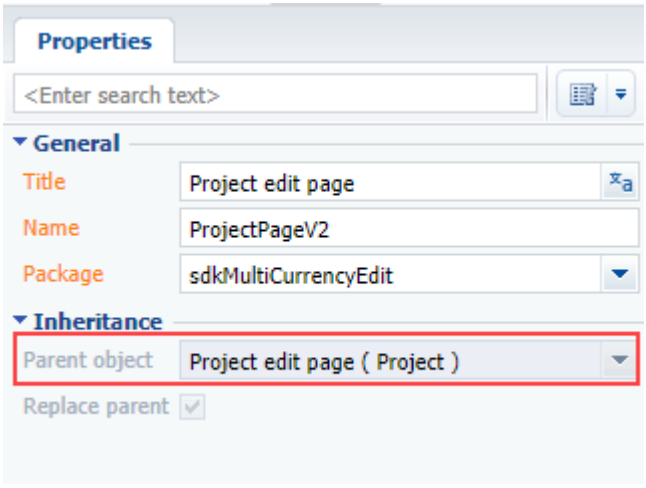
Fig. 6. The [UsrCurrencyRate] column properties



2. Create a replacing edit page for a project in a custom package

A replacing client module must be created and [Project edit page] (*OrderPageV2*) must be specified as the parent object in it (Fig. 7). The procedure of creating a replacing page is covered in the**"Creating a client schema"**article.

Fig. 7. Properties of the [Projects] replacing edit page



Specify the following modules as dependencies when declaring view model class: *MoneyModule*, *MultiCurrencyEdit*, *MultiCurrencyEditUtilities* (see the source code below).

3. Add necessary attributes

Specify the *UsrCurrency*, *UsrCurrencyRate*, *UsrAmount* and *UsrPrimaryAmount* attributes that correspond to the added columns of the object schema in the *attributes* property of the edit page of view model schema.

The multi-currency module operates only with the *Currency* column. Create the *Currency* attribute and declare there a virtual column. Bind this column with the previously created *UsrCurrency* column via the handler method (see the following code below).

5. Connect the Terrasoft.MultiCurrencyEditUtilities mixin to the view model

Declare the *Terrasoft.MultiCurrencyEditUtilities* mixin in the *mixins* properties of the page view model schema. Initialize it in the overridden *init()* method view model schema (see the following code below).

6. Implement the recalculation logic according to the currency.

Add handler methods of the attribute dependencies in the *methods* collection of the (see the following code below).

7. Add multi-currency field on the page

Add the configuration object with the settings of the multi-currency field to the *diff* array of the view model schema of the edit page.

The replacing schema source code is as follows:

```
// Specify modules as dependencies in the view model class declaration
// MoneyModule, MultiCurrencyEdit, MultiCurrencyEditUtilities
define("ProjectPageV2", ["MoneyModule", "MultiCurrencyEdit",
"MultiCurrencyEditUtilities"],
    function(MoneyModule, MultiCurrencyEdit, MultiCurrencyEditUtilities) {
        return {
             // Edit page object schema name.
            entitySchemaName: "Project",
            // The attributes property of the view model.
            attributes: {
                // Currency.
                "UsrCurrency": {
                    // Attribute data type is a lookup.
                    "dataValueType": this.Terrasoft.DataValueType.LOOKUP,
```

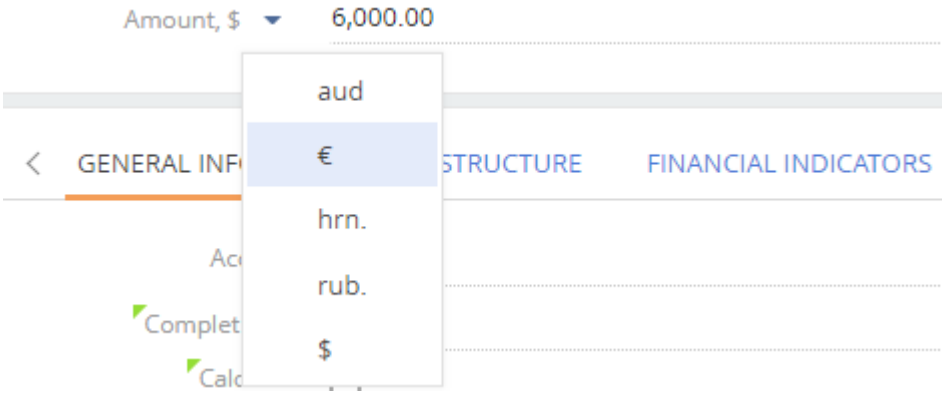
```
                    // Configuration of the lookup.
                    "lookupListConfig": {
                        "columns": ["Division", "Symbol"]
                    }
                },
                // Exchange rate.
                "UsrCurrencyRate": {
                    "dataValueType": this.Terrasoft.DataValueType.FLOAT,
                    // Attribute dependencies.
                    "dependencies": [
                        {
                            // Columns on which the attribute depends.
                            "columns": ["UsrCurrency"],
                            // Handler method.
                            "methodName": "setCurrencyRate"
                        }
                    ]
                },
                // Amount.
                "UsrAmount": {
                    "dataValueType": this.Terrasoft.DataValueType.FLOAT,
                    "dependencies": [
                        {
                            "columns": ["UsrCurrencyRate", "UsrCurrency"],
                            "methodName": "recalculateAmount"
                        }
                    ]
                },
                // Amount in base currency.
                "UsrPrimaryAmount": {
                    "dependencies": [
                        {
                            "columns": ["UsrAmount"],
                            "methodName": "recalculatePrimaryAmount"
                        }
                    ]
                },
                // Currency is a virtual column for compatibility with the
MultiCurrencyEditUtilities module.
                "Currency": {
                    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                    "dataValueType": this.Terrasoft.DataValueType.LOOKUP,
                    "lookupListConfig": {
                        "columns": ["Division"]
                    },
                    "dependencies": [
                        {
                            "columns": ["Currency"],
                            "methodName": "onVirtualCurrencyChange"
                        }
                    ]
                }
            },
            // View model mixins.
            mixins: {
                // Mixin that controls multicurrency on the edit page.
                MultiCurrencyEditUtilities: "Terrasoft.MultiCurrencyEditUtilities"
            },
            // Methods of the page view model.
            methods: {
                // Overriding the Terrasoft.BasePageV2.init() basic method.
                init: function() {
```

```
                    // Calling the parent implementation of init method.
                    this.callParent(arguments);
                    // Initialization of the mixin controlling the multi-currency.
                    this.mixins.MultiCurrencyEditUtilities.init.call(this);
                },
                // Sets the exchange rate.
                setCurrencyRate: function() {
                    //Loads the exchange rate at the beginning of the project.
                    MoneyModule.LoadCurrencyRate.call(this, "UsrCurrency",
"UsrCurrencyRate", this.get("StartDate"));
                },
                // Recalculates amount.
                recalculateAmount: function() {
                    var currency = this.get("UsrCurrency");
                    var division = currency ? currency.Division : null;
                    MoneyModule.RecalcCurrencyValue.call(this, "UsrCurrencyRate",
"UsrAmount", "UsrPrimaryAmount", division);
                },
                // Recalculates amount in base currency.
                recalculatePrimaryAmount: function() {
                    var currency = this.get("UsrCurrency");
                    var division = currency ? currency.Division : null;
                    MoneyModule.RecalcBaseValue.call(this, "UsrCurrencyRate",
"UsrAmount", "UsrPrimaryAmount", division);
                },
                // The handler of the currency virtual column change.
                onVirtualCurrencyChange: function() {
                    var currency = this.get("Currency");
                    this.set("UsrCurrency", currency);
                }
            },
            // Setting up the visualization of a multi-currency field on the edit
page.
            diff: /**SCHEMA_DIFF*/[
                // Metadata for adding the[Amount] field.
                {
                    // Adding operation.
                    "operation": "insert",
                    // The meta-name of the parent container to which the component
is added.
                    "parentName": "Header",
                    // The field is added to the parent container's collection.
                    "propertyName": "items",
                    // The meta-name of the schema component above which the action
is performed.
                    "name": "UsrAmount",
                    // Properties passed to the component's constructor.
                    "values": {
                        // The name of the column of the view model to which the
binding is performed.
                        "bindTo": "UsrAmount",
                        // Element location in the container.
                        "layout": { "column": 0, "row": 2, "colSpan": 12 },
                        // The name of the column that contains the amount in the
base currency.
                        "primaryAmount": "UsrPrimaryAmount",
                        // The name of the column that contains the currency of the
amount.
                        "currency": "UsrCurrency",
                        // The name of the column that contains the exchange rate.
                        "rate": "UsrCurrencyRate",
                        // The property that defines the availability for editing the
```

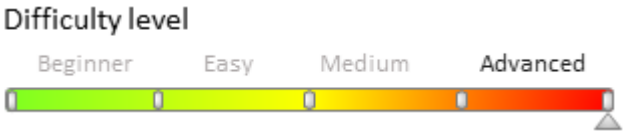
```
amount field in the base currency.
                        "primaryAmountEnabled": false,
                        // Generator of the control view.
                        "generator": "MultiCurrencyEditViewGenerator.generate"
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
    });
```

After saving the schema and refreshing the application page the [Amount] multi-currency field will be displayed on the project edit page (Fig .1). The value of the field will be automatically recalculated after selecting a currency from the drop-down list (Fig. 8).

Fig. 8. Drop-down list of currencies



How to add custom logic to the existing controls



Introduction

Controls are objects used to create an interface between the user and a bpm’online application. For example, buttons, fields, checkboxes, etc,

All controls in bpm’online are inherited from the *Terrasoft.controls.Component* class. Full list of classes that implement bpm’online components is available by link in the “**JavaScript API for platform core (on-line documentation)**”.

According to the Open–closed principle, you cannot add custom logic to the existing control. For this you need to create a new class that inherits functions of the existing class of the control. And implement new functions in the successor class.

Steps to add new functions:

1. Create new client module.

2. In the client module, declare a class that inherits the existing control. Implement necessary functions in the class.

3. Add a new item to the bpm’online interface.

Case description

Create control which enables to enter only integer values in the specified range. Perform the checking of the entered value by pressing the Enter key and display the corresponding message if the number is outside the range. Use the *Terrasoft.controls.IntegerEdit* control as parent.

Source code

You can download the package with case implementation using the following link.

Case implementation algorithm

1. Create a client module

The procedure for creating a custom schema is covered in the "**Creating a custom client module schema**".

Run the [Add] – [Module] menu command on the [Schemas] tab of the [Configuration] section.

Specify following properties of the schema:

- [Name] – "UsrLimitedIntegerEdit"
- [Title] – "UsrLimitedIntegerEdit"

Add the following source code to the schema:

```
// Declaration of the module.
define("UsrLimitedIntegerEdit", [], function () {
});
```

2. Create a class of the control

Modify the source code according to the example below.

```
define("UsrLimitedIntegerEdit", [], function () {
    //  Declaration of the class of the control.
    Ext.define("Terrasoft.controls.UsrLimitedIntegerEdit", {
        // Base class.
        extend: "Terrasoft.controls.IntegerEdit",
        // Alias (abbreviated name of the class)..
        alternateClassName: "Terrasoft.UsrLimitedIntegerEdit",
        // The smallest allowed value.
        minLimit: -1000,
        // The highest value allowed.
        maxLimit: 1000,
        // A method for checking for an occurrence in the range of valid values.
        isOutOfLimits: function (numericValue) {
            if (numericValue < this.minLimit || numericValue > this.maxLimit) {
                return true;
            }
            return false;
        },
        // Override the method of the event handler for pressing the Enter key.
        onEnterKeyPressed: function () {
            // Call the basic functionality.
            this.callParent(arguments);
            // Get the entered value.
            var value = this.getTypedValue();
            // Reduction to a numberic type.
            var numericValue = this.parseNumber(value);
            // Check for occurrence in the range of acceptable values.
            var outOfLimits = this.isOutOfLimits(numericValue);
            if (outOfLimits) {
                // Form the warning message.
                var msg = "Value " + numericValue + " is out of limits [" +
this.minLimit + ", " + this.maxLimit + "]";
                // Modify the configuration object to display a warning message.
                this.validationInfo.isValid = false;
```

```
                this.validationInfo.invalidMessage = msg;
            }
            else{
                // Modify the configuration object to hide the warning message.
                this.validationInfo.isValid = true;
                this.validationInfo.invalidMessage ="";
            }
            // Call the logic for displaying the warning message.
            this.setMarkOut();
        },
    });
});
```

> **NOTE**
>
> You can use the logic of the *onEnterKeyPressed()* method in the in the *onBlur()* event handler.

Save the schema.

Ecxept the *extend* and *alternateClassName* standard properties, the *minLimit* and *maxLimit* properties that specify the range of allowed values are added to the class. Default values are used for these properties.

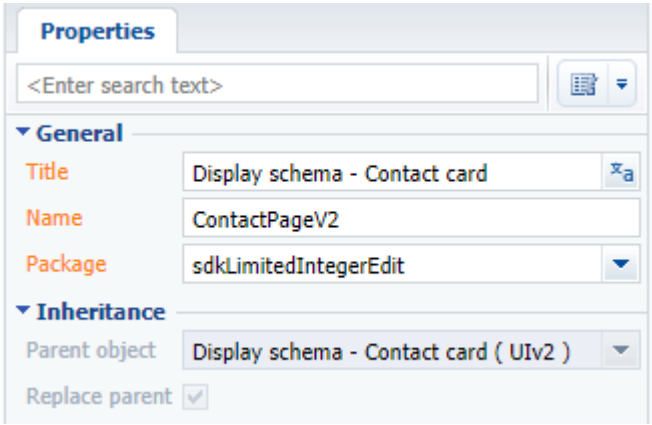
The required control logic is implemented in the *onEnterKeyPressed* override method After calling the base logic in which the generation of the value change events is performed, the entered value is checked for validity. If the number is not valid, the corresponding warning message is displayed in the input field. The *isOutOfLimits*method is provided to check the occurrence of the entered value in the range of allowed values.

> **ATTENTION**
>
> With this implementation, despite the output of the corresponding warning, the entered value is still stored and transferred to the schema view model in which the component will be used.

3. Add the control to the bpm'online interface

To add the created control to the bpm'online, create the replacing schema, for example, the contact record page. Create a replacing client module and specify the [Display schema – Contact card] (*ContactPageV2*) schema as parent object (Fig. 1). Creating a replacing page is covered in the "**Creating a custom client module schema**" article.

Fig. 1. Properties of the replacing edit page



Add the following source code to the schema:

```
// Declaration of the module. Be sure to specify the dependency
// of the module in which the class of the control is declared.
define("ContactPageV2", ["UsrLimitedIntegerEdit"],
    function () {
        return {
```

```
            attributes: {
                // Attribute associated with the value in the control.
                "ScoresAttribute": {
                    // Attribute data type is integer.
                    "dataValueType": this.Terrasoft.DataValueType.INTEGER,
                    // Attribute type is a virtual column.
                    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                    // The default value.
                    "value": 0
                }
            },
            diff: /**SCHEMA_DIFF*/[
                {
                    // The type of operation is the addition.
                    "operation": "insert",
                    // The name of the container to which the control is added.
                    "parentName": "ProfileContainer",
                    // The name of the property in the container to which you want to
add
                    // instance of the control.
                    "propertyName": "items",
                    // The name of the control.
                    "name": "Scores",
                    // Header.
                    "caption": "Scores",
                    // Values passed to the properties of the control.
                    "values": {
                        // The type of the control is the component.
                        "itemType": Terrasoft.ViewItemType.COMPONENT,
                        // The name of the class.
                        "className": "Terrasoft.UsrLimitedIntegerEdit",
                        // The value property of the component is associated with the
ScoresAttribute attribute.
                        "value": { "bindTo": "ScoresAttribute" },
                        // Values for the minLimit property.
                        "minLimit": -300,
                        // Values for the maxLimit property.
                        "maxLimit": 300,
                        // The location of the component in the container.
                        "layout": {
                            "column": 0,
                            "row": 6,
                            "colSpan": 24,
                            "rowSpan": 1
                        }
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
    });
```

Save the schema.

The added *ScoresAttribute* attribute contains the value connected to the value entered in input field of the control. You can use an integer column of the object connected to the view model of the record edit page instead of the attribute.

The configuration object determining the values of the properties of control entity is added to the *diff* array. The value of the "value" property is connected to the *ScoresAttribute* attribute. The values that specify a valid input range are assigned to the *minLimit* and *maxLimit* properties.

 ATTENTION

If the *minLimit* and *maxLimit* properties are not explicitly specified in the configuration object, the default range (-1000, 1000) will be applied.

As a result, the integer field will be added to the contact record page (Fig. 2). The warning message will be displayed in the field if the invalid message will be entered (Fig. 3).

Fig. 2. Case result

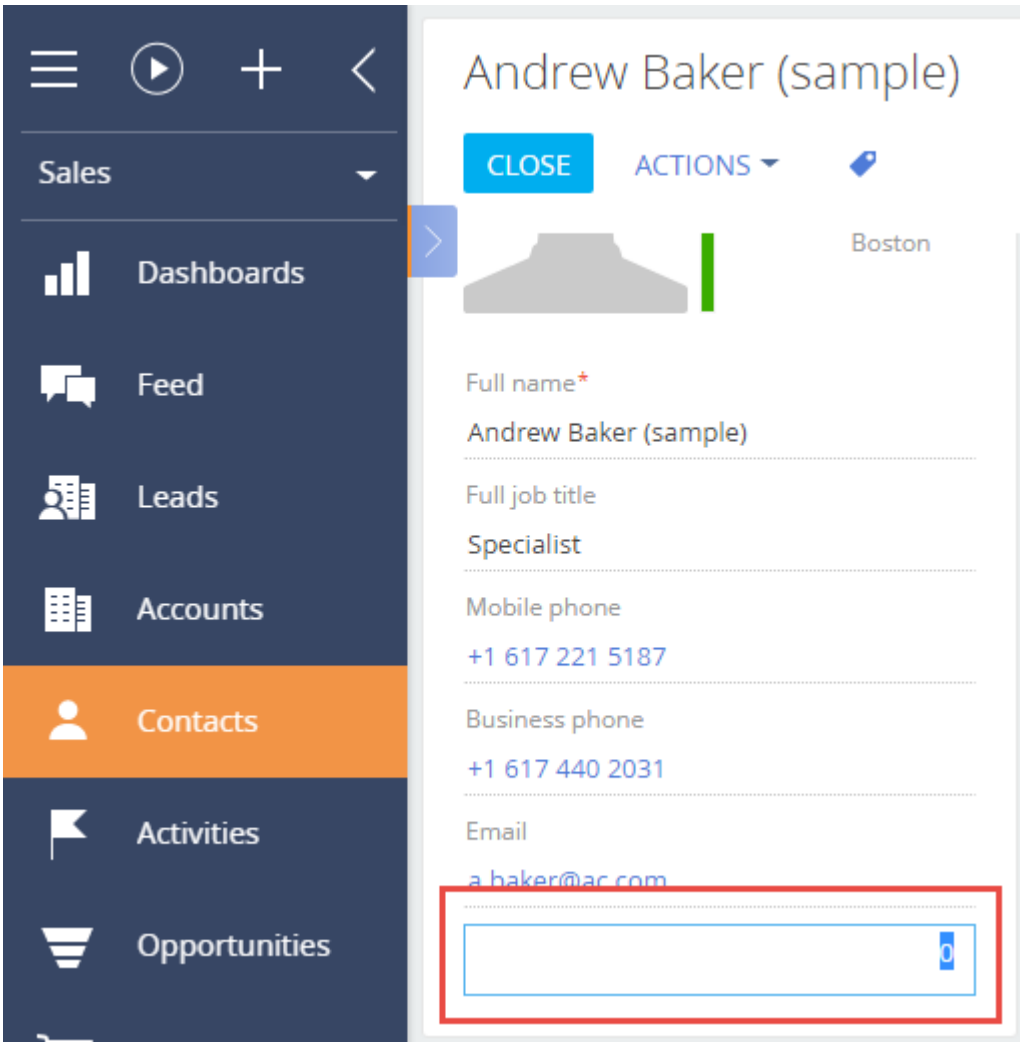
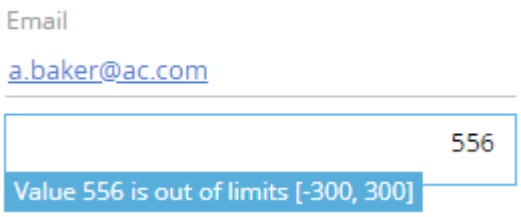
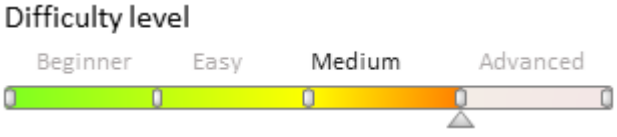


Fig. 3. Displaying of warning message



Adding calculated fields



Introduction

A calculated field is a page control whose value is generated based on the status and values in other elements on this page.

In bpm’online, calculated fields are based on the bpm’online client mechanism, which uses subscriptions to changes in view model schema attributes. For any attribute, you can set a configuration object and specify object schema column names. If the values in these columns change, the value of the calculated column will be updated. You can also specify the handler method for this event.

The general sequence of adding a calculated field is as follows:

1. Add a column for storing the values of the calculated field to the page object schema.
2. In the page view model, set up attribute dependencies by specifying column names from which it depends and the handler name.
3. Add the implementation of the handler method to the method collection of the view model.
4. Set up the display of the calculated field in the *diff* property of the view model.

Setting up dependencies of the calculated field

In the *attributes* view model property, add an attribute for which the dependency is set up.

Declare a *dependencies* property, which is an array of configuration objects, each of which contains the following properties:

- *Columns* – an array of columns whose values determine the value of the current column.
- *methodName* – handler method name.

If the value of at least one of these columns changes in the view model, the event handler method (whose name is specified in the *methodName* property) will be called.

The handler method implementation must be added to the collection of the view methods.

Case description

Add [Payment balance] to display the balance between order amount and payment amount on the order edit page.

NOTE

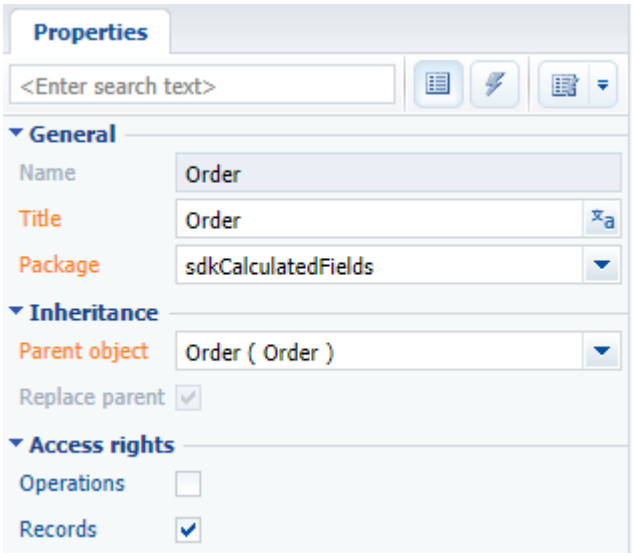
You can add fields to the edit page manually or using the section wizard. For more on adding fields to edit pages see the “**Adding a new field to the edit page**” article.

Case implementation algorithm

1. Create a replacing object

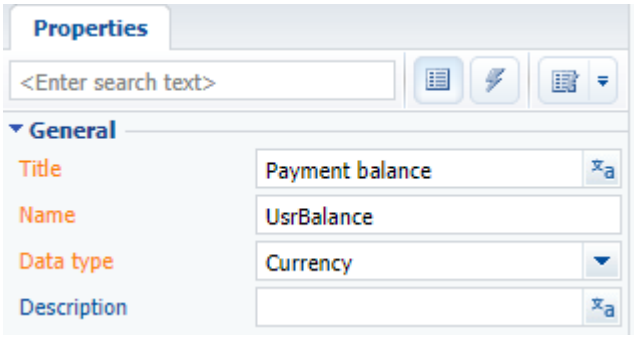
Select the custom package on the [Schemas] tab an select the [Replacing object] in the [Add] menu. Select the [Order] object as the parent object (Fig. 1).

Fig. 1. Properties of the [Order] replacing object



Add a new [Payment balance] column of the [Currency] type to the replacing object (Fig. 2).

Fig. 2. Adding a custom column to the replacing object

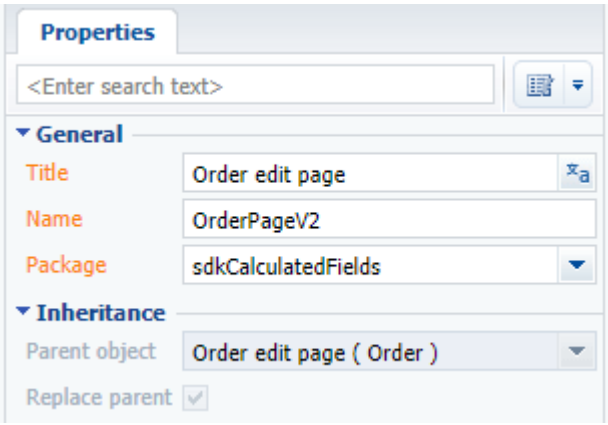


Publish the object.

2. Create a replacing client module for the order edit page

Create a replacing client module and specify the [Order edit page] (*OrderPageV2*) module as parent (Fig. 3). The procedure for creating a replacing page is covered in the "**Creating a custom client module schema**" article.

Fig. 3. Order edit page replacing schema properties



3. Set up the display of the [Payment balance] field

To do this, describe the configuration object with the required parameters in the *diff* property of the view model. Page schema source code is available below

4. Add UsrBalance attribute to the model schema

To do this, add the UsrBalance attribute to the collection of the *attributes* property in the source code of the page view model. Specify dependency from the [Amount] and [PaymentAmount] columns, as well as the *calculateBalance()* handler method (which will be calculating the value of the [UsrBalance] column) in the configuration object of the UsrBalance attribute.

5. Add the needed methods in the methods collection of the view model

In the *methods* collection of the view model, add the *calculateBalance()* handler method that will handle the editing of the [Amount] and [PaymentAmount] columns. This method is used in the *UsrBalance* attribute.

Override the base virtual method *onEntityInitialized()*. The *onEntityInitialized()* method is triggered after the edit page object schema is initialized. Calling the calculateBalance handler method to this method will ensure the calculation of the amount to be paid at the moment the order page opens and not only when the dependency columns are edited.

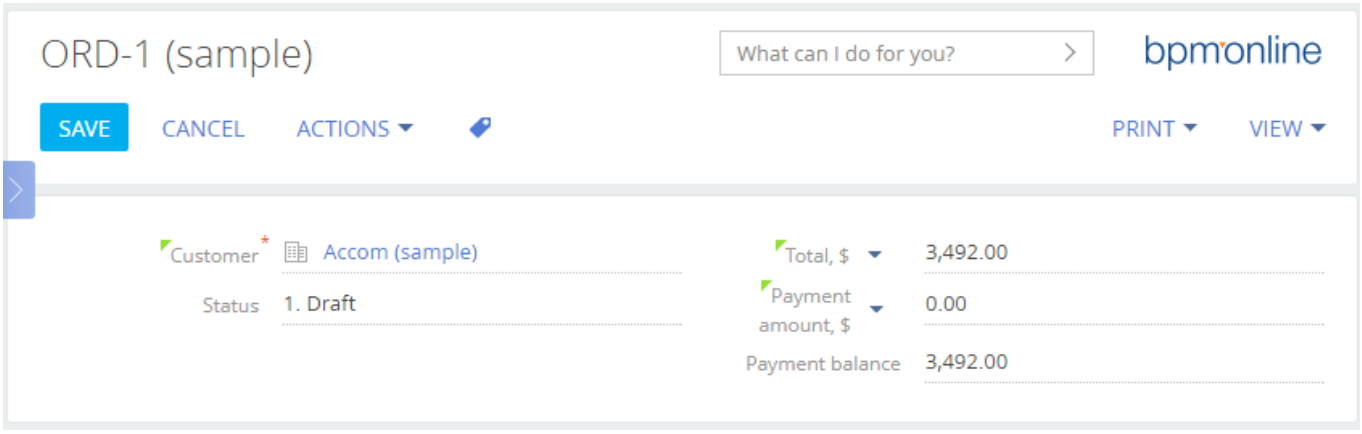
The complete source code of the module is available below:

```
define("OrderPageV2", [], function() {
    return {
        // Edit page object schema name.
        entitySchemaName: "Order",
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        // The attributes property of the view model.
        attributes: {
            // Name of the view model attribute.
            "UsrBalance": {
                // Data type of the view model column.
                dataValueType: Terrasoft.DataValueType.FLOAT,
                // Array of configuration objects that determines [UsrBalance] column
dependencies.
                dependencies: [
                    {
                        // The value in the [UsrBalance] column depends on the
[Amount]
                        // and [PaymentAmount] columns.
                        columns: ["Amount", "PaymentAmount"],
                        // Handler method, which is called on modifying the value of
the on of the columns: [Amount]
                        // and [PaymentAmount].
                        methodName: "calculateBalance"
                    }
                ]
            }
        },
        // Collection of the edit page view model methods.
        methods: {
            // Overriding the base Terrasoft.BasePageV2.onEntityInitialized method,
which
            // is triggerd after the edit page object schema has been initialized.
            onEntityInitialized: function() {
                // Method parent implementation is called.
                this.callParent(arguments);
                // Calling the handler method, which calculates the value in the
[UsrBalance] column.
                this.calculateBalance();
            },
            // Handler method that calculates the value in the [UsrBalance] column.
            calculateBalance: function() {
                // Checking whether the [Amount] and [PaymentAmount] columns are
initialized
                // when the edit page is opened. If not, then zero values are set for
```

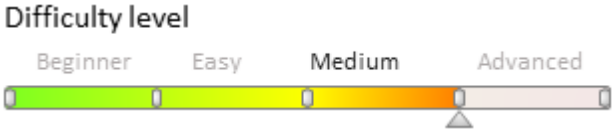
```
them.
                var amount = this.get("Amount");
                if (!amount) {
                    amount = 0;
                }
                var paymentAmount = this.get("PaymentAmount");
                if (!paymentAmount) {
                    paymentAmount = 0;
                }
                // Calculating the margin between the values in the [Amount] and
[PaymentAmount] columns.
                var result = amount - paymentAmount;
                // The calculation result is set as the value in the [UsrBalance]
column.
                this.set("UsrBalance", result);
            }
        },
        // Visual display of the [UsrBalance] column on the edit page.
        diff: /**SCHEMA_DIFF*/[
            {
                "operation": "insert",
                "parentName": "Header",
                "propertyName": "items",
                "name": "UsrBalance",
                "values": {
                    "bindTo": "UsrBalance",
                    "layout": {"column": 12, "row": 2, "colSpan": 12}
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

After saving the schema, updating the web page and clearing the cache, a new [Payment balance] will appear on the order page. The value in this field will be calculated based on the values in the [Total] and [Payment amount] fields (Fig. 4).

Fig. 4. Case result demonstration



How to set a default value for a field



Introduction

In bpm’online, you can define the default values for edit page control elements.

You can set a default value in two ways:

1. Set the value on the business object column level. When creating a new object, its certain page fields should be populated with some initially known values. In such cases, indicate these values for the corresponding object columns as the default values in object designer.

Types of default values.

Name	Description
Set constant	String, number, lookup value, Boolean.
Set from system setting	The complete list of system settings is available in the [System settings] section. It can be supplemented with custom system settings.
Set from system variable	Bpm’online system variables are global variables that store information about system-wide setting values. Unlike system settings, whose values can differ depending on different users, system variable values always remain the same for all users. The full list of system variables is implemented on the kernel level and cannot be changed by user: • New identifier • New sequential identifier • Current user • Contact of the current user • Account of the current user • Current date and time value • Current date value • Current time value
Default value not set	

2. Specify in the edit page source code. In some cases, it is impossible to set a default value via the object column properties. For example, these can be estimated values which are calculated by other column values of the object, etc. In such a case, you can set a default value only via programming means.

Source code

You can download the package with case implementation using the following link.

Example of setting a field default value via object column properties

Case description

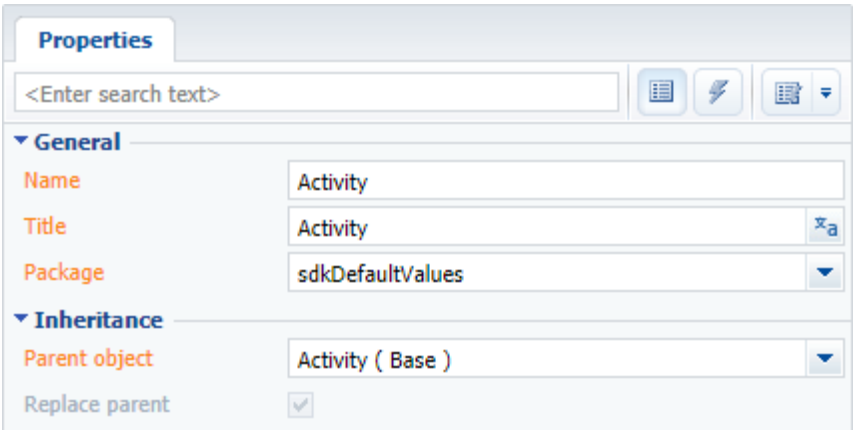
When creating a new activity, the [Show in calendar] checkbox should be set by default.

Case implementation algorithm

1. Creating the [Activity] replacing object in the custom package

Create the [Activity] replacing object (Fig.1). Learn more about creating a replacing object in the “**Creating the entity schema**” article.

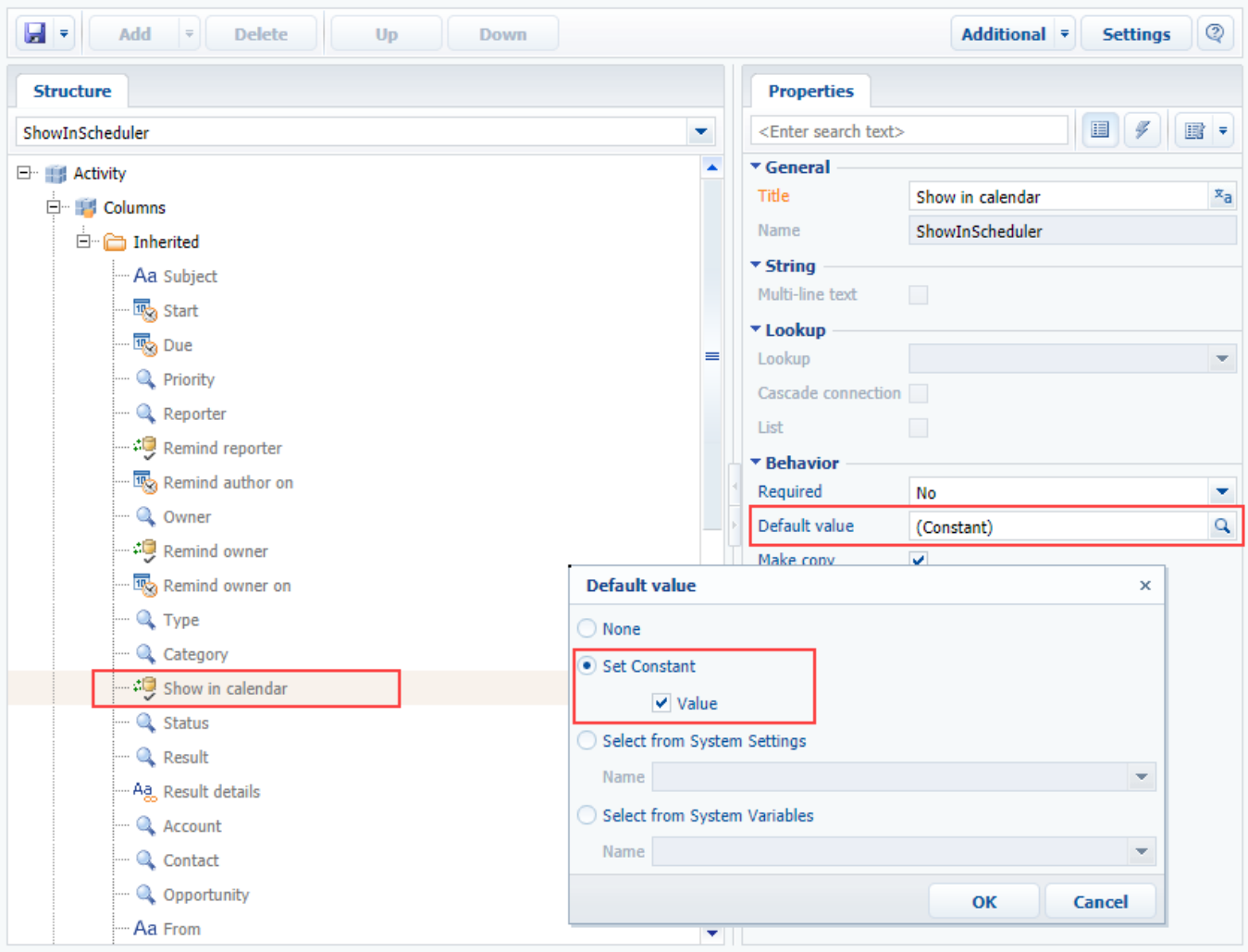
Fig. 1. The [Activity] replacing object properties



2. Setting the default value for the [Show in calendar] column

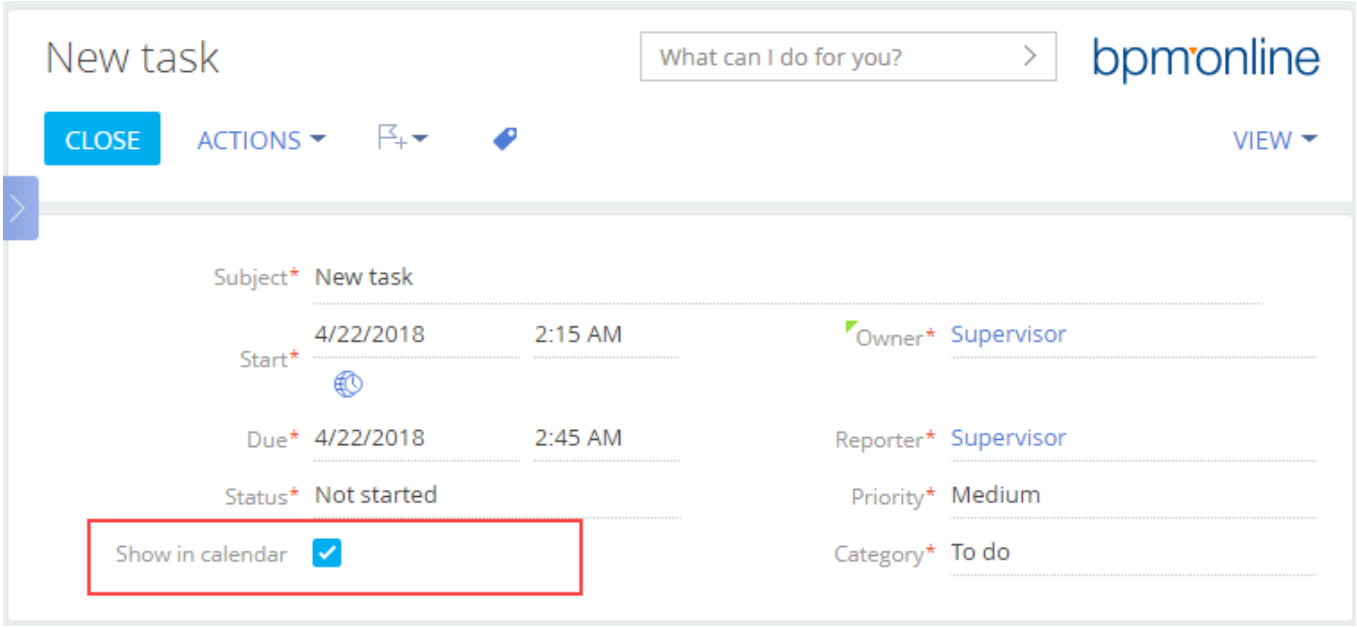
Select the [Show in calendar] column from the inherited column list and edit its [Default value] property as shown in fig.2. To implement the case, select a constant value as the default one.

Fig. 2. Setting the default value for the [Show in calendar] column



After you publish the schema, update the page and clear the cache. The [Show in calendar] field will be selected on the activity edit page when adding a new activity.

Fig. 3. Demonstration of setting the default value



Example of setting a default value in the edit page source code

Case description

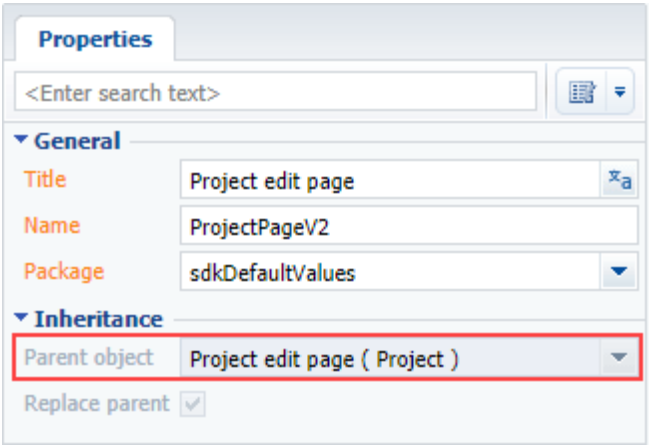
The default value in the [Deadline] field on the project edit page should be as follows: the [Start] field value plus 10 days.

Case implementation algorithm

1. Create a project replacing edit page in custom package

Create a replacing client module and specify the [Project edit page], *ProjectPageV2* schema as its parent object (Fig. 4). The procedure of creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 4. Replacing edit page properties



2. Add the implementation of the following methods to the method collection of the page view model

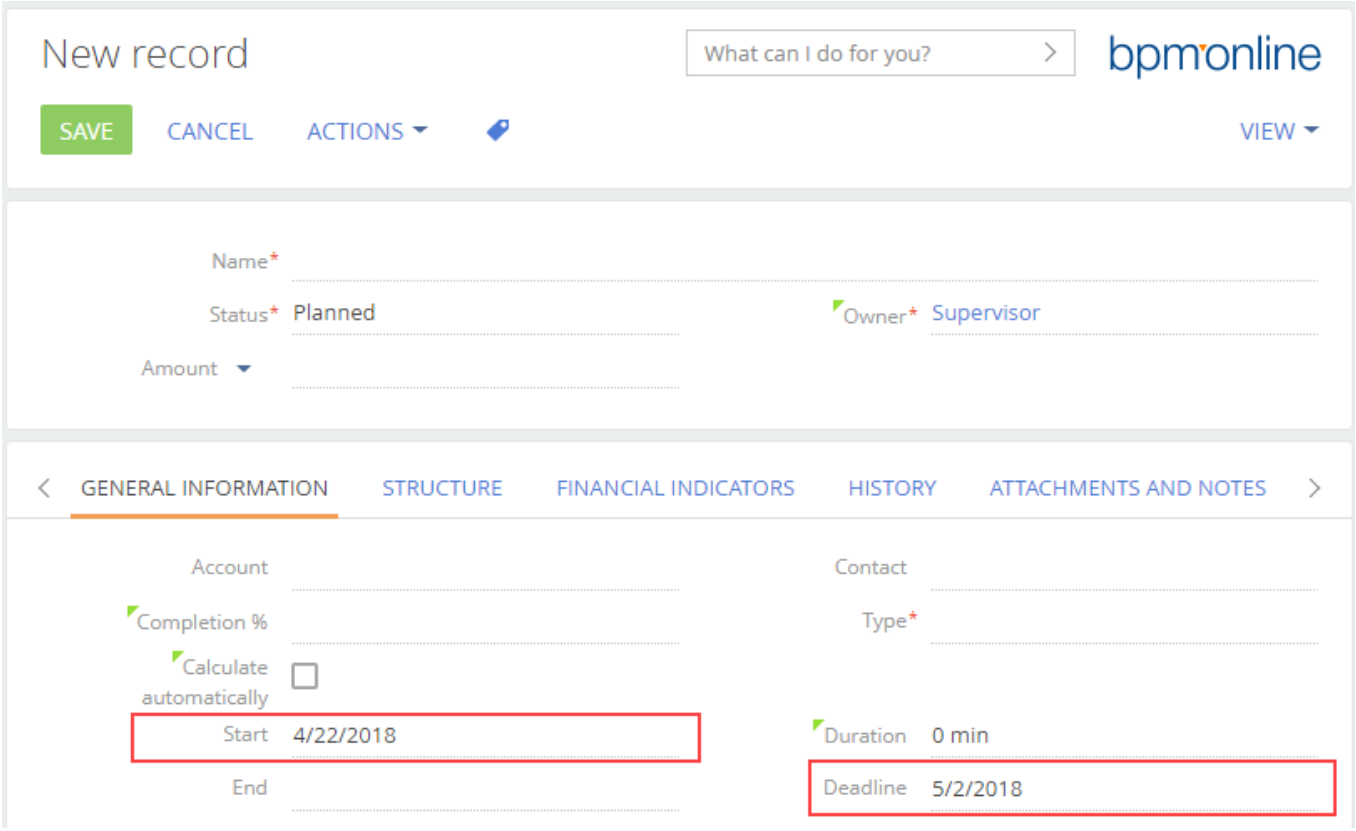
- *setDeadline()* – handler method. Calculates the [Deadline] field value.
- *onEntityInitialized()* – an overridden base virtual method. Triggered upon termination of object schema initialization. Add the handler method call to set the [Deadline] field value to it when opening the edit page.

The replacing schema source code is as follows:

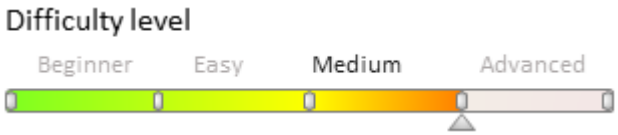
```
define("ProjectPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Project",
        methods: {
            // Overriding the Terrasoft.BasePageV2.onEntityInitialized() base method.
            // Triggered upon termination of edit page object schema initialization.
            onEntityInitialized: function() {
                // Calling of method parent implementation.
                this.callParent(arguments);
                // Calling of handler method that calculates the [Deadline] field
value.
                this.setDeadline();
            },
            // Handler method. Calculates the [Deadline] field value.
            setDeadline: function() {
                // The [Deadline] column value.
                var deadline = this.get("Deadline");
                // Is a new record mode set?
                var newmode = this.isNewMode();
                // If the value is not set and the new record mode is set.
                if (!deadline && newmode) {
                    // Receipt of the [Start] column value.
                    var newDate = new Date(this.get("StartDate"));
                    newDate.setDate(newDate.getDate() + 10);
                    // Setting of the [Deadline] column value.
                    this.set("Deadline", newDate);
                }
            }
        }
    };
});
```

Save the schema and update the application page. A date that equals the [Start] field date plus 10 days will be set in the [Deadline] field (Fig.5).

Fig. 5. Demonstration of setting the calculated default value



How to add the field validation



Introduction

Validation is the verification of field values for their compliance with certain requirements. Values of the bpm'online page fields are validated at the level of the page view model columns. The logic of the field value validation is implemented in the custom validation method.

Validator is the method of the view model where values of the view model column are analyzed for compliance with business requirements. This method must return validation results as an object with the following property:

- *invalidMessage* – a message string displayed under the field when making an attempt to save a page with an invalid field value and in the data window when saving a page with the field that did not passed validation.

If the value validation is successful, the validator method returns the object with empty string.

To start the field validation, the corresponding view model column must be bound to a specific validator. For this purpose, override the *setValidationConfig()* base method and call the *addColumnValidator()* method in it.

The *addColumnValidator()* method accepts two parameters:

- name of the view model column, to which the validator is bound.
- name of the column value validator method.

> ⚠ ATTENTION
>
> If the field is validated in the replacement client schema of the base page, the parent implementation of the *setValidationConfig()* method must be called before calling the *addColumnValidator()* method to correctly initialize validators of the base page fields.

To add validation of field values:

1. Add the validator method to the collection of methods of the view model that will check a field value.
2. Override the *setValidationConfig()* method and connect the validator to the corresponding view model column in it.

Source code

You can download the package with case implementation using the following link.

Example 1

Case description

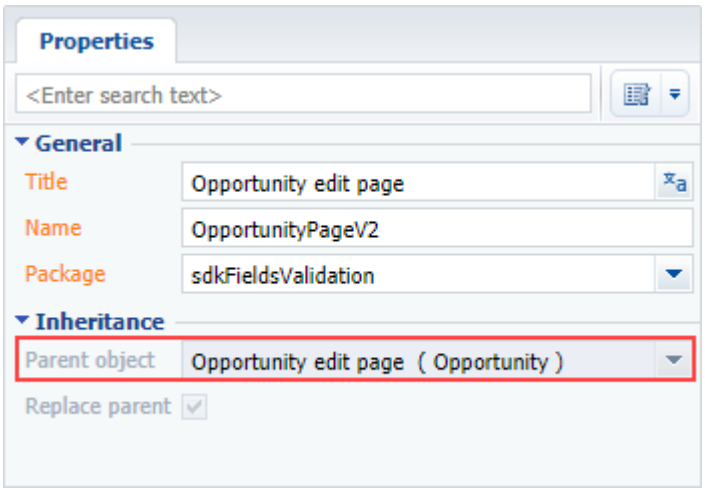
Set the validation on the opportunity page as follows: the date in the [Created on] field must be earlier than the date in the [Closed on] field.

Case implementation algorithm

1. Create a replacement client module of the opportunity edit page

A replacing client module must be created and *[OpportunityPageV2]* must be specified as the parent object in it (Fig. 1). Creating a replacing page is covered in the "**Creating a custom client module schema**" article.

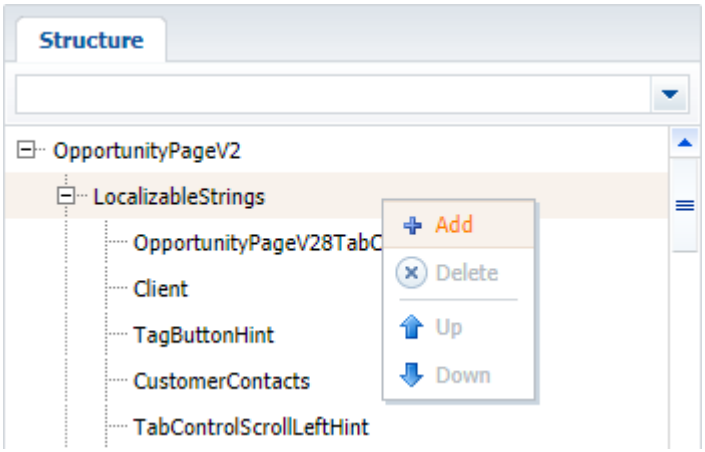
Fig. 1. Properties of the replacing edit page



2. Add an error string to the collection of localizable strings of the page replacing schema

Create a new localizable string (Fig. 2).

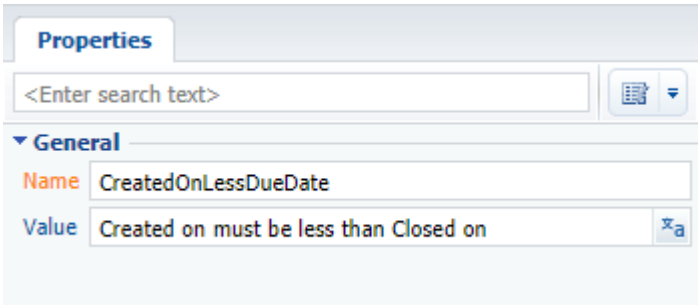
Fig. 2. Adding localized string to the schema



For the created string, specify (Fig. 3):

- [Name] – "CreatedOnLessDueDate".
- [Value] – "Created on must be less than Closed on".

Fig. 3. Properties of the custom localizable string



3. Add the implementation of methods in the *methods* collection of the view model

- *dueDateValidator()* – validator method that determines if the condition is fulfilled.
- *setValidationConfig()* – an overridden base method in which the validator method is bound to the [DueDate] and [CreatedOn] columns.

The replacing schema source code is as follows:

```
define("OpportunityPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Opportunity",
        methods: {
            // Validate method for values in the [DueDate] and [CreatedOn] columns.
            dueDateValidator: function() {
                // Variable for storing a validation error message.
                var invalidMessage = "";
                // Checking values in the [DueDate] and [CreatedOn] columns.
                if (this.get("DueDate") < this.get("CreatedOn")) {
                    // If the value of the [DueDate] column is less than the value
                    // of the [CreatedOn] column a value of the localizable string
is
                    // placed into the variable along with the validation error
message
                    // in the invalidMessage variable.
                    invalidMessage =
this.get("Resources.Strings.CreatedOnLessDueDate");
                }
                // Object whose properties contain validation error messages.
                // If the validation is successful, empty strings are returned to
```

```
the
                 // object.
                return {
                    // Validation error message.
                    invalidMessage: invalidMessage
                };
            },
            // Redefining the base method initiating custom validators.
            setValidationConfig: function() {
                // This calls the initialization of validators for the parent view
model.
                this.callParent(arguments);
                // The dueDateValidator() validate method is added for the [DueDate]
column.
                this.addColumnValidator("DueDate", this.dueDateValidator);
                // The dueDateValidator() validate method is added for the
[CreatedOn] column.
                this.addColumnValidator("CreatedOn", this.dueDateValidator);
            }
        }
    };
});
```

After you save the schema and refresh bpm'online page, a string with the corresponding message (Fig. 4) will appear on the opportunity edit page when entering the date of closing or date of creation which does not satisfy the validation condition (the date of creation must be before than the date of closing). The data window will appear when making an attempt to save the opportunity (Fig. 5).

Fig. 4. Case results: invalid date message

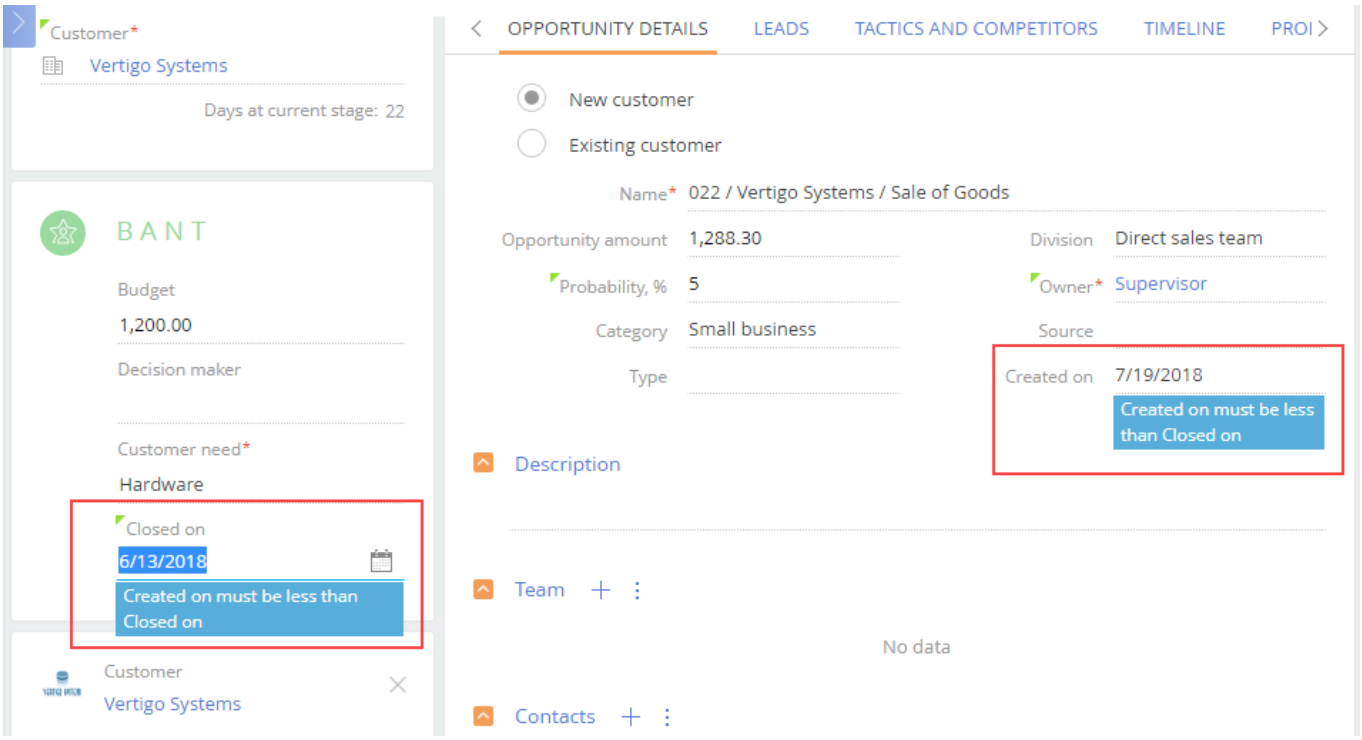
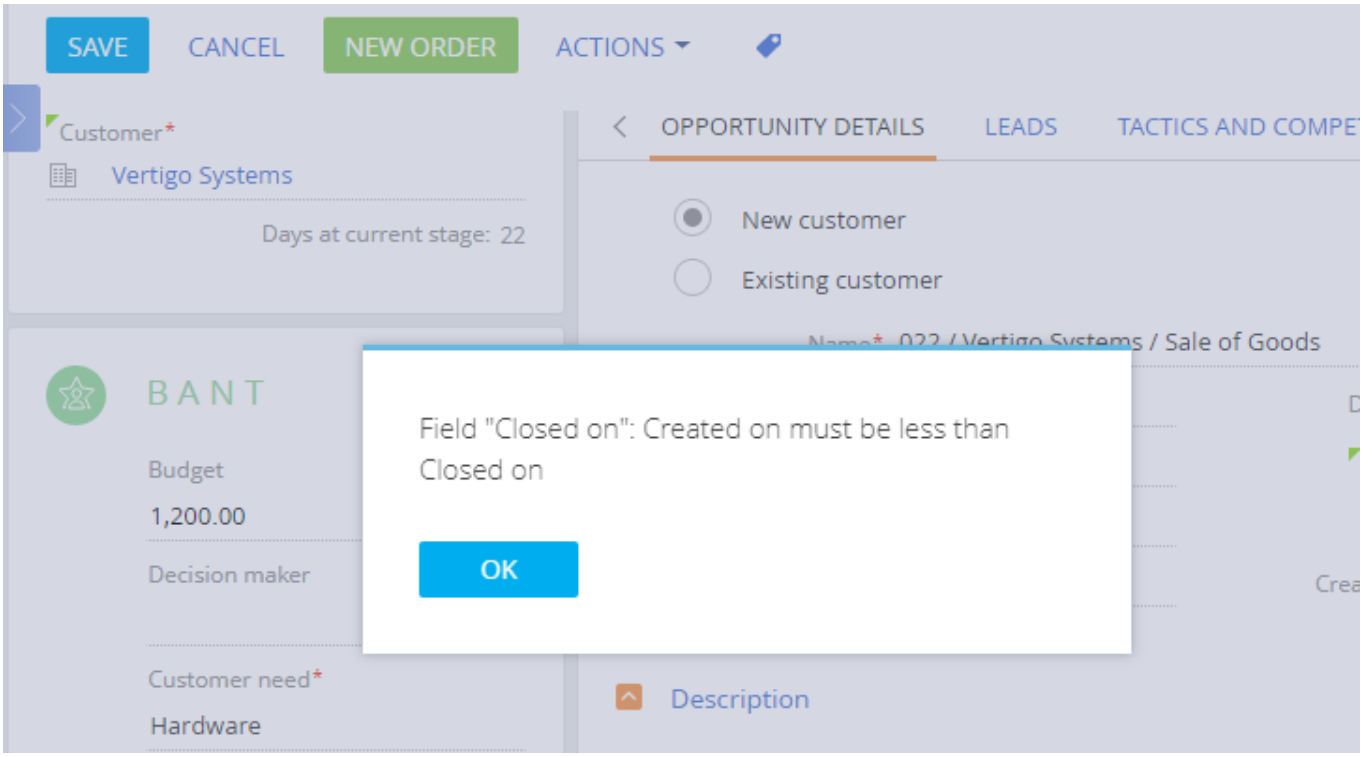


Fig. 5. Case results: message when saving



Example 2

Case description

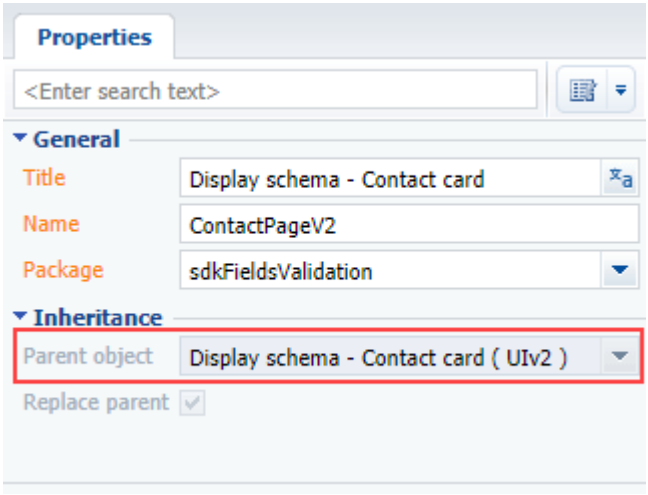
Set the [Business phone] field validation as follows: phone number must correspond to the following mask: +44 XXX XXX XXXX, otherwise the “Enter the number in the “+44 XXX XXX XXXX” format ” message appears.

Case implementation algorithm

1. Create a replacing client module

Create a replacing client module and specify the [Display schema – Contact card] (*ContactPageV2*) schema as parent object (Fig. 6). Creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 6. Properties of the replacing edit page



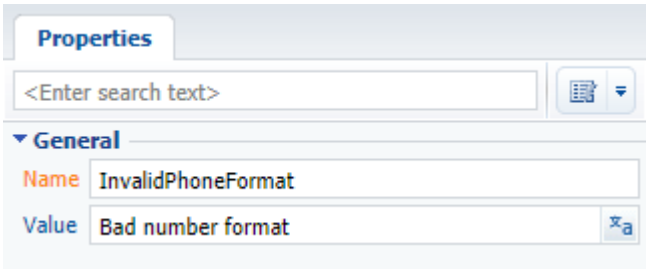
2. Add an error string to the collection of localizable strings of the page replacing schema

Create a new localizable string (Fig. 2).

For the created string, specify (Fig. 7):

- [Name – "InvalidPhoneFormat".
- [Value] – "Enter the number in the "+44 XXX XXX XXXX" format".

Fig. 7. Properties of the custom localizable string



3. Add the implementation of methods in the *methods* collection of the view model

- *phoneValidator()* – validator method that determines if the condition is fulfilled.
- *setValidationConfig()* – an overridden base method in which the validator method is bound to the [Phone] column.

The replacing schema source code is as follows:

```
define("ContactPageV2", ["ConfigurationConstants"], function(ConfigurationConstants)
{
    return {
        entitySchemaName: "Contact",
        methods: {
            // Redefining the base method initiating custom validators.
            setValidationConfig: function() {
                // Calls the initialization of validators for the parent view model.
                this.callParent(arguments);
                // The phoneValidator() validate method is added to the [Phone]
column.
                this.addColumnValidator("Phone", this.phoneValidator);
            },
            phoneValidator: function(value) {
                // Variable for stroing a validation error message.
                var invalidMessage = "";
                // Variable for stroing the number check result.
                var isValid = true;
                // Variable for the phone number.
                var number = value || this.get("Phone");
                // Determining the correctness of the number format using a regular
expression.
                isValid = (Ext.isEmpty(number) ||
                    new RegExp("^\\+44\\s[0-9]{3}\\s[0-9]{3}\\s[0-9]
{4}$").test(number));
                // If the format of the number is incorrect, then an error message is
filled in.
                if (!isValid) {
                    invalidMessage =
this.get("Resources.Strings.InvalidPhoneFormat");
                }
                // Object which properties contain validation error messages.
                // If the validation is successful, empty strings are returned to the
object.
                return {
                    invalidMessage: invalidMessage
                };
```

```
                }
            }
        };
});
```

When the schema is saved and the system web-page is updated, the verification of the number format validity will be preformed on the contact or account edit page when a new phone number is added. If the format is incorrect, a string with a corresponding message will appear (Fig. 8, 9).

Fig. 8. Case results: Message about the incorrect format

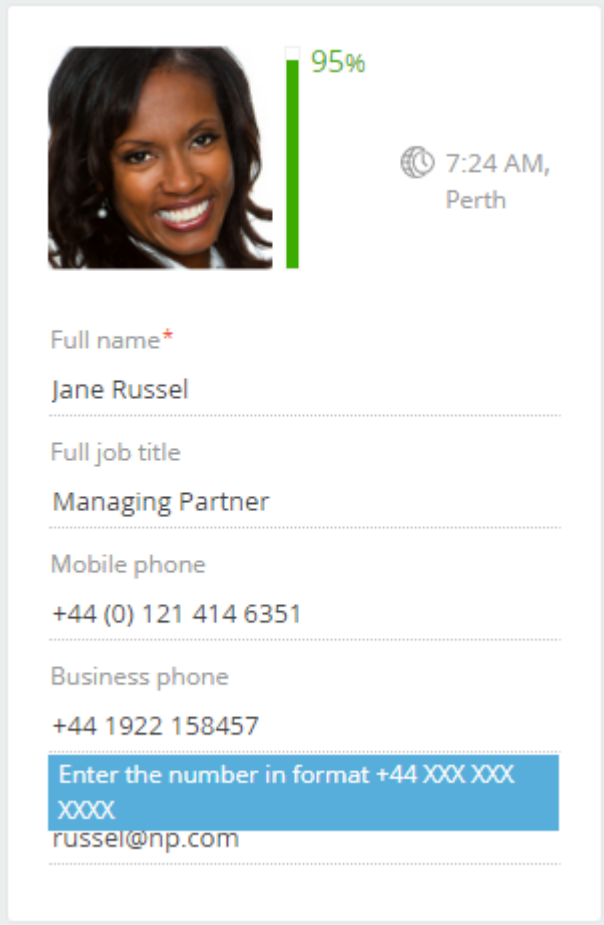
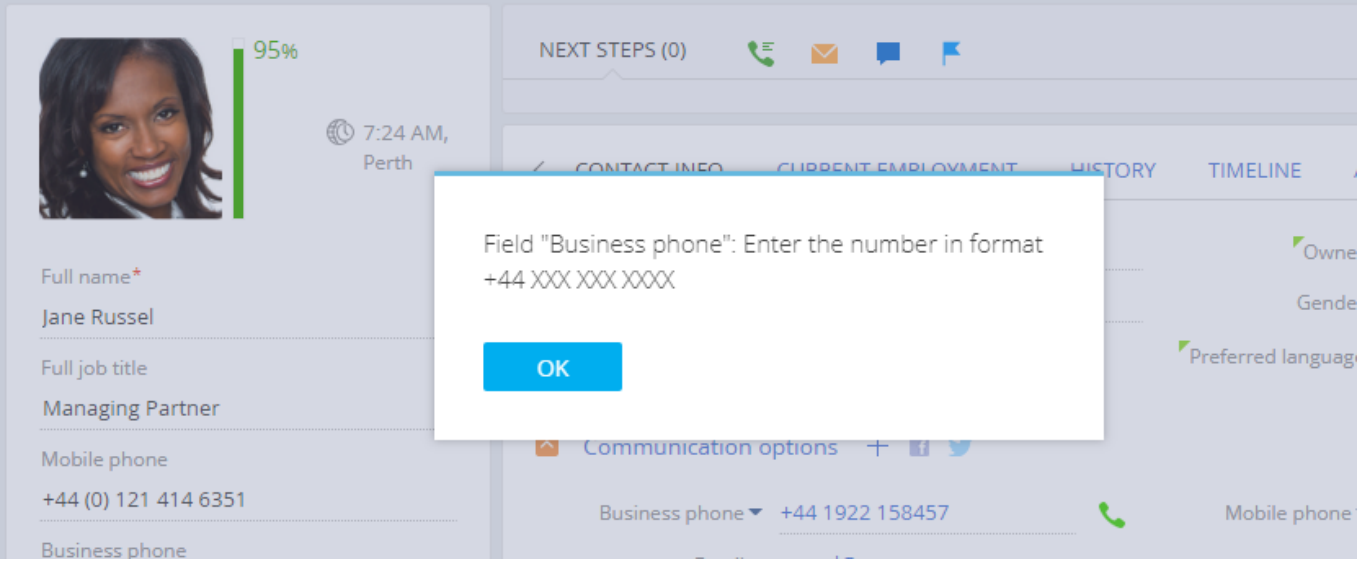
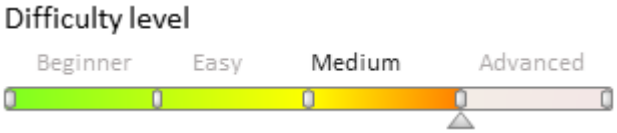


Fig. 9. Case results: message when saving



Using filtration for lookup fields. Examples



Introduction

There are two methods of using the filtration in bpm'online for lookup fields of the edit page:

1. The [FILTRATION] business rule.
2. Explicit indication of filters in the column description of the attributes model property.

The use of the [FILTRATION] business rule is expedient if a simple filter by a specific value or attribute must be used for the field. The business rules are detailed in the **Setting the edit page fields using business rules**. The detailed case for using the [FILTRATION] business rule is set forth in the **The FILTRATION rule use case** article.

If arbitrary filtration (sorting and addition of supplementary columns to a query when a drop-down list is displayed) is required, the explicit description should be used in the *attributes* model property.

Setting lookup field filters in the *attributes* model property:

1. The name of the column for which filters are set must be added to the *attributes* property of the view model.
2. The *lookupListConfig* property must be declared for this column. It represents a configuration item containing the following properties (not required):
   - *columns* – an array of column names to be added to a request in addition to Id and the primary display column.
   - *orders* – an array of configuration objects determining the data sorting when displayed.
   - *filter* – the method for returning the object of the *Terrasoft.BaseFilter* class or its inheritor, will be applied, in turn, to a request.
   - or *filters* – an array of filters (methods for returning collections of the *Terrasoft.FilterGroup* class).

Filters are added to a collection using the *add()* method which has the following parameters:

Name	Data type	Description
key	String	key
item	Mixed	Element.
index	Number	Index for insert. If not entered, the index to be inserted is not rated.

The object of the *Terrasoft.BaseFilter* class or its inheritor is the *item* parameter. The methods for creating filters with descriptions are given in Table 1 of the **EntitySchemaQuery filters handling** article.

> ⚠ ATTENTION
>
> Filters are combined by default in the collection using the AND logic operator. If the OR operator is to be used, this must be indicated explicitly in the *logicalOperation* property of the *Terrasoft.FilterGroup* object.

Case description

When a value is added to the [Owner] field of the account edit page, display only those contact lookup values for which the following conditions are fulfilled:

- a system user associated with this contact is available
- this user is active.

Source code

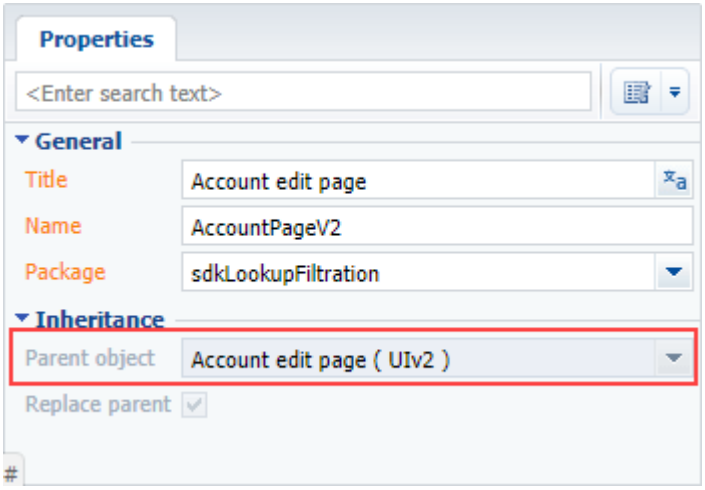
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Create a replacing account edit page

A replacing client module must be created and *[AccountPageV2]* must be specified as the parent object in it (Fig. 1). The procedure of creating a replacing page is covered in the"**Creating a custom client module schema**" article.

Fig. 1. Properties of the replacing edit page



2. Add the attribute with filtration to the attributes property of the view model

Specify the type of column data – the *Terrasoft.DataValueType.LOOKUP* in the configuration object, in the [Owner] attribute and describe the configuration object of the *lookupListConfig* lookup field. Add the filters property to *lookupListConfig*, which represents the function for returning the *filters* collection. Add a function that returns a collection of filters to the array.

The replacing schema source code is as follows:

```
define("AccountPageV2", [], function() {
    return {
        // Name of the edit page object schema
        "entitySchemaName": "Account",
        // List of the schema attributes.
        "attributes": {
            // Name of the view model column.
            "Owner": {
                // Attribute data type.
                "dataValueType": Terrasoft.DataValueType.LOOKUP,
                // The configuration object of the LOOKUP type.
                "lookupListConfig": {
                    // Array of filters used for the query that forms the lookup
field data.
                    "filters": [
                        function() {
                            var filterGroup = Ext.create("Terrasoft.FilterGroup");
                            // Adding the "IsUser" filter to the resulting filters
collection.
                            // The filter provides for the selection of all records
in the Contact core schema
                            // to which the Id column from the SysAdminUnit schema is
connected, for which
```

```
                            // Id is not equal to null.
                            filterGroup.add("IsUser",
                                Terrasoft.createColumnIsNotNullFilter("
[SysAdminUnit:Contact].Id"));
                                                        // Adding the "IsActive"
filter to the resultant filters collection.
                            // The filter provides for the selection of all records
from the core schema.
                            // Contact to which the Id column from the SysAdminUnit
schema, for which
                            // Active=true, is connected.
                            filterGroup.add("IsActive",
                                Terrasoft.createColumnFilterWithParameter(
                                    Terrasoft.ComparisonType.EQUAL,
                                    "[SysAdminUnit:Contact].Active",
                                    true));
                            return filterGroup;
                        }
                    ]
                }
            }
        }
    };
});
```

When the schema is saved and the system web-page is updated, only values from the contact lookup which comply with custom conditions will be displayed on the account edit page when adding a value to the [Owner] field on the account edit page (Fig. 2, Fig. 3). I.e:

- a system user associated with this contact is available
- this user is active.

Fig. 2. Account profile with the owner

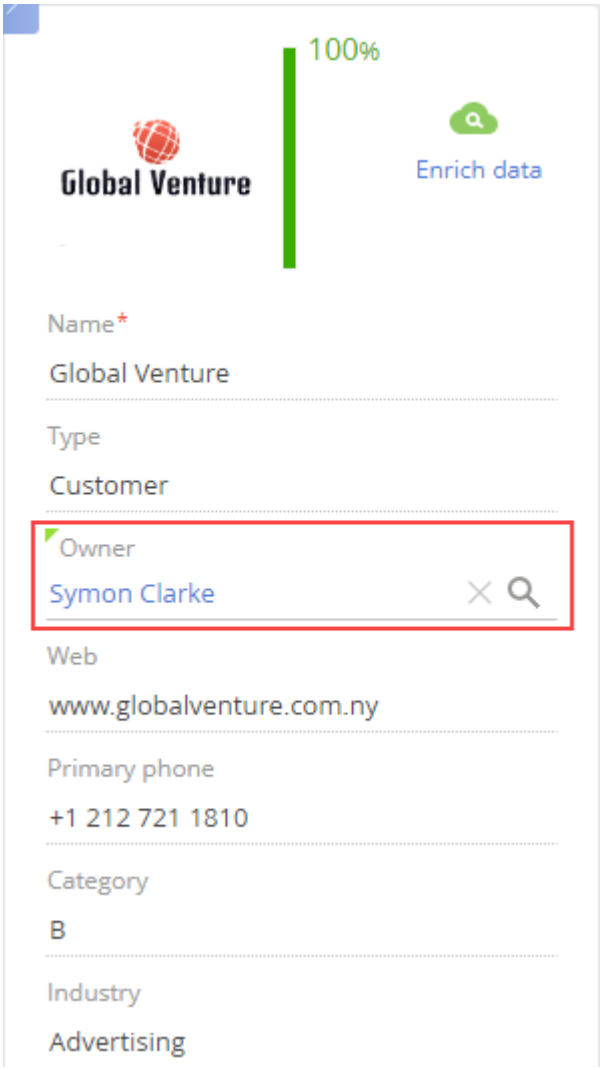
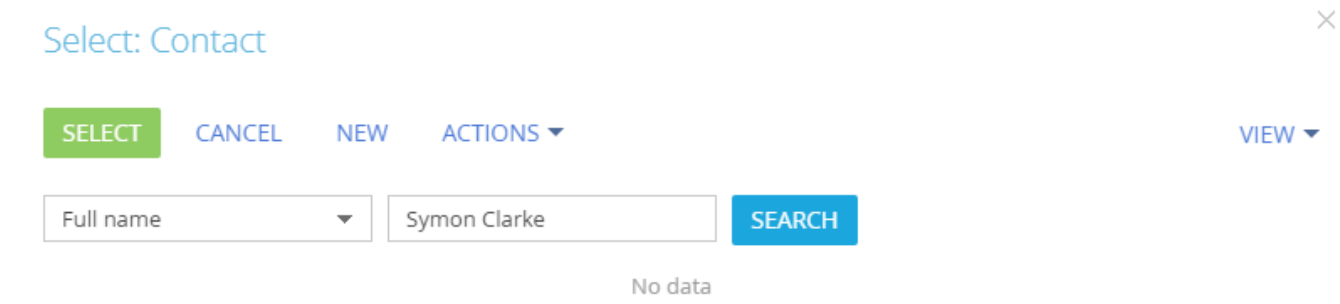
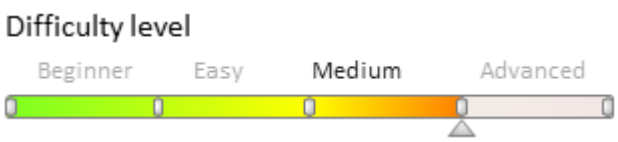


Fig. 3. The owner is disable in the filtered contact lookup



Adding an action panel



Introduction

Starting with version 7.8.0, bpm'online has a new edit page module – the "Action panel" (ActionsDashboard). An action panel displays information about the current status of and actions for working with the current record.

For more information about action panel, please see the "**Action dashboard**" article.

General procedure of adding an action panel on a page:

1. Create a Schema of the Edit Page View Model inherited from the *SectionActionsDashboard* module.
2. Create a replacing page schema.
3. Set up the module in the *modules* property of the page view model.
4. In the "diff" array of the page view model, add the module on the page.

Case description

Add an action panel to the order edit page.

Source code

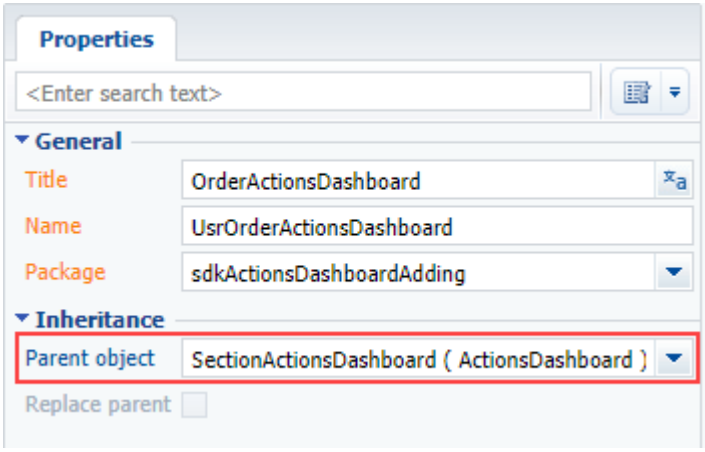
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Create a client schema of the OrderActionsDashboard view model

Specify the *SectionActionsDashboard* schema as a parent object (Fig. 1).

Fig. 1. Properties of the client schema



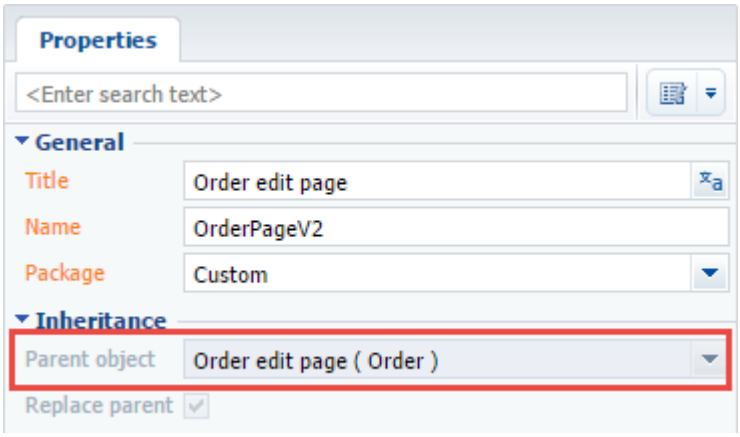
The client schema source code is as follows:

```
define("UsrOrderActionsDashboard", [], function () {
    return {
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        methods: {},
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    };
});
```

2. Create a replacing order edit page

A replacing client module must be created and [Order edit page] (*OrderPageV2*) must be specified as the parent object in it (Fig. 2). Creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 2. Properties of the replacing edit page



3. Add a configuration object with the module settings in the modules collection of the page schema

Add the code of the page replacing module to the [Source code] tab: Add a configuration object with the module settings in it to the *modules* collection of the view model.

4. Add a configuration object with the settings determining the module position in the diff array

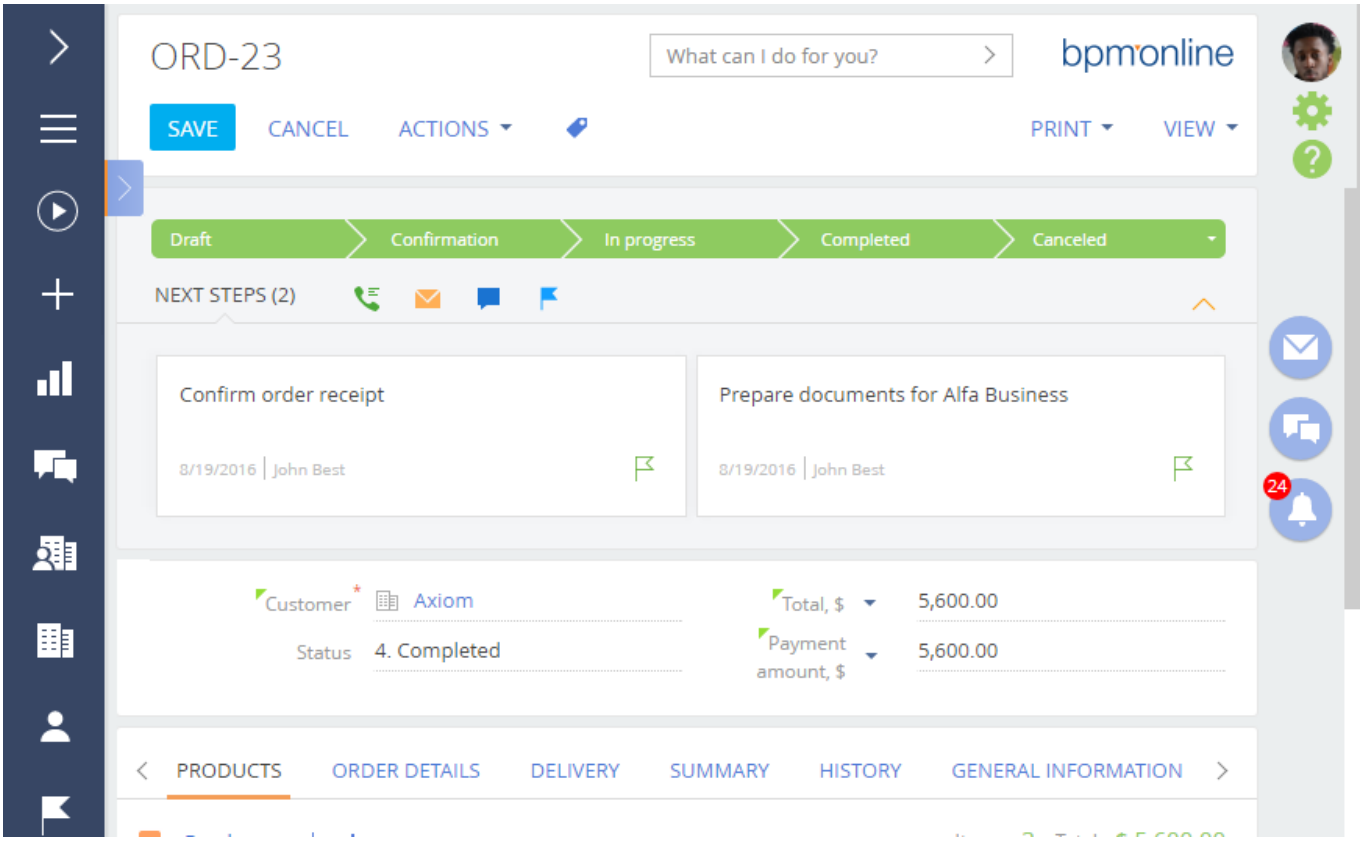
The replacing schema source code is as follows:

```
define("OrderPageV2", [],
    function () {
        return {
            entitySchemaName: "Order",
            attributes: {},
            modules: /**SCHEMA_MODULES*/{
                "ActionsDashboardModule": {
                    "config": {
                        "isSchemaConfigInitialized": true,
                        // Schema name.
                        "schemaName": "OrderActionsDashboard",
                        "useHistoryState": false,
                        "parameters": {
                            // Configuration object of the module.
                            "viewModelConfig": {
                                // Schema name of the page entity.
                                "entitySchemaName": "Order",
                                // Configuration object of the Actions block.
                                "actionsConfig": {
                                    // Schema name for loading items to Actions.
                                    "schemaName": "OrderStatus",
                                     // Column name in the parent schema that
references the schema that contains Actions elements.
                                     // If not specified, takes the schemaName value.
                                    "columnName": "Status",
                                    // Column name for element sorting.
                                    "orderColumnName": "Position",
                                    // Column name for item sorting in the item menu.
                                    "innerOrderColumnName": "Position"
                                },
                                // Displaying the action panel module, the value is
[true] by default.
                                "useDashboard": true,
                                // Displaying the Content block, [true] by default.
                                "contentVisible": true,
                                // Risplaying the Header block, [true] by default.
```

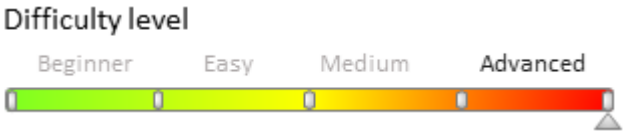
```
                                "headerVisible": true,
                                // The configuration object of the panel elements.
                                "dashboardConfig": {
                                    // Connection between activities and page object.
                                    "Activity": {
                                        // Page object column name.
                                        "masterColumnName": "Id",
                                        // Clumn name in the [Activity] object.o
                                        "referenceColumnName": "Order"
                                    }
                                }
                            }
                        }
                    }
                }
            }/**SCHEMA_MODULES*/,
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            methods: {},
            diff: /**SCHEMA_DIFF*/[
                {
                    "operation": "insert",
                    "name": "ActionsDashboardModule",
                    "parentName": "ActionDashboardContainer",
                    "propertyName": "items",
                    "values": {
                        "classes": { wrapClassName: ["actions-dashboard-module"] },
                        "itemType": Terrasoft.ViewItemType.MODULE
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
    });
```

After saving the schema and updating the bpm'online web page, the action panel will be added to the order page. The action panel will contain the order status and connected uncompleted activities (Fig. 3).

Fig. 3. Demonstrating the case implementation result



Adding a new channel to the action panel



Introduction

Starting with version 7.8.0, bpm'online has a new edit page module – the "Action panel" (ActionsDashboard). An action panel displays information about the current status of and actions for working with the current record.

For more information about action panel, please see the "**Action dashboard**" article. The ways of adding the action panel to the section page are described in the "**Adding an action panel**" article.

ActionsDashboard channels are a way of communicating with a contact. A channel is created for every section in which it's connected to, for example, a case, contact or lead.

Case description

Add a new custom channel to the action dashboard of the contact edit page. The channel must have the same functionality as the call results channel (*CallMessagePublisher* channel).

Source code

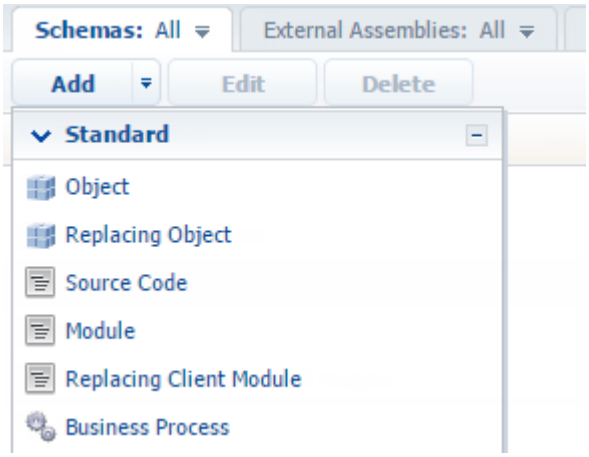
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Add the *UsrCallsMessagePublisher* source code schema

Perform the [Add] > [Source code] menu command on the [Schema] tab in the [Configuration] section (Fig. 1).

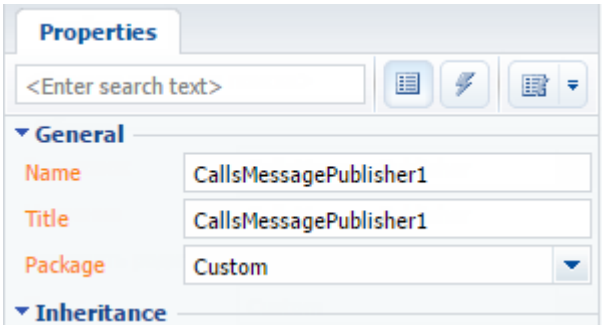
Fig. 1. Adding source code schema



For the created schema specify (Fig. 2):

- [Title] – "Call message logging publisher"
- [Name] – "UsrCallsMessagePublisher"

Fig. 2. The Source code schema properties



In the created schema, add the new *CallsMessagePublisher* class inherited from the *BaseMessagePublisher* class to the *Terrasoft.Configuration* namespace. The *BaseMessagePublisher* class contains the basic logic to save an object in the database and the basic logic of event handlers. The inheritor class will contain the logic for a particular sender, for example, filling of columns of the *Activity* object and the subsequent sending of the message.

To implement the new *CallsMessagePublisher* class, you must add the following source code in the created schema.

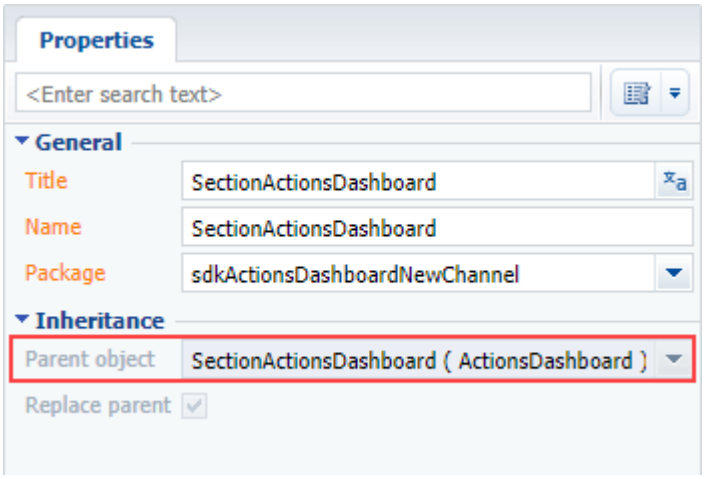
```
using System.Collections.Generic;
using Terrasoft.Core;

namespace Terrasoft.Configuration
{
    // The BaseMessagePublisher heir class.
    public class CallsMessagePublisher : BaseMessagePublisher
    {
        // Class constructor.
        public CallsMessagePublisher(UserConnection userConnection,
Dictionary<string, string> entityFieldsData)
            : base(userConnection, entityFieldsData) {
            //The schema the CallsMessagePublisher works with.
            EntitySchemaName = "Activity";
        }
    }
}
```

Save and publish the schema.

2. Create the SectionActionsDashboard replacing client schema

Create a replacing client module and specify the *SectionActionsDashboard* as parent object (Fig. 3). The procedure of creating a replacing page is covered in the "**Creating a custom client module schema**" article.

Fig. 3. Properties of the replacing schema



NOTE

If you want to add a channel to only one edit page, you must create a new module named [*section_name*]*SectionActionsDashboard* (e.g. *BooksSectionActionsDashboard*) and set *SectionActionsDashboard* as the parent schema.

Specify the module which should be rendered in this channel on one of the tabs in the replacing schema *diff* property. Set the operations of inserting the *CallsMessageTab* tab and message container in this property. The new channel will be visible on the edit pages of those sections, which are connected to *SectionActionsDashboard*.

In the *methods* property override the *getSectionPublishers()* method that will add the new channel to the list of message publishers, and the *getExtendedConfig()* method, in which the tab settings are configured.

For the *getExtendedConfig()* method to run correctly, you must upload the channel icon and specify it in the *ImageSrc* parameter. The icons used in this example can be downloaded **here ('CallsMessageTabImage.svg' in the on-line documentation)**.

You should also override the *onGetRecordInfoForPublisher()* method and add the *getContactEntityParameterValue()* method that defines the contact value from the edit page.

The replacing schema source code is as follows:

```
define("SectionActionsDashboard", ["SectionActionsDashboardResources",
"UsrCallsMessagePublisherModule"],
    function(resources) {
        return {
            attributes: {},
            messages: {},
            methods: {
                // Method sets the channel tab display settings in the action
dashboard.
                getExtendedConfig: function() {
                    // Parent method calling.
                    var config = this.callParent(arguments);
                    var lczImages = resources.localizableImages;
                    config.CallsMessageTab = {
                        // Tab image.
                        "ImageSrc":
this.Terrasoft.ImageUrlBuilder.getUrl(lczImages.CallsMessageTabImage),
                        // Marker value.
                        "MarkerValue": "calls-message-tab",
```

```
                        // Alignment.
                        "Align": this.Terrasoft.Align.RIGHT,
                        // Tag.
                        "Tag": "UsrCalls"
                    };
                    return config;
                },
                // Redefines the parent object and adds the contact value from the
edit page
                // of the section that contains the action dashboard.
                onGetRecordInfoForPublisher: function() {
                    var info = this.callParent(arguments);
                    info.additionalInfo.contact =
this.getContactEntityParameterValue(info.relationSchemaName);
                    return info;
                },
                // Defines the contact value from the section edit page
                // that contains the action dashboard.
                getContactEntityParameterValue: function(relationSchemaName) {
                    var contact;
                    if (relationSchemaName === "Contact") {
                        var id = this.getMasterEntityParameterValue("Id");
                        var name = this.getMasterEntityParameterValue("Name");
                        if (id && name) {
                            contact = {value: id, displayValue: name};
                        }
                    } else {
                        contact = this.getMasterEntityParameterValue("Contact");
                    }
                    return contact;
                },
                //Adds the created channel to the message publisher list.
                getSectionPublishers: function() {
                    var publishers = this.callParent(arguments);
                    publishers.push("UsrCalls");
                    return publishers;
                }
            },
            // An array of modifications, with which the representation of the module
is built in the interface of the system.
            diff: /**SCHEMA_DIFF*/[
                // ДAdding the CallsMessageTab tab.
                {
                    // operation type — insertion.
                    "operation": "insert",
                    // Tab name.
                    "name": "CallsMessageTab",
                    // Parent element name.
                    "parentName": "Tabs",
                    // Property name.
                    "propertyName": "tabs",
                    // Property configuration object.
                    "values": {
                        // Child elements array.
                        "items": []
                    }
                },
                // Adding message container.
                {
                    "operation": "insert",
                    "name": "CallsMessageTabContainer",
                    "parentName": "CallsMessageTab",
```

```
                    "propertyName": "items",
                    "values": {
                        // Element type — container.
                        "itemType": this.Terrasoft.ViewItemType.CONTAINER,
                        // Container CSS class.
                        "classes": {
                            "wrapClassName": ["calls-message-content"]
                        },
                        "items": []
                    }
                },
                // Adding the UsrCallsMessageModule module.
                {
                    "operation": "insert",
                    "name": "UsrCallsMessageModule",
                    "parentName": "CallsMessageTab",
                    "propertyName": "items",
                    "values": {
                        // Tab module CSS class.
                        "classes": {
                            "wrapClassName": ["calls-message-module", "message-
module"]
                        },
                        // Element type — module.
                        "itemType": this.Terrasoft.ViewItemType.MODULE,
                        // Module name.
                        "moduleName": "UsrCallsMessagePublisherModule",
                        // Binding the method executed after the element has been
rendered.
                        "afterrender": {
                            "bindTo": "onMessageModuleRendered"
                        },
                        // Binding the method executed after the element has been
rerendered.
                        "afterrerender": {
                            "bindTo": "onMessageModuleRendered"
                        }
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
    }
);
```

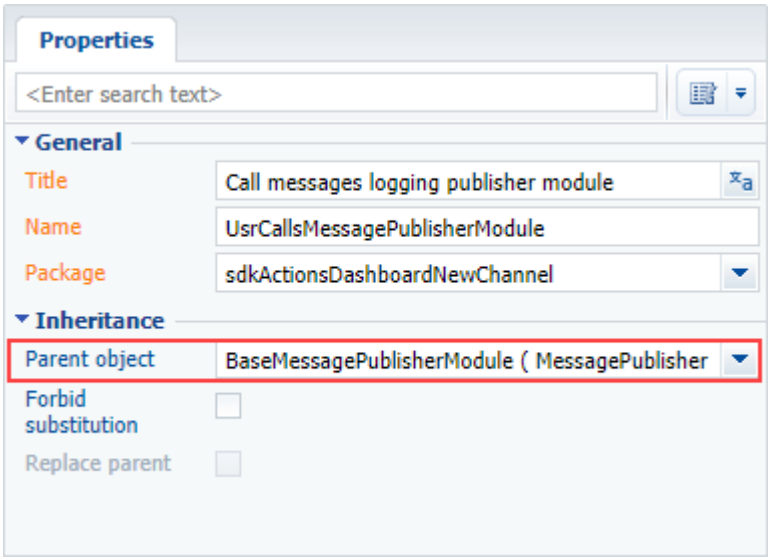
3. Create the UsrCallsMessagePublisherModule module

The *UsrCallsMessagePublisherModule* serves as container that renders the *SectionActionsDashboard* page with implemented logic of added channel in the *UsrCallsMessagePublisherPage*.

Set following properties for the module (Fig. 4):

- [Title] – "Call messages logging publisher module"
- [Name] – "UsrCallsMessagePublisherModule"
- [Parent object] – *BaseMessagePublisherModule*.

Fig. 4. Properties of the module



The module source code:

```
define("UsrCallsMessagePublisherModule", ["BaseMessagePublisherModule"],
    function() {
        // Defining the class.
        Ext.define("Terrasoft.configuration.UsrCallsMessagePublisherModule", {
            // Basic class.
            extend: "Terrasoft.BaseMessagePublisherModule",
            // Short class name.
            alternateClassName: "Terrasoft.UsrCallsMessagePublisherModule",
            // Initialization of the page that will be rendered in this module.
            initSchemaName: function() {
                this.schemaName = "UsrCallsMessagePublisherPage";
            }
        });
        // Returns the class object defined in the module.
        return Terrasoft.UsrCallsMessagePublisherModule;
    });
```

4. Create the UsrCallsMessagePublisherPage page

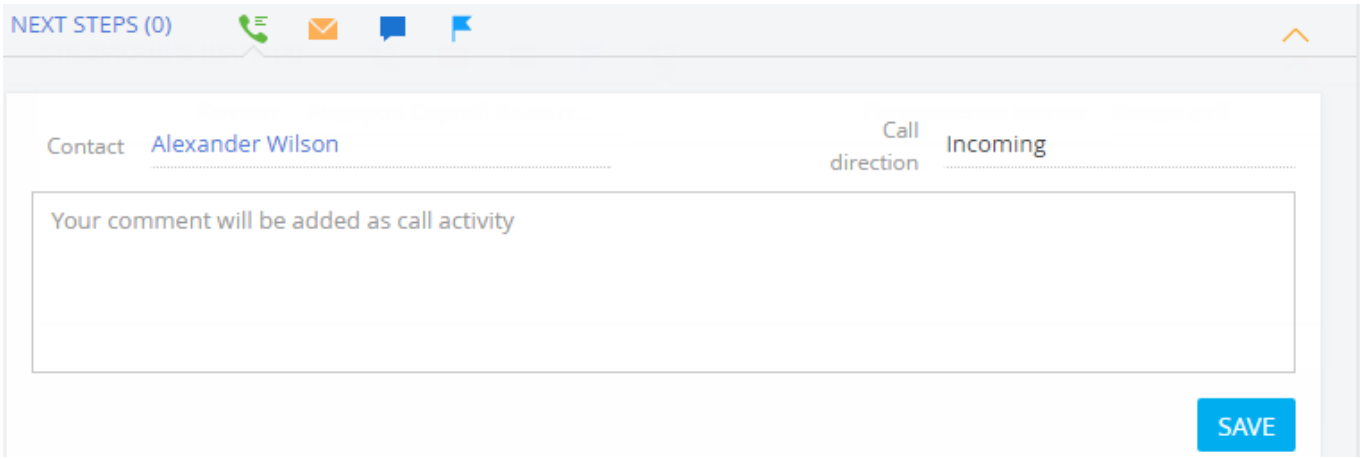
For the created page set the *BaseMessagePublisherPage* schema of the *MessagePublisher* package as parent object. Set the "UsrCallsMessagePublisherPage" value as the title and name.

In the source code page, specify the schema name of the object that will run along with the page (in this case, *Activity*), implement the logic of message publication and override the *getServiceConfig* method, in which you must set the class name from the configuration.

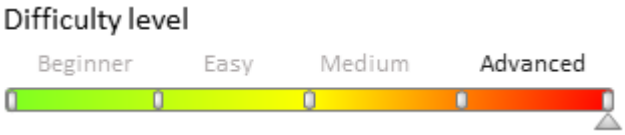
```
//Sets the class that will work with this page.
getServiceConfig: function() {
   return {
      className: "Terrasoft.Configuration.CallsMessagePublisher"
  };
}
```

Implementation of message publication logic contains big number of methods, attributes and properties. The full source code of the *UsrCallsMessagePublisherPage* schema can be found in the sdkActionsDashboardNewChannel package. The source code shows the implementation of the *CallMessagePublisher* channel that is used for logging incoming and outgoing calls.
As a result you will get the new channel in the *SectionActionsDashboard* (Fig. 5).

Fig. 5. An example of a custom CallsMessagePublisher channel in the SectionActionsDashboard of the [Contacts] section.



Displaying contact's time zone



General information

The ability to work with different timezones was introduced in version 7.8. Contact pages now display information about the contact's local timezone. Timezone values, such as time difference between the user's timezone and the contact's timezone, are calculated automatically. For more information about timezone calculation please see the "How to determine the current local time for a contact". The information is displayed in the element generated by the view generator. This element cannot be added to a page with the section wizard.

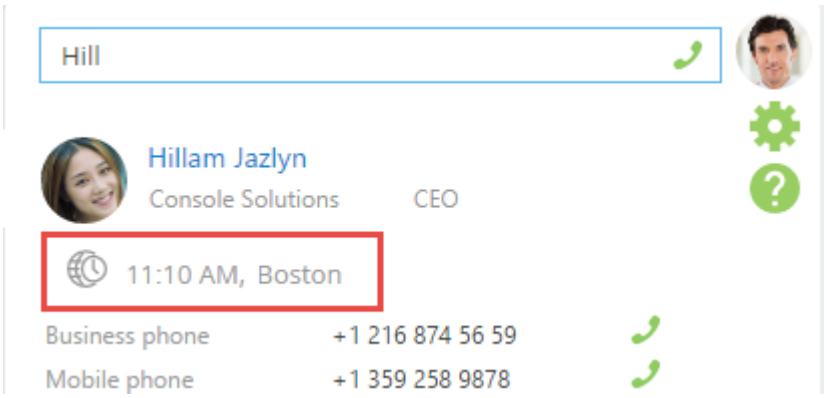
To add a contact timezone display element to a custom page:

1. Create a replacing page schema module.
2. Add timezone display element.
3. Connect timezone search.
4. Set up display styles.

Case description

Add a contact timezone display element to the call panel. The contact's current local time must be displayed when searching for phone numbers to ensure that subscribers in different timezones are contacted during business hours only.

Fig. 1. Displaying subscriber's current local time



Source code

You can download the package with case implementation using the following link.

Case implementation algorithm

1. Create a replacing page schema module

To modify this schema, add a replacing client module in the custom package (Fig. 2) and specify *SubscriberSearchResultItem* as the parent object and custom package (Fig. 3). Creating a replacing page is covered in the "**Creating a custom client module schema**" article.

Fig. 2. Creating a replacing module

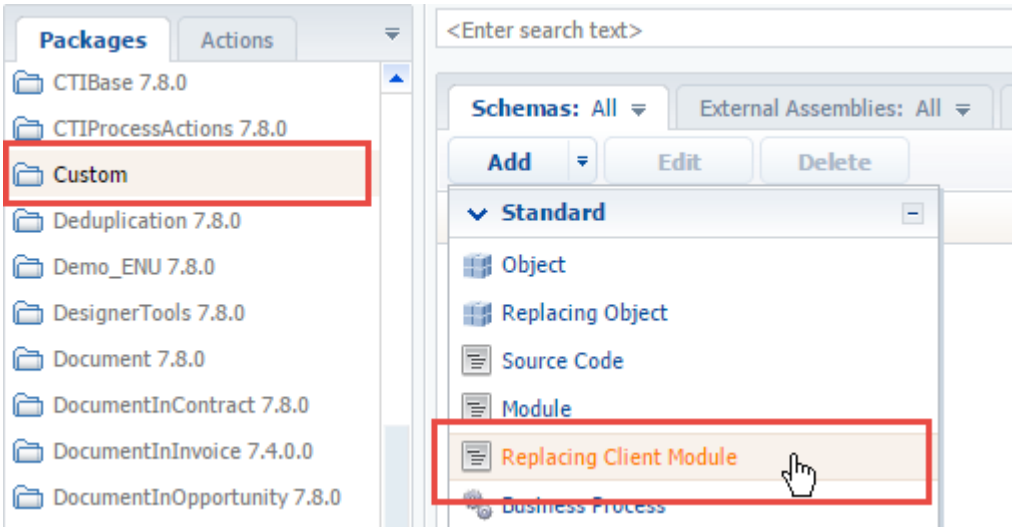
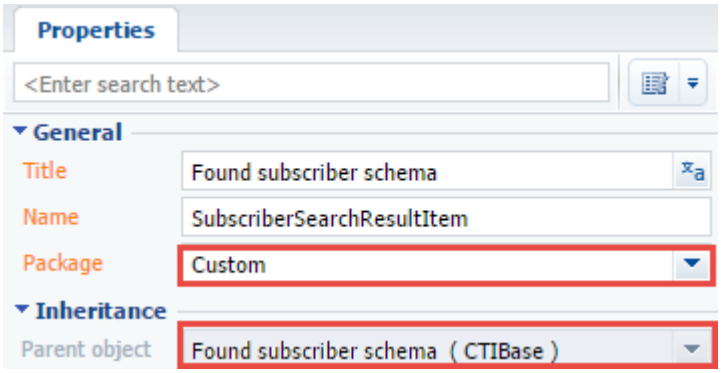


Fig. 3 Replacing module properties



2. Add timezone display element

Add modules according to the schema dependency:

- *TimezoneGenerator* – module that helps to create contact's timezone display item.
- *TimezoneMixin* – mixin is used to search for contact's timezone.

Add configuration object for displaying a contact's timezone item to the *diff* array.

3. Connect timezone search

To run the contact timezone search, pass the contact's unique Id to the *init* method of the *TimezoneMixin*. As a result of execution the following attributes will be set:

- *TimeZoneCaption* – contact's timezone name and current local time.
- *TimeZoneCity* – name of the city for which the timezone is set.

Source code of the schema:

```
// Declaring schema.
define("SubscriberSearchResultItem",
    // Dependency from TimezoneGenerator, TimezoneMixin.
    ["TimezoneGenerator", "TimezoneMixin",
    // Dependency from the module with style.
    "css!UsrSubscriberSearchResultItemCss"],
        function() {
            return {
                // Block for creating attributes.
                attributes: {
                    // Namne of the attribute that controls timexone element display
status.
                    "IsShowTimeZone": {
                        // Attribute data type.
                        "dataValueType": Terrasoft.DataValueType.BOOLEAN,
                        // Attribute type in model.
                        "type": Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                        // Default value.
                        "value": true
                    }
                },
                // Mixin connection block.
                mixins: {
                    // Connecting mixin.
                    TimezoneMixin: "Terrasoft.TimezoneMixin"
                },
                // Method definition block.
                methods: {
                    // Class constructor.
                    constructor: function() {
                        // Selecting base constructor.
                        this.callParent(arguments);
                        // Indicates if the subscriber is a contact.
                        var isContact = this.isContactType();
                        // The element is displayed if the subscriber is a contact.
                        this.set("IsShowTimeZone", isContact);
                        // If the subscriber is a contact.
                        if (isContact) {
                            // Contact Id.
                            var contactId = this.get("Id");
                            // Searching for contact's timezone.
                            this.mixins.TimezoneMixin.init.call(this, contactId);
                        }
                    },
                    // Gets an indicator that the subscriber is a contact.
                    isContactType: function() {
                        // Subscriber type.
                        var type = this.get("Type");
                        // Gets comparison result.
                        return type === "Contact";
                    }
                },
                // Array of modifications.
                diff: [
                    {
                        // Adding a new element.
                        "operation": "insert",
                        // Parent element is SubscriberSearchResultItemContainer.
                        "parentName": "SubscriberSearchResultItemContainer",
                        // New element is added to the collection of elements of the
```

```
parent.
                        "propertyName": "items",
                        // Element name.
                        "name": "TimezoneContact",
                        // Element properties.
                        "values": {
                            // Element type.
                            "itemType": Terrasoft.ViewItemType.CONTAINER,
                            // Generator method is called for generating view
configuration.
                            "generator": "TimezoneGenerator.generateTimeZone",
                            // Container visibility is bound to an attribute.
                            "visible": {"bindTo": "IsShowTimeZone"},
                            // Element style.
                            "wrapClass": ["subscriber-data", "timezone"],
                            // Binding title to attribute.
                            "timeZoneCaption": {"bindTo": "TimeZoneCaption"},
                            // Binding cisty to attribute.
                            "timeZoneCity": {"bindTo": "TimeZoneCity"}
                        },
                        // Element position in the parent container.
                        "index": 2
                    }
                ]
            };
        });
```

As a result, the contact's current local time and city are displayed.

4. Display style setup

During the previous steps, the configuration object placed in the *diff* array already has preliminary display settings.

Use the *index* property to adjust element positioning. By default, the elements are placed one after another in the *SubscriberSearchResultItemContainer*:

- the first element is subscriber photo with index “0”,
- then subscriber information with index "1",
- then subscriber phone numbers with index "2".

If you set the index value to "2", the element will be displayed between subscriber information and the list of phone numbers.

Use the *subscriber-data* CSS-class to set styles for text elements in the schema. The element generator provides the *wrapClass* property to manage styles.

Create the *UsrSubscriberSearchResultItemCss* module in the custom package for positioning of the element and its visual highlighting.

In the LESS tab of the created module, add CSS-selectors for classes that will determine the required styles.

```
/* Display style setup for the added element.*/
.ctiPanelMain .search-result-items-list-container .timezone {
    /* Top padding.*/
    padding-top: 13px;
    /* Bottom margin.*/
    margin-bottom: -10px;
}
/* Setting styles to display contact time.*/
.ctiPanelMain .search-result-items-list-container .timezone-caption {
    /* Left padding.*/
    padding-left: 10px;
    /* Text color.*/
    color: rgb(255, 174, 0);
    /* Text font - bold.*/
```

```
    font-weight: bold;
}
/* Display style setup for contact city.*/
.ctiPanelMain .search-result-items-list-container .timezone-city {
    /* Left padding.*/
    padding-left: 10px;
}
```

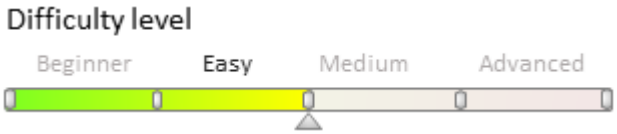
To load module with styles, add the following code to the module and save the schema.

```
define("UsrSubscriberSearchResultItemCss", [],
        function() {
            return {};
        });
```

Add the *UsrSubscriberSearchResultItemCss* module to the *SubscriberSearchResultItem* dependencies.

After saving the schema and refreshing the application page, the subscriber search results will be displayed as shown on Fig. 1.

How to display the difference between dates on edit page fields



Bpm'online uses the capabilities of the standard *Date* JavaScript-object to work with dates on the client's side of the application. For example, the *Date.prototype.getDate()* method is used to display the day of the month for a specific date in accordance with the local time, and the *Date.prototype.setDate()* method is used to set the day of the month relative to the current month. All properties and methods of the *Date* object may be found in the documentation.

For example, when creating a new contract, the [End Date] field should display a date that is 5 days longer than the [Start Day] field. To do this:

1. Create a replacing *ContactPageV2* edit page schema of the [Contracts] section. The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**".

2. Add the following code to the created module:

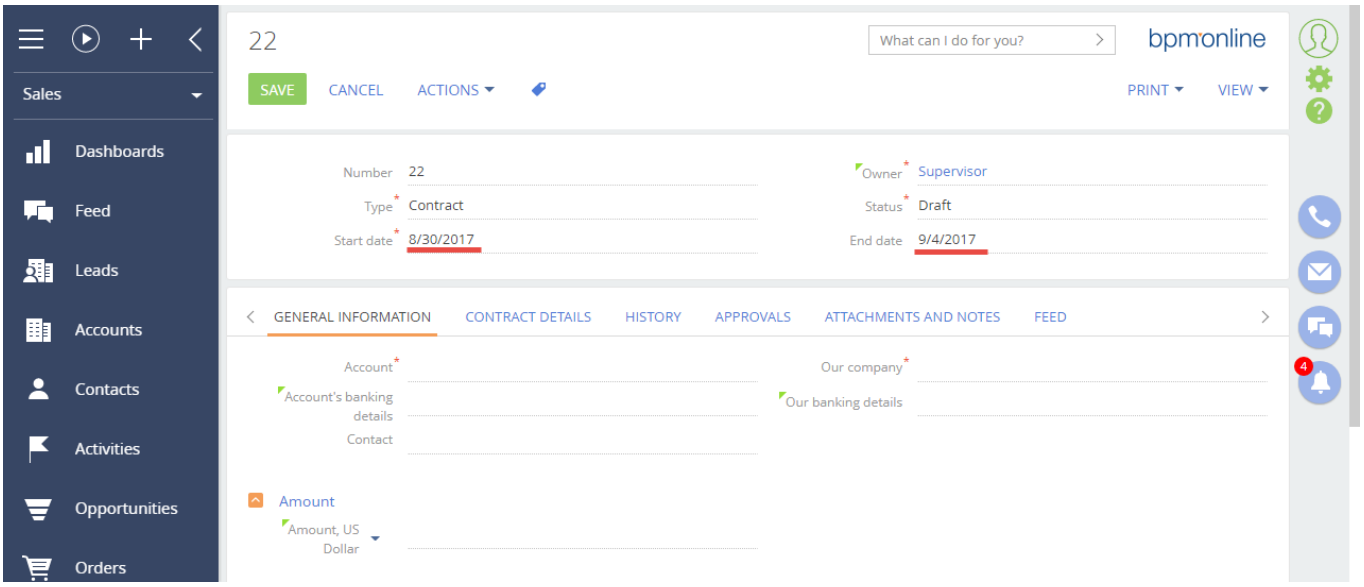
```
define("ContractPageV2", [], function() {
    return {
        entitySchemaName: "Contract",
        methods: {
            // The date is set after the object is initialized.
            onEntityInitialized: function() {
                // Checking the mode of the new record.
                if ((this.isAddMode() && this.Ext.isEmpty(this.get("EndDate")))) {
                    // Calling an auxiliary method.
                    this.setEndDate(this.get("StartDate"), 5);
                }
                // Calling the base functionality.
                this.callParent(arguments);
            },
            // Auxiliary method for setting the date.
            setEndDate: function(date, dateOffsetInDays) {
                var offsetDate = new Date();
                offsetDate.setDate(date.getDate() + dateOffsetInDays);
                this.set("EndDate", offsetDate);
            }
        }
    };
});
```

3. Save the changes.

4. Refresh browser page.

As a result, while adding a new contract. its end date will be 5 days ahead of its start date.

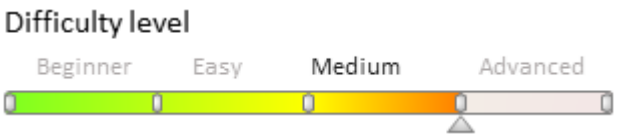
Fig. 1. Case result



NOTE

In order for the date in the [End Date] field to be recalculated automatically when the user changes the [Start Day] field, it is necessary to use the functionality of the computed fields. Please refer to the “**Adding calculated fields**” article for more details.

How to block fields of the edit page



Introduction

During the development of the bpm’online custom functions you may need to block all fields and details on the page when specific condition is met. Mechanism of blocking of the edit page fields can simplify the process without creating a number of business rules.

More information about blocking of the page fields can be found in the “**Blocking edit page fields** ”.

ATTENTION

Blocking mechanism is implemented in the bpm'online version 7.11.1 or higher.

Case description

Block all the fields on the invoice edit page if the invoice is on the [Paid] stage. The [Payment status] field and the [Activities] detail should stay editable.

ATTENTION

If the field has binding for the *enabled* property in the *diff* array element or in the BINDPARAMETER

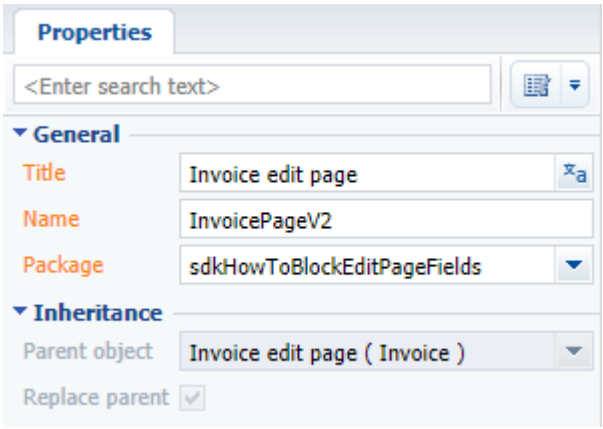
business rule, the mechanism will not block this field.

Case implementation algorithm

1. Create a replacing schema of the invoice edit page

Create a replacing client module and specify the [Invoice edit page] schema as parent (Fig. 1). The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**" article.

Fig. 1. The properties of the [Invoice edit page] schema



2. Add schema source code

Add the source code of implementation of the [Invoice edit page] replacing schema on the [Source code] panel of schema designer. The source code is available below:

```
define("InvoicePageV2", ["InvoiceConfigurationConstants"],
function(InvoiceConfigurationConstants) {
    return {
        entitySchemaName: "Invoice",
        attributes: {
            // Status of the blocking of fields.
            "IsModelItemsEnabled": {
                dataValueType: Terrasoft.DataValueType.BOOLEAN,
                value: true,
                dependencies: [{
                    columns: ["PaymentStatus"],
                    methodName: "setCardLockoutStatus"
                }]
            }
        },
        methods: {
            getDisableExclusionsColumnTags: function() {
                // The [Payment status] field should not be blocked.
                return ["PaymentStatus"];
            },

            getDisableExclusionsDetailSchemaNames: function() {
                // Also, the "Activity" detail is not blocked.
                return ["ActivityDetailV2"];
            },
            setCardLockoutStatus: function() {
                // Get current invoice status.
                var state = this.get("PaymentStatus");
                // If the current account status is "paid", then block the fields.
                if (state.value ===
InvoiceConfigurationConstants.Invoice.PaymentStatus.Paid) {
```

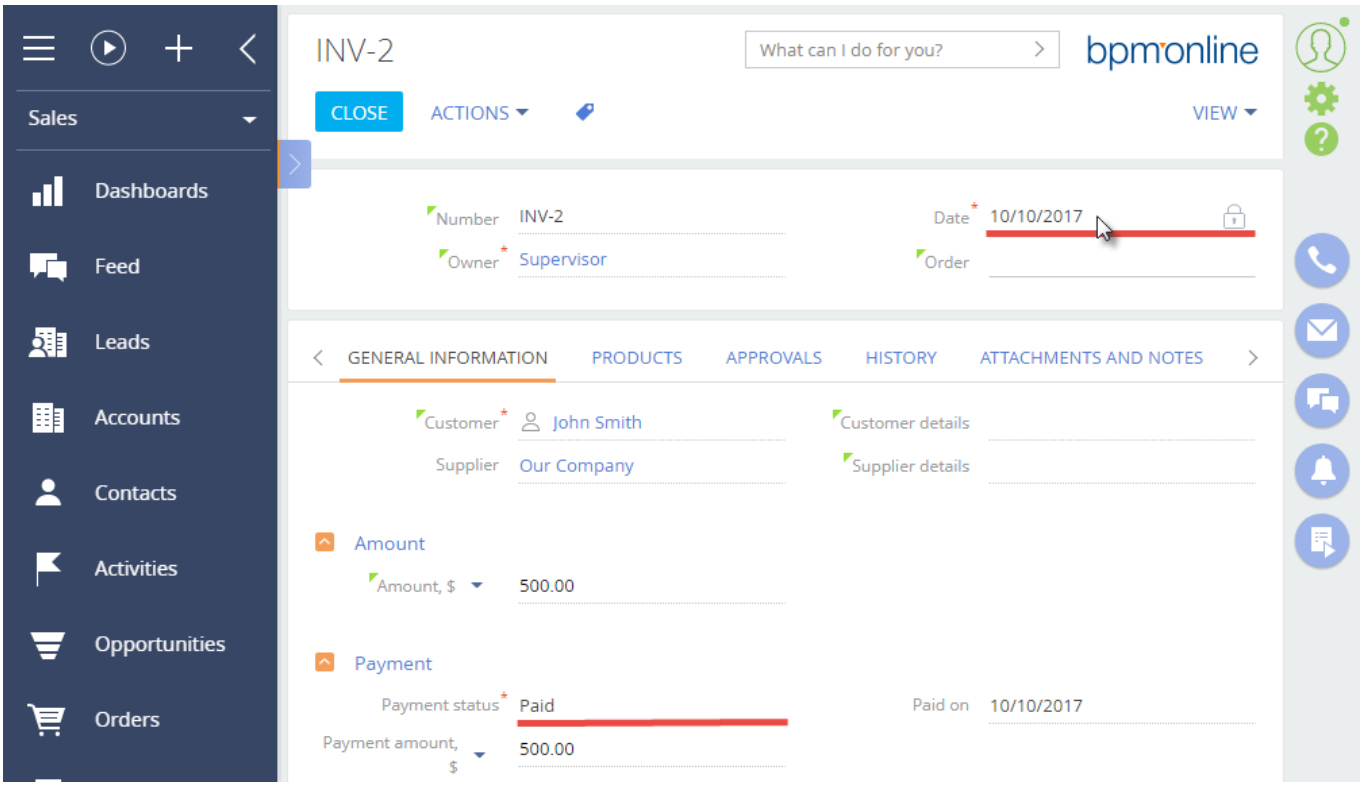
```
                    // Set a property that stores the field lock flag.
                    this.set("IsModelItemsEnabled", false);
                } else {
                    // Otherwise, unlock the fields.
                    this.set("IsModelItemsEnabled", true);
                }
            },
            onEntityInitialized: function() {
                this.callParent(arguments);
                // Set the status of the blocking of fields.
                this.setCardLockoutStatus();
            }
        },
        diff: /**SCHEMA_DIFF*/[
            {
                "operation": "merge",
                "name": "CardContentWrapper",
                "values": {
                    "generator": "DisableControlsGenerator.generatePartial"
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

The *IsModelItemsEnabled* attribute will be defined and methods for blocking and unlocking the field of the invoice edit page will be implemented. The *CardContentWrapper* container of the edit page is used as the container of the blocking mechanism.

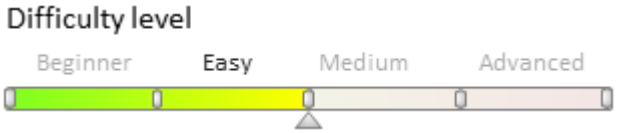
Save the schema after adding the source code. Then clear the browser cache.

As a result, the most of the invoice fields will be blocked after changing the status to the [Paid]. Fields and details specified in the exceptions for blocking will stay unlocked. The fields that have the *enabled* property explicitly set to *true* will stay unlocked

Fig. 2. Case result



Adding details



Overview

Details are the elements of the section record edit page that display supplemental data for a primary section object. The details displayed on the tabs of section edit page in the tabs container.

There are four main types of details:

1. A detail with adding page is a standard bpm’online detail. It can be created manually or added to the section via the detail wizard. More information about detail creation can be found in the “**Adding an edit page detail**” and “**Creating a detail in wizards**” articles.
2. A detail with editable list different from the standard detail. The data can be added, deleted and modified directly in the detail. More information about creation of the detail with editable list can be found in the “**Adding a detail with an editable list**” article.
3. A detail with selection from lookup – data are selected from a lookup displayed in the modal window. More information about detail creation can be found in the “**Creating a detail with selection from lookup**” article.
4. A detail with edit fields – data are filled in and edited in the detail data fields. More information about detail creation can be found in the “**Creating a custom detail with fields**” and “**Advanced settings of a custom detail with fields**” article.

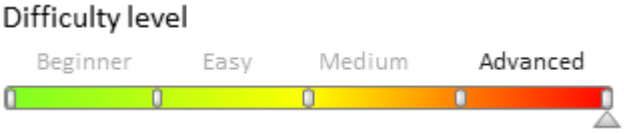
More information about the details of each type is described in the “**Details**” article of the “**Application interface and structure**” section.

> ⚠ ATTENTION
>
> Main schemas, classes, methods and properties of detail functions are described in the “**Details**” article of the “**Controls**” section.

Contents

- **Adding an edit page detail**
- **Adding a detail with an editable list**
- **Creating a detail with selection from lookup**
- **Adding multiple records to a detail**
- **Creating a custom detail with fields**
- **Advanced settings of a custom detail with fields**
- **Creating a detail in wizards**
- **Adding the [Attachments] detail**
- **Displaying additional columns on the [Attachments] tab**
- **How to hide menu commands of the detail with list**

Adding an edit page detail



General provisions

Details are special elements of section edit pages, which display additional data that is not stored in the fields of the section primary object. The details are displayed on the edit page tabs in the tab container.

There are four basic types of details:

- details with a record edit page;
- details with edit fields;
- details with editable list;
- details with lookup selection.

For more information on details, please see the "**Details**" article in the "**Application interface and structure**" section.

An edit page detail is a standard bpm'online detail that can be added to sections using the Detail WIzard. Below is an example of adding an edit page detail without the use of the Detail Wizard.

General procedure for adding an edit page detail to an existing section

To add a custom edit page detail:

1. Create a detail object schema.
2. Create a detail schema.
3. Create a detail edit page schema.
4. Set up the detail in a replacing section edit page schema.
5. Register links between object, detail and detail page schemas, using special SQL queries to system tables.
6. Set up the detail fields.

> ⚠ NOTE!
>
> To bind custom schemas, make changes to the SysModuleEdit, SysModulentity and SysDetail system tables in the bpm'online database, using SQL queries.
>
> Be extremely careful when composing and executing SQL queries to the database. Executing an invalid query may damage existing data and disrupt system operation.

Case description

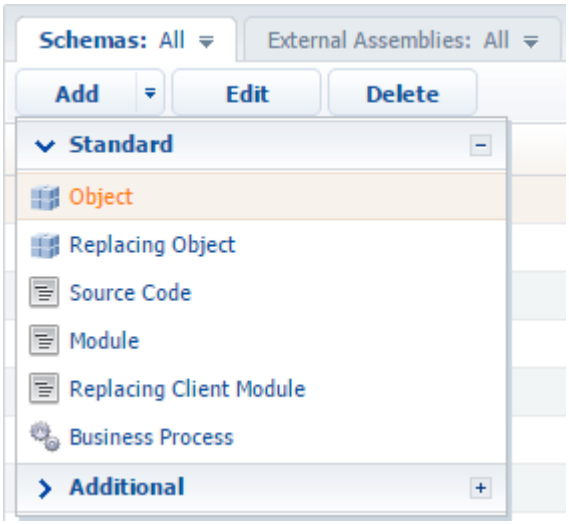
Create a custom [Couriers] detail in the [Orders] section. The detail must contain the list of couriers for the current order.

Case implementation algorithm

1. Create detail object schema

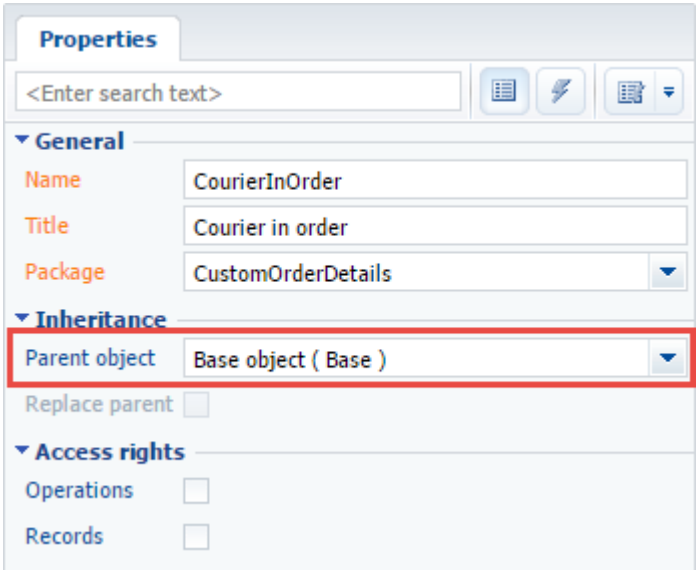
In the settings mode, go to the [Configuration] section, open the [Schemas] tab and execute the [Add] > [Object] menu command (Fig. 1).

Fig. 1. Adding a detail object schema



In the opened **Object Designer**, fill out the properties of the detail object schema, as shown on Fig. 2.

Fig. 2. Setup of detail object schema properties



Add a lookup column [Order], which will connect the detail object with the section object, and a lookup [Contact] column where the courier's contact will be stored. Both columns must be required. Column properties setup is shown on Fig. 3 and Fig. 4.

Fig. 3. Specifying the properties of the [Order] column

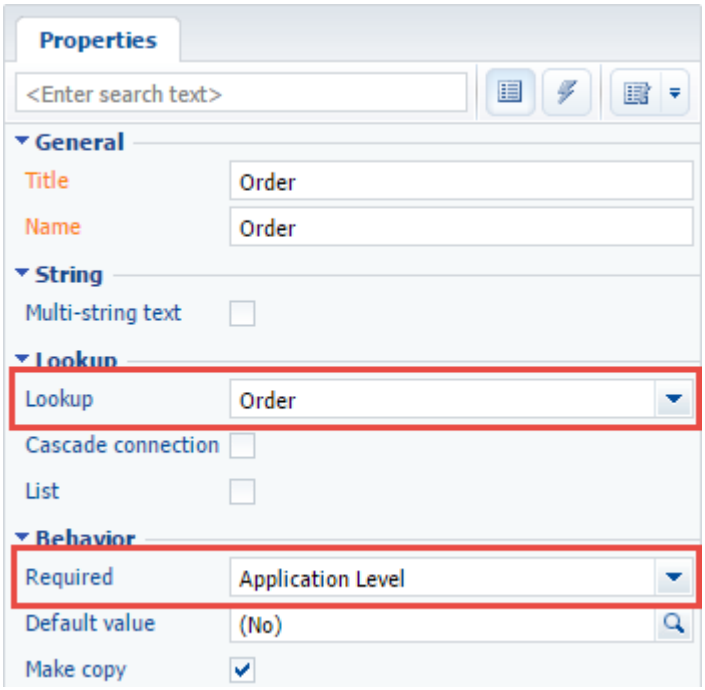
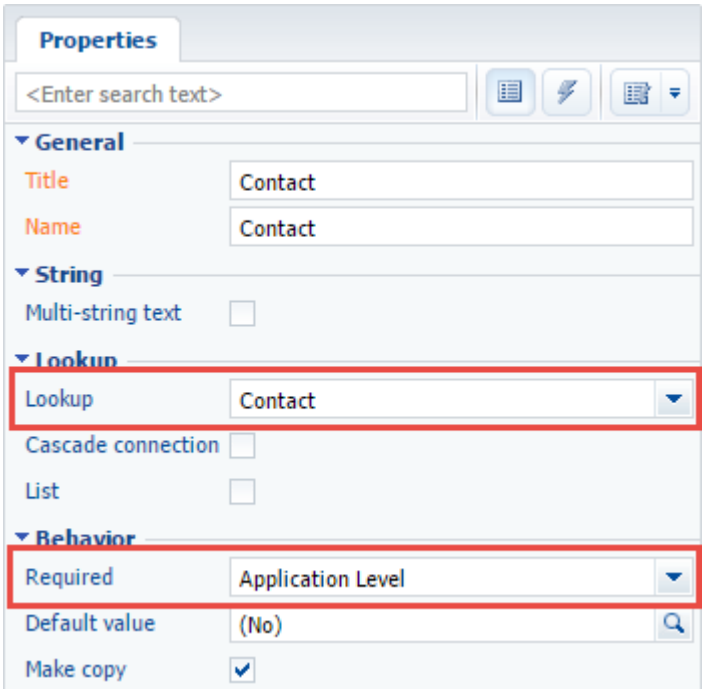


Fig. 4. Specifying the properties of the [Contact] column



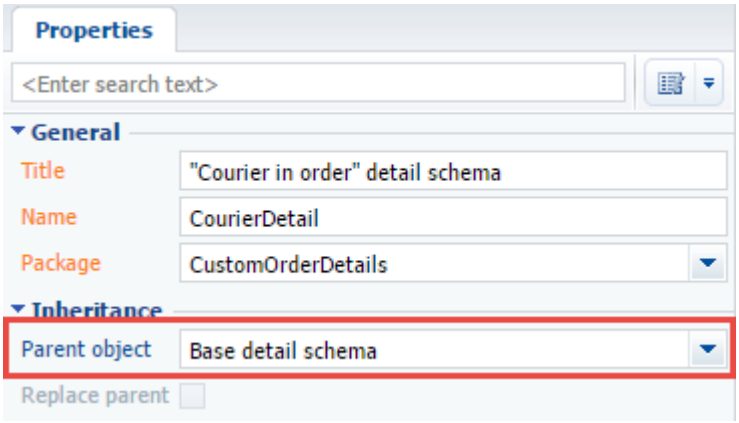
Object schema must be saved and published.

2. Create detail schema

In the settings mode, go to the [Configuration] section, open the [Schemas] tab and execute the [Add] > [Module] menu command (Fig. 1).

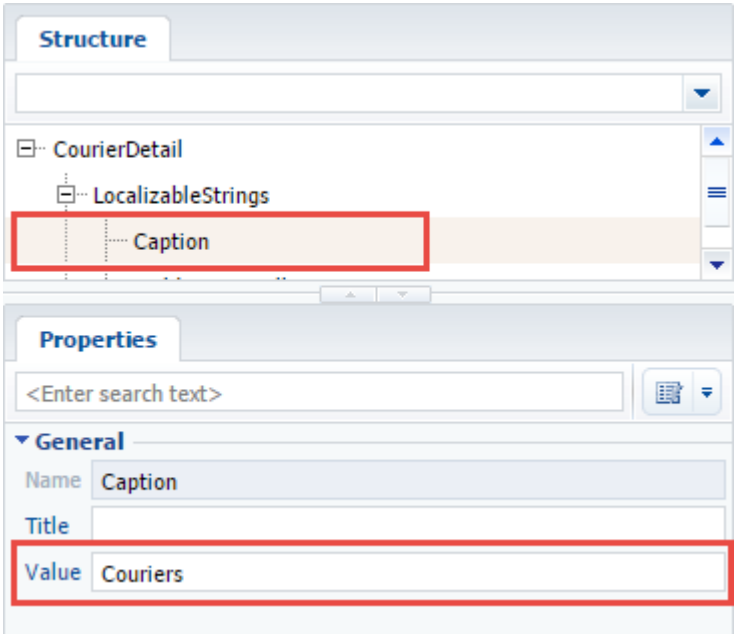
The new module must inherit the functionality of the base detail schema with list *BaseGridDetailV2*, which is available in the *NUI* package. To do this, specify this schema as the parent for the created detail schema (Fig. 5). The rest of the properties must be set up as shown on Fig. 5. In the [Package] property, the system will specify the current package name.

Fig. 5. Detail schema properties



Set the *Couriers* value for the [Caption] localizable string of the detail schema (Fig. 6). The [Caption] string contains detail title, shown on the edit page.

Fig. 6. Setting the value of the [Caption] string



Below is the detail schema source code, which must be added to the [Source code] section of the client module designer. The code defines the schema name and its connection to the object schema.

```
define("CourierDetail", [], function() {
    return {
        // Detail object schema name.
        entitySchemaName: "CourierInOrder",
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        methods: {},
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    };
});
```

Save the detail schema to apply the changes.

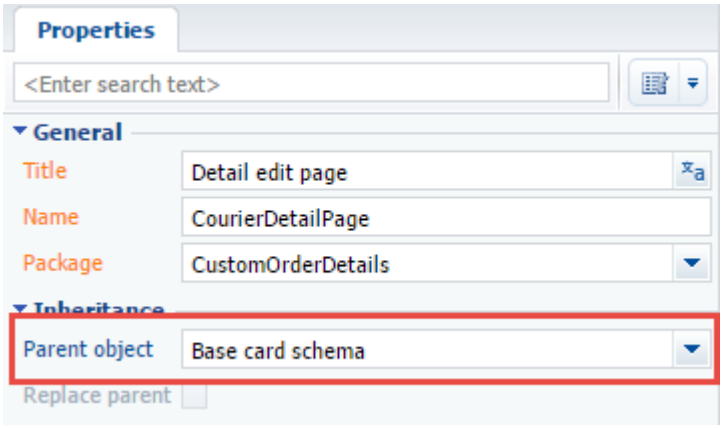
3. Create detail edit page schema

In the settings mode, go to the [Configuration] section, open the [Schemas] tab and execute the [Add] > [Module] menu command (Fig. 1).

The created detail edit page schema must inherit the functionality of base page schema *BasePageV2*, which is available in the *NUI* package. To do this, specify this schema as parent (Fig. 7). The rest of the properties must be set

up as shown on Fig. 7. In the [Package] property, the system will specify the current package name.

Fig. 7. Detail edit page schema properties



To set up fields displayed on the detail edit page, add the following code to the [Source code] section of the client module designer. In this code, in the diff array, configuration objects of metadata for adding, setting location and binding the order and courier fields are specified.

```
define("CourierDetailPage", [], function() {
    return {
        // Detail object schema name.
        entitySchemaName: "CourierInOrder",
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        // Array with modifications.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding the [Order] field.
            {
                "operation": "insert",
                //HField name.
                "name": "Order",
                "values": {
                    // Field position setup on the edit page.
                    "layout": {
                        "colSpan": 12,
                        "rowSpan": 1,
                        "column": 0,
                        "row": 0,
                        "layoutName": "Header"
                    },
                    // Binding to the [Order] column of the object schema.
                    "bindTo": "Order"
                },
                "parentName": "Header",
                "propertyName": "items",
                "index": 0
            },
            // Metadata for adding the [Contact] field.
            {
                "operation": "insert",
                //Field name.
                "name": "Contact",
                "values": {
                    // Field position setup on the edit page.
                    "layout": {
                        "colSpan": 12,
                        "rowSpan": 1,
                        "column": 12,
                        "row": 0,
```

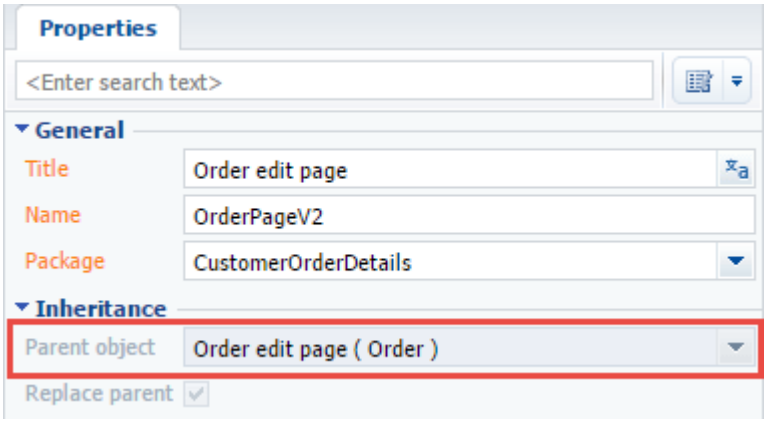
```
                        "layoutName": "Header"
                    },
                    // Binding to the [Contact] column of the object schema.
                    "bindTo": "Contact"
                },
                "parentName": "Header",
                "propertyName": "items",
                "index": 1
            }
        ]/**SCHEMA_DIFF*/,
        methods: {},
        rules: {}
    };
});
```

Save the detail edit page schema to apply the changes.

4. Set up detail in replacing section edit page schema

To display the detail on the order edit page, first create a replacing client module, and specify the order edit page *OrderPageV2* (located in the Order package) as the parent (Fig. 8). For more information on creating replacing schemas, please see the "**Creating a custom client module schema**" article.

Fig. 8. Order edit page replacing schema properties



To display the [Couriers] detail on the [Delivery] tab of the order edit page, add the following source code. In the *details* section, a new *CourierDetail* model will be defined, and its location on the edit page will be specified in the modification section of the *diff* array.

```
define("OrderPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Order",
        // List of edit page details being added.
        details: /**SCHEMA_DETAILS*/{
            // [Couriers] detail setup.
            "CourierDetail": {
                // Detail schema name.
                "schemaName": "CourierDetail",
                // Detail object schema name.
                "entitySchemaName": "CourierInOrder",
                // Filtering contacts for current order only.
                "filter": {
                    // Detail object schema column.
                    "detailColumn": "Order",
                    // Section object schema column.
                    "masterColumn": "Id"
                }
            }
```

```
        }/**SCHEMA_DETAILS*/,
        // Array of modifications.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding the [Couriers] detail.
            {
                "operation": "insert",
                // Detail name.
                "name": "CourierDetail",
                "values": {
                    "itemType": 2,
                    "markerValue": "added-detail"
                },
                // Parent container ([Delivery] tab).
                "parentName": "OrderDeliveryTab",
                // Container where detail is located.
                "propertyName": "items",
                // Index in the list of added elements.
                "index": 3
            }
        ]/**SCHEMA_DIFF*/,
        methods: {},
        rules: {}
    };
});
```

To apply the changes, the replacing page schema must be saved.

After this, the detail will be displayed on the edit page of the [Orders] section. New records cannot be added to the detail until the connections between the detail schemas are registered.

5. Register connections between the schemas using SQL queries to system tables

To register connection between a detail object schema and detail schema, execute the following SQL query.

```
DECLARE
-- Schema name of the created detail.
@DetailSchemaName NCHAR(100) = 'CourierDetail',
-- Detail title.
@DetailCaption NCHAR(100) = 'Couriers',
--Schema name of the object, to which the detail is bound.
@EntitySchemaName NCHAR(100) = 'CourierInOrder'

INSERT INTO SysDetail(
    ProcessListeners,
    Caption,
    DetailSchemaUId,
    EntitySchemaUId
)
VALUES (
    0,
    @DetailCaption,
    (SELECT TOP 1 UId
    FROM SysSchema
    WHERE name = @DetailSchemaName),
    (SELECT TOP 1 UId
    FROM SysSchema
    WHERE name = @EntitySchemaName)
)
```

To register connection between the detail object schema and edit page schema, execute the following SQL query.

```
DECLARE
-- Schema name of the detail page
```

```
@CardSchemaName NCHAR(100) = 'CourierDetailPage',
-- Object schema.
@EntitySchemaName NCHAR(100) = 'CourierInOrder',
-- Detail page caption.
@PageCaption NCHAR(100) = 'Page of detail "Courier In Order"',
-- Empty string.
@Blank NCHAR(100) = ''

INSERT INTO SysModuleEntity(
    ProcessListeners,
    SysEntitySchemaUId
)
VALUES(
    0,
    (SELECT TOP 1 UId
    FROM SysSchema
    WHERE Name = @EntitySchemaName
    )
)

INSERT INTO SysModuleEdit(
    SysModuleEntityId,
    UseModuleDetails,
    Position,
    HelpContextId,
    ProcessListeners,
    CardSchemaUId,
    ActionKindCaption,
    ActionKindName,
    PageCaption
)
VALUES (
    (SELECT TOP 1 Id
    FROM SysModuleEntity
    WHERE SysEntitySchemaUId = (
        SELECT TOP 1 UId
        FROM SysSchema
        WHERE Name = @EntitySchemaName
        )
    ),
    1,
    0,
    @Blank,
    0,
    (SELECT TOP 1 UId
     FROM SysSchema
     WHERE name = @CardSchemaName
    ),
    @Blank,
    @Blank,
    @PageCaption
)
```

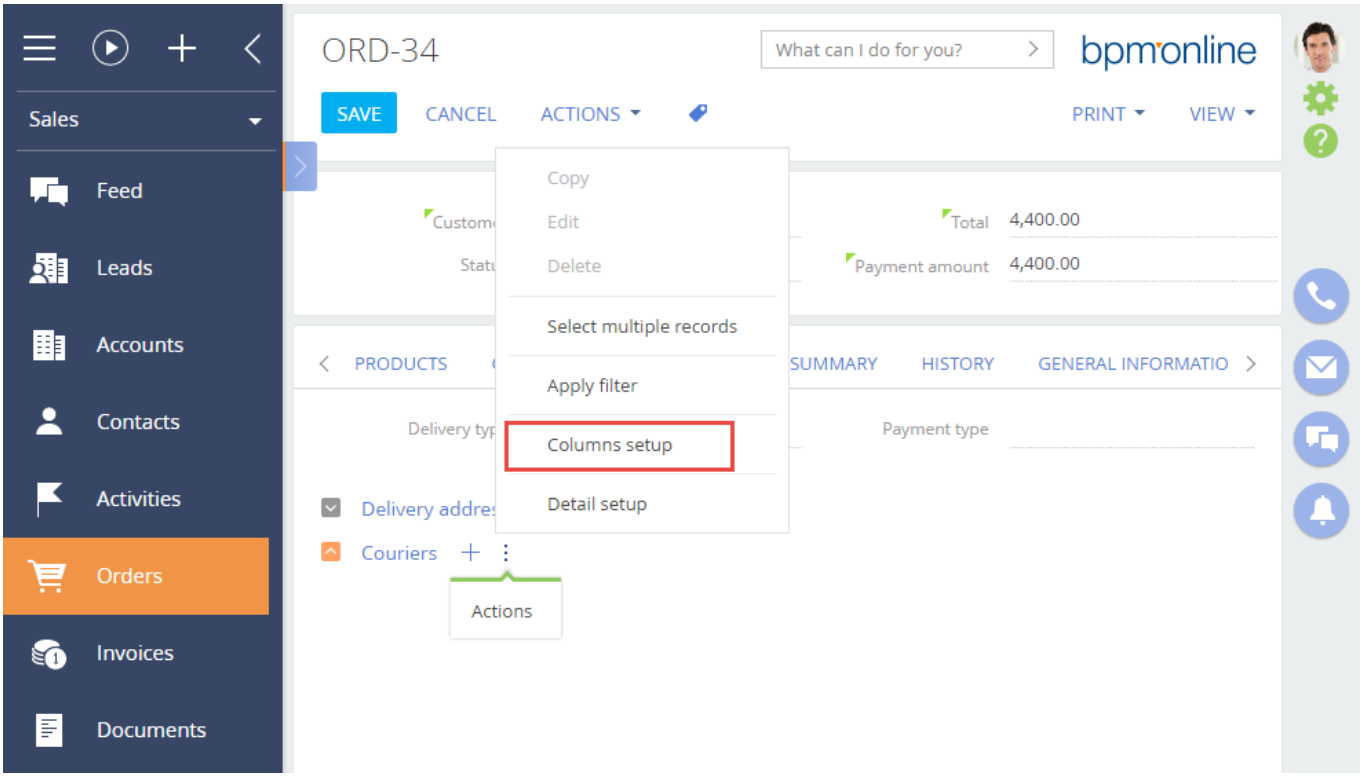
> **Attention**
>
> To apply these changes on the application level, restart the application site in IIS, or compile the application in the [Configuration] section.

6. Set up detail list columns

At this stage, the detail is completely operational, however contacts are not displayed on the detail, because the

detail list does not have displayed columns specified for it. Go to the detail menu and set up columns to display (Fig. 9).

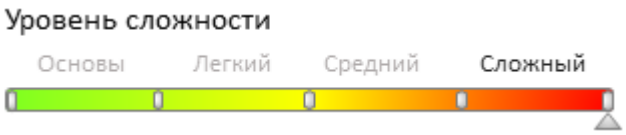
Fig. 9. Detail actions menu



See also:

- **Creating a detail in wizards**
- **Adding a detail with an editable list**
- **Creating a detail with selection from lookup**
- **Creating a custom detail with fields**

Adding a detail with an editable list



Introduction

Details are elements of section edit pages. Details display records bound to the current section record by a lookup field. These can be, for example, records from other sections, whose primary objects contain lookup columns linked to the primary object of the current section. Details are displayed on tabs of section edit pages.

More information about details is available in the “**Details**” article of the “**Application interface and structure**” section.

> ⚠ ATTENTION
>
> A detail with an editable list is not a standard bpm'online detail and can not be added to sections using the Detail Wizard.

To add a custom detail with editable list to an existing section:

1. Create a detail object schema.
2. Create ad configure the detail schema.
3. Set up the detail in the section edit page replacing schema.
4. Set up detail fields.

Case description

Create a custom [Courier services] detail in the [Orders] section. The detail should display a list of courier services for the current order.

Source code

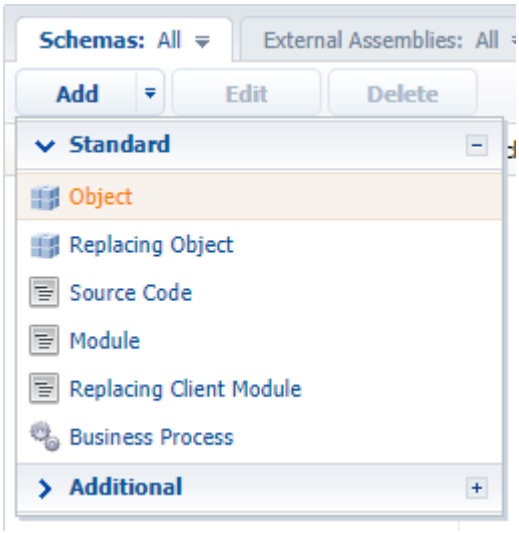
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Creating a detail object schema.

Perform the [Add] – [Object] action on the [Schemas] tab of the [Configuration] section.

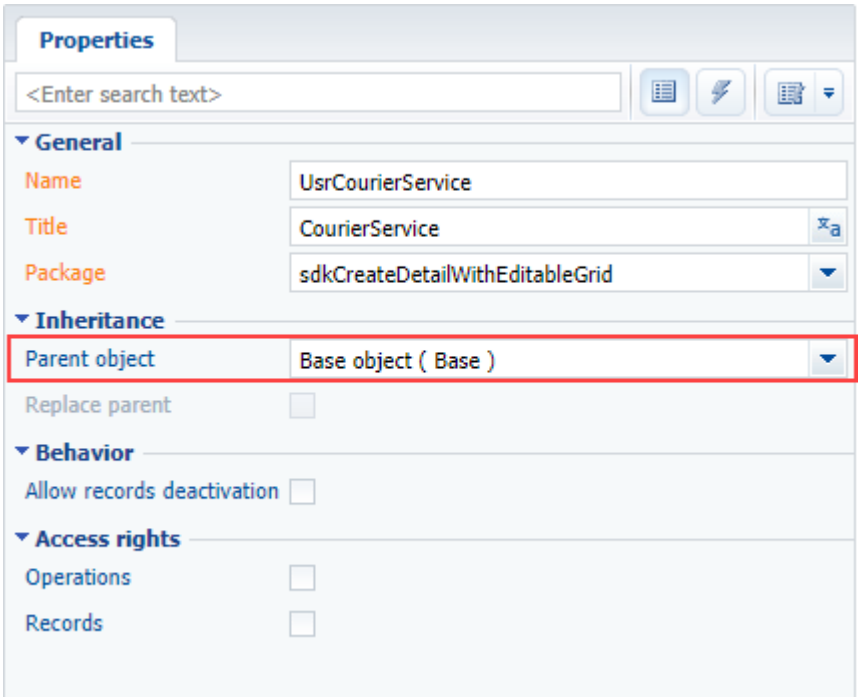
Fig. 1. Adding a detail object schema



For the created object schema (Fig. 2) specify following:

- [Name] – "UsrCourierService”
- [Title] – "CourierService"
- [Parent object] – [Base object].

Fig. 2. Setup of detail object schema properties



Add the [Order] lookup column to the object schema establishing connection with the [Orders] section and the [Account] lookup column containing the account carrying out delivery for this order. Mark both columns as “required” to avoid adding empty records. Column setup is shown in fig. 3 and fig. 4.

Fig. 3. The [Order] column properties setup

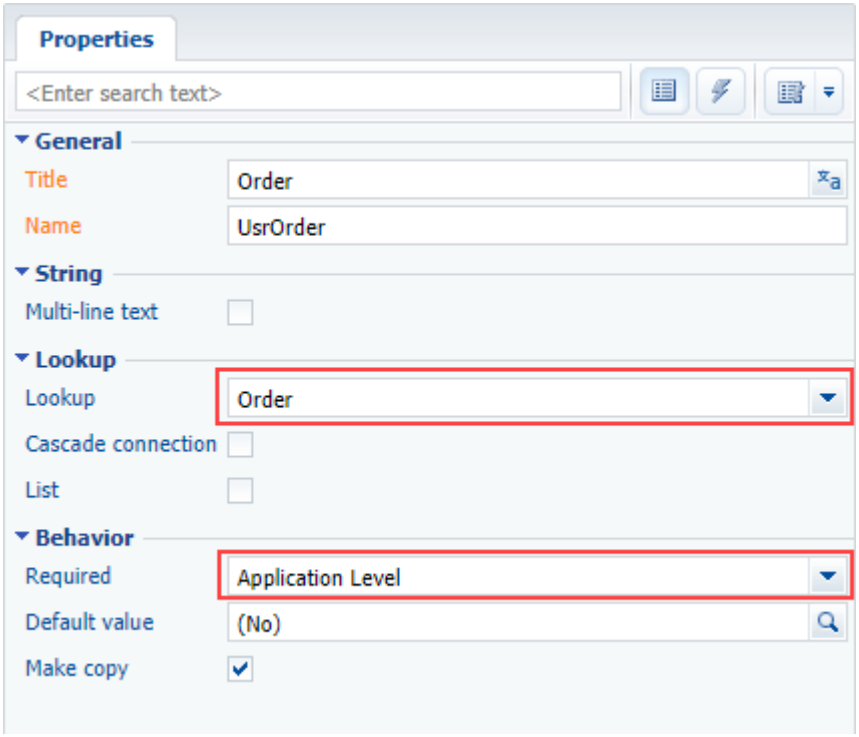
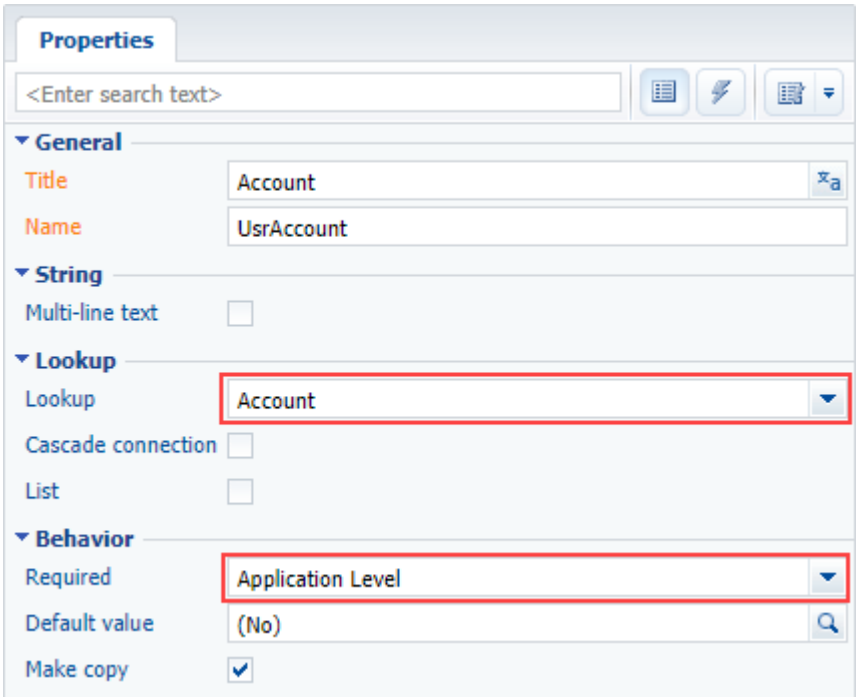


Fig. 4. The [Account] column properties setup



Save and publish the object schema.

2. Creating a detail schema.

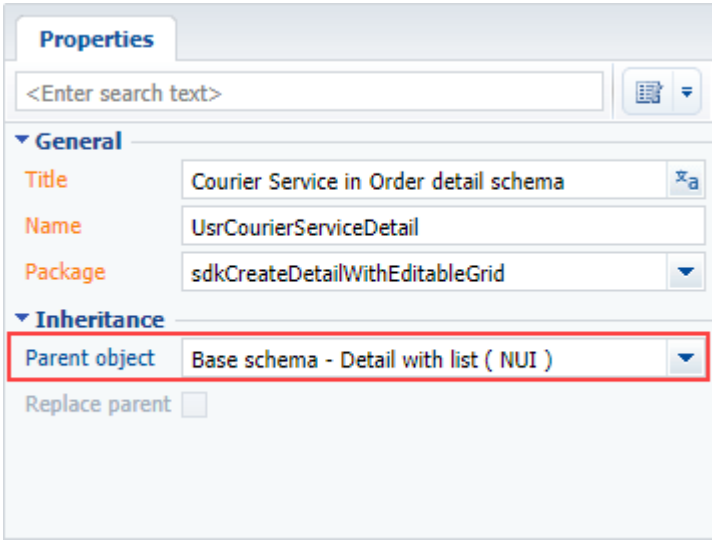
Run the [Add] – [Module] menu command on the [Schemas] tab of the [Configuration] section (fig. 1).

The created module should inherit the *BaseGridDetailV2* base detail list schema functions (specify it as the parent schema) defined in the *NUI* package.

Specify other properties (Fig. 5):

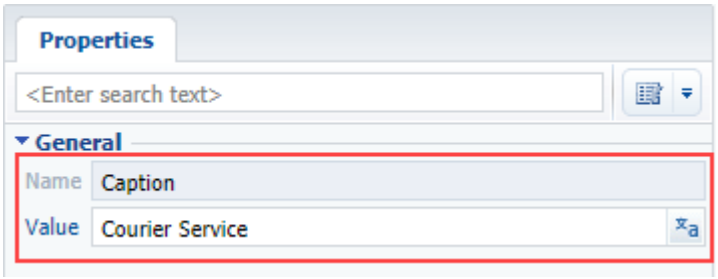
- [Name] – "UsrCourierServiceDetail"
- [Title] – "Courier Service in Order detail schema".

Fig. 5. Detail schema properties



Set the [Courier Service] value for the [Caption] localizable string of the detail list schema (fig. 6). The [Caption] localizable string stores the detail caption, displayed on the edit page.

Fig. 6. Setting the [Caption] localizable string value



To make the list of the detail editable, make the following changes to the detail schema:

1. Add dependencies from the *ConfigurationGrid*, *ConfigurationGridGenerator*, *ConfigurationGridUtilities* modules.
2. Connect the *ConfigurationGridUtilities* mixin.
3. Set the *IsEditable* attribute value to *true*.
4. In the **diff array of modifications**, add the configuration object in which the properties are set and handler methods for detail list events are bound.

Below is the detail schema source code with comments:

```
// Defining schema and setting its dependencies from other modules.
define("UsrCourierServiceDetail", ["ConfigurationGrid", "ConfigurationGridGenerator",
    "ConfigurationGridUtilities"], function() {
    return {
        // Detail object schema name.
        entitySchemaName: "UsrCourierService",
        // Schema attribute list.
        attributes: {
            // Determines whether the editing is enabled.
            "IsEditable": {
                // Data type – logic.
                dataValueType: Terrasoft.DataValueType.BOOLEAN,
                // Attribute type – virtual column of the view model.
                type: Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                // Set value.
                value: true
            }
        },
        // Used mixins.
        mixins: {
            ConfigurationGridUtilities: "Terrasoft.ConfigurationGridUtilities"
        },
        // Array with view model modifications.
        diff: /**SCHEMA_DIFF*/[
            {
                // Operation type – merging.
                "operation": "merge",
                // Name of the schema element, with which the action is performed.
                "name": "DataGrid",
                // Object, whose properties will be joined with the schema element
properties.
                "values": {
                    // Class name
                    "className": "Terrasoft.ConfigurationGrid",
                    // View generator must generate only part of view.
                    "generator": "ConfigurationGridGenerator.generatePartial",
                    // Binding the edit elements configuration obtaining event
                    // of the active page to handler method.
                    "generateControlsConfig": {"bindTo":
"generateActiveRowControlsConfig"},
                    // Binding the active record changing event to handler method.
                    "changeRow": {"bindTo": "changeRow"},
```

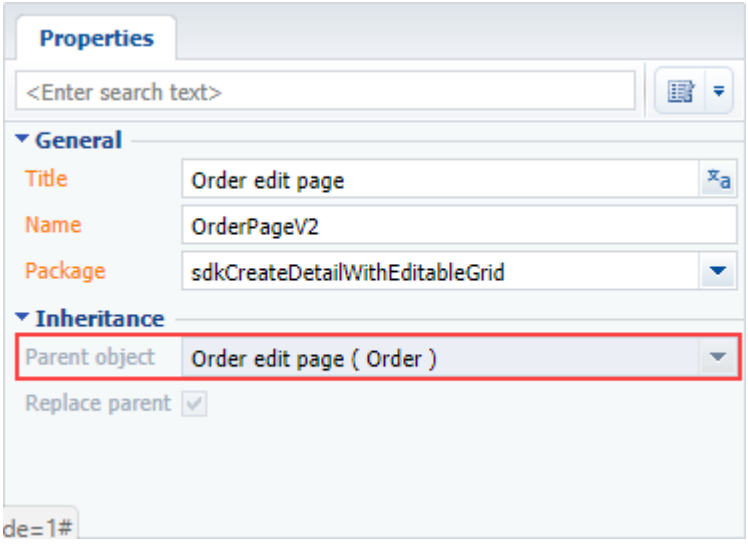
```
                    // Binding the record selection cancellation event to handler
method.
                    "unSelectRow": {"bindTo": "unSelectRow"},
                    // Binding of the list click event to handler method.
                    "onGridClick": {"bindTo": "onGridClick"},
                    // Actions performed with active record.
                    "activeRowActions": [
                        // [Save] action setup.
                        {
                            // Class name of the control element, with which the
action is connected.
                            "className": "Terrasoft.Button",
                            // Display style — transparent button.
                            "style":
this.Terrasoft.controls.ButtonEnums.style.TRANSPARENT,
                            // Tag.
                            "tag": "save",
                            // Marker value.
                            "markerValue": "save",
                            // Binding button image.
                            "imageConfig": {"bindTo": "Resources.Images.SaveIcon"}
                        },
                        // [Cancel] action setup.
                        {
                            "className": "Terrasoft.Button",
                            "style":
this.Terrasoft.controls.ButtonEnums.style.TRANSPARENT,
                            "tag": "cancel",
                            "markerValue": "cancel",
                            "imageConfig": {"bindTo": "Resources.Images.CancelIcon"}
                        },
                        // [Delete] action setup.
                        {
                            "className": "Terrasoft.Button",
                            "style":
this.Terrasoft.controls.ButtonEnums.style.TRANSPARENT,
                            "tag": "remove",
                            "markerValue": "remove",
                            "imageConfig": {"bindTo": "Resources.Images.RemoveIcon"}
                        }
                    ],
                    // Binding to method that initializes subscription to events
                    // of clicking buttons in the active row.
                    "initActiveRowKeyMap": {"bindTo": "initActiveRowKeyMap"},
                    // Binding the active record action completion event to handler
method.
                    "activeRowAction": {"bindTo": "onActiveRowAction"},
                    // Identifies whether multiple records can be selected.
                    "multiSelect": false
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

Save the created detail list schema to apply the changes.

3. Setting up detail in the section edit page replacing schema.

To display the detail on the order edit page, create a client replacing module and indicate the [Order edit page] as the parent object, *OrderPageV2*) (fig. 7). Creating a replacing page is covered in the "**Creating a custom client module schema**" article.

Fig. 7. Properties of the order edit page replacing schema



To display the [Courier Service] detail on the [Delivery] tab of the order edit page, add the following source code. It defines a new *UsrCourierServiceDetail* detail in the *details* section and its location on the order edit page is specified in the *diff* modification array section.

```
define("OrderPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Order",
        // List of edit page details being added.
        details: /**SCHEMA_DETAILS*/{
            // [Courier services] detail setup.
            "UsrCourierServiceDetail": {
                // Detail schema name.
                "schemaName": "UsrCourierServiceDetail",
                // Detail object schema name.
                "entitySchemaName": "UsrCourierService",
                // Filtering displayed contacts for current order only.
                "filter": {
                    // Detail object schema column.
                    "detailColumn": "UsrOrder",
                    // Section object schema column.
                    "masterColumn": "Id"
                }
            }
        }/**SCHEMA_DETAILS*/,
        // Array with modifications.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding the [Courier services] detail.
            {
                "operation": "insert",
                // Detail name.
                "name": "UsrCourierServiceDetail",
                "values": {
                    "itemType": Terrasoft.core.enums.ViewItemType.DETAIL,
                    "markerValue": "added-detail"
                },
                // Containers, where the detail is located.
                // Detail is placed on the [Delivery] tab.
                "parentName": "OrderDeliveryTab",
                "propertyName": "items",
                // Index in the list of added elements.
                "index": 3
```

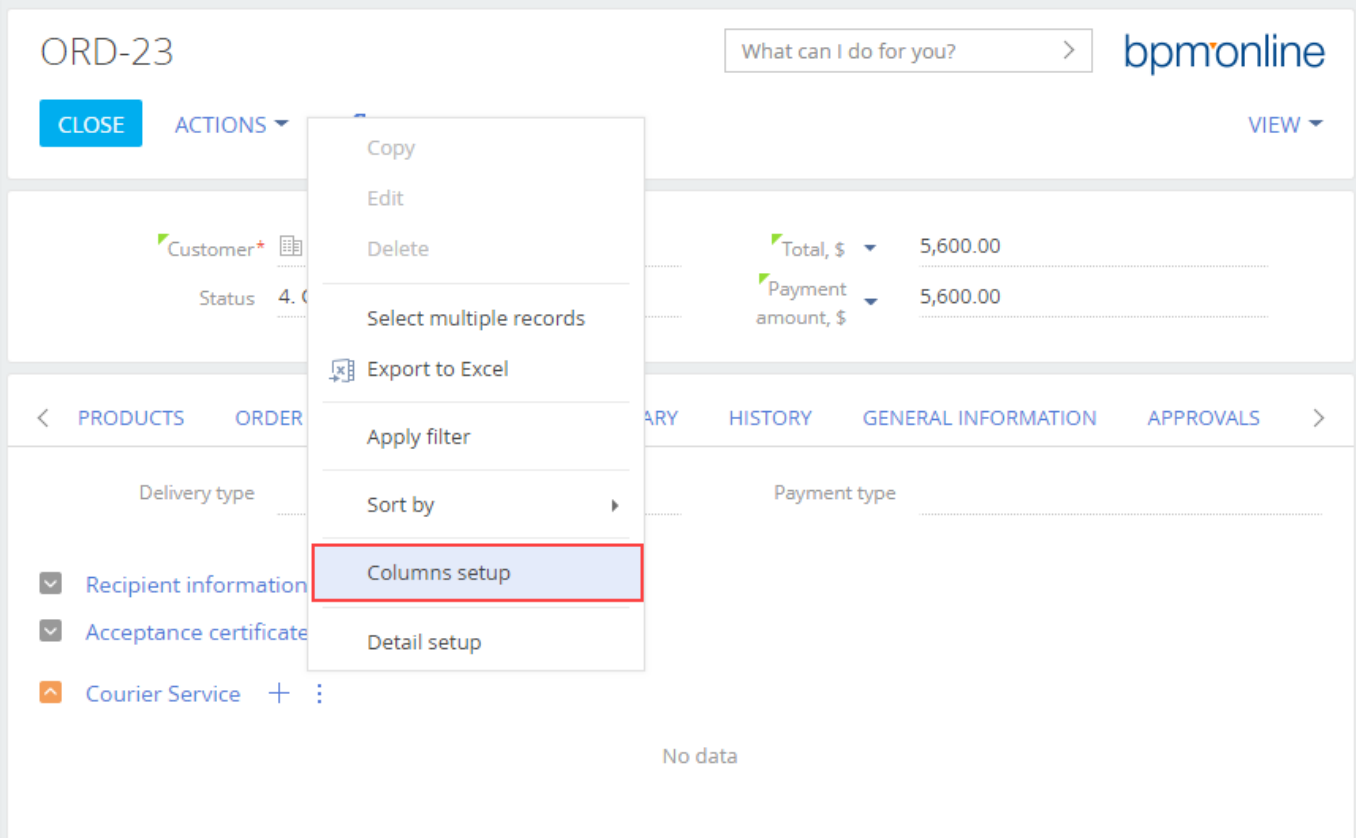
```
                }
            ]/**SCHEMA_DIFF*/
        };
});
```

Save a replacing schema of the edit page

4. Setup of detail list columns.

At this stage, the detail is completely operational, however accounts are not displayed on the detail, because the detail list does not have displayed columns specified for it. Go to the detail menu and set up the columns to be displayed (Fig. 8)

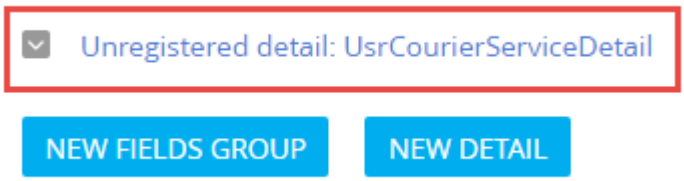
Fig. 8. Detail action menu



Detail wizard, Section wizard and details with editable lists

A detail with an editable list is not a standard bpm'online detail and can not be added to sections using the Detail Wizard. If you try perform the [Detail setup] action (Fig. 8), bpm’online will display the “detail is not registered” message. The Section wizard displays a similar message (Fig. 9).

Fig. 9. Unregistered detail in the Section wizard



Usually details with editable lists do not require registration. However, if you still need to register such a detail for some reason, use the following SQL script:

```
DECLARE
```

```
    -- Name of the created pop-up summary view schema.
    @ClientUnitSchemaName NVARCHAR(100) = 'UsrCourierServiceDetail',
    -- Name of the object schema, to which the pop-up summary is bound.
    @EntitySchemaName NVARCHAR(100) = 'UsrCourierService',
    -- Detail name.
    @DetailCaption NVARCHAR(100) = 'Courier service'

INSERT INTO SysDetail(Caption, DetailSchemaUId, EntitySchemaUId)
VALUES(@DetailCaption,
     (SELECT TOP 1 UId
      from SysSchema
      WHERE Name = @ClientUnitSchemaName),
      (SELECT TOP 1 UId
      from SysSchema
      WHERE Name = @EntitySchemaName))
```

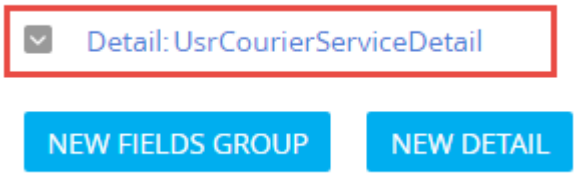
ATTENTION

To register a custom detail, make changes to the *SysDetails* table in the bpm'online database using the SQL query.

Pay high attention to creating and executing the SQL query. Executing an incorrect SQL query can damage the existing data and disrupt the system.

The detail becomes available in the corresponding lookup and displayed as registered in the section wizard after you update the page. After this you can use the Section wizard to add this detail to other tabs of the [Orders] edit page.

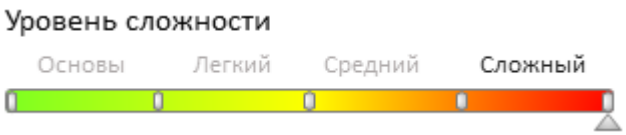
Fig. 9. Registered detail is displayed in the Section wizard



See also

- **Creating a detail in wizards**
- **Adding a detail with an editable list**
- **Creating a detail with selection from lookup**
- **Creating a custom detail with fields**

Creating a detail with selection from lookup



Introduction

Details are elements of section edit pages. Details display records bound to the current section record by a lookup field. These can be, for example, records from other sections, whose primary objects contain lookup columns linked to the primary object of the current section. Details are displayed on tabs of section edit pages.

More information about details is available in the "**Details**" article of the "**Application interface and structure**" section.

Since a detail with selection from lookup is not a standard bpm'online detail, it is not enough to use section wizard to add it to the section. Below we will describe a way of adding such a detail to a section edit page.

To add a custom detail with selection from lookup to an existing section:

1. Create a detail object schema.
2. Create a detail via detail wizard and add it to the section.
3. Configure the detail schema.
4. Set up detail fields.

Case description

Create the [Acceptance certificates] custom detail on the [Delivery] tab of the [Orders] section. The detail should display the list of documents – acceptance certificates for the current order.

Source code

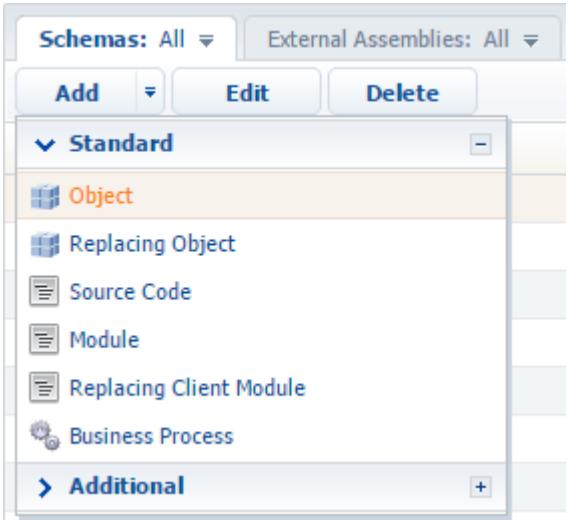
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Creating a detail object schema

Run the [Add] – [Object] menu command on the [Schemas] tab of the [Configuration] section (Fig. 1).

Fig. 1. Adding a detail object schema



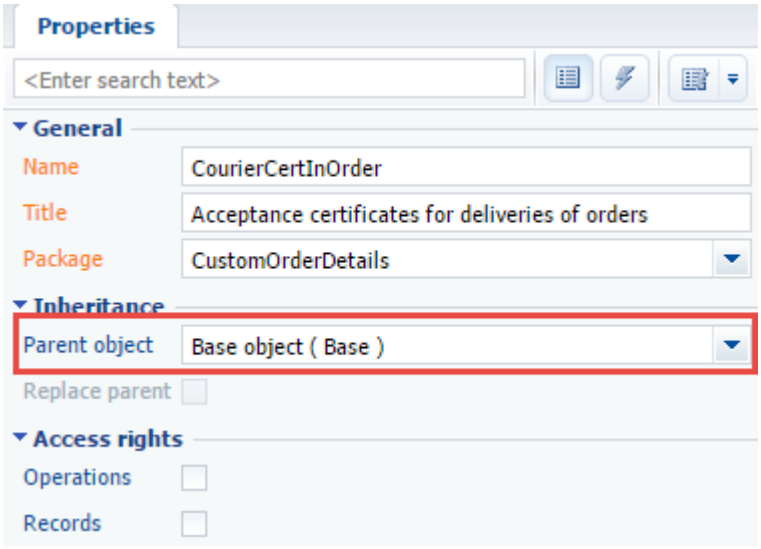
Select the "Base object" as the parent object (Fig. 2).

Object properties:

- [Name] – "UsrCourierCertInOrder"
- [Title] – "Acceptance certificates for deliveries of orders"

Find more information about object schema property setup in object designer in the "**Workspace of the Object Designer**" article.

Fig. 2. Setup of detail object schema properties



Add the [Order] lookup column to the object schema establishing connection with the [Orders] section and the [Document] lookup column containing the acceptance certificate. Mark both columns as “required” to avoid adding empty records. Column setup is shown in fig. 3 and fig. 4.

Fig. 3. The [Order] column property setup

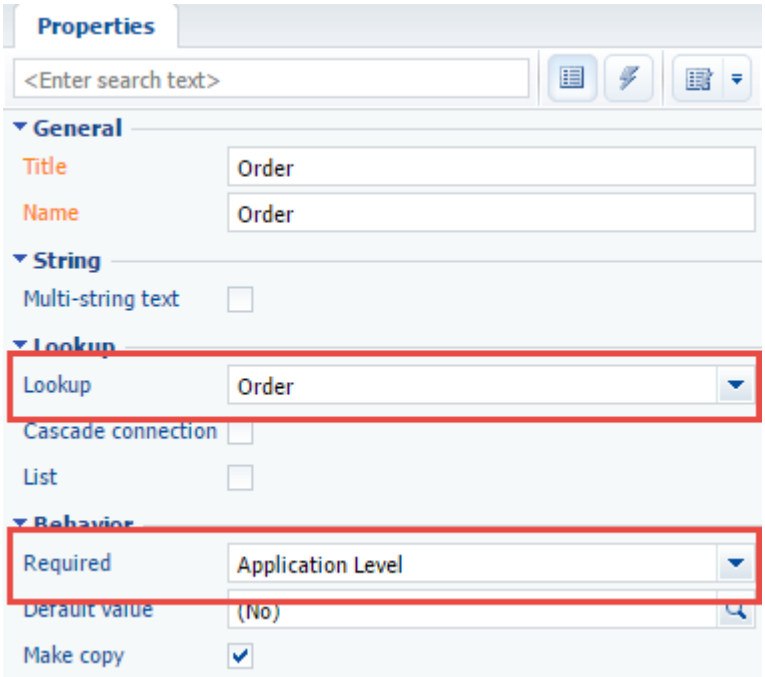
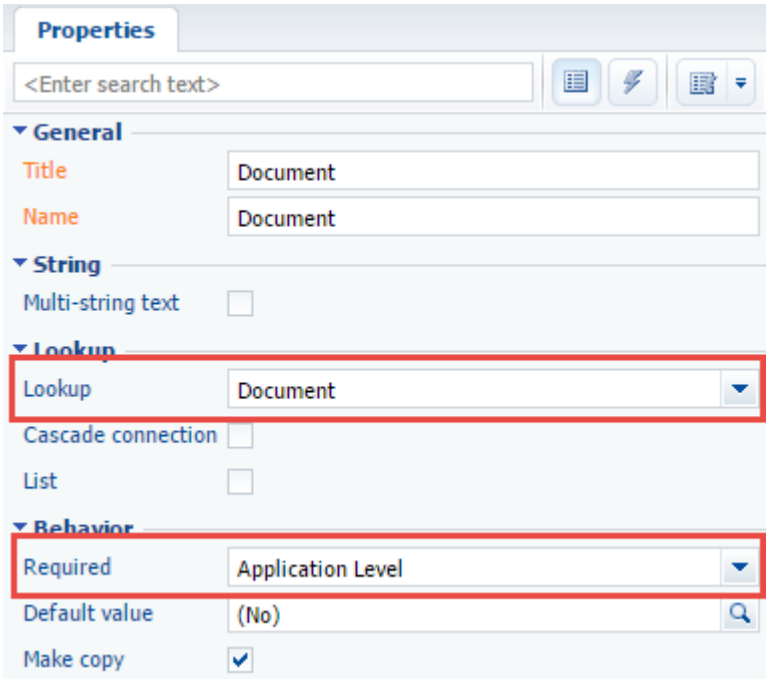


Fig. 4. The [Document] column property setup



Save and publish the object schema.

2. Creating a detail and adding it to the section

Via detail wizard create the [Acceptance certificates] detail (Fig. 5). Add the detail to the [Delivery] tab of the [Orders] section page via section wizard (Fig. 6).

Fig. 5. Creating the detail

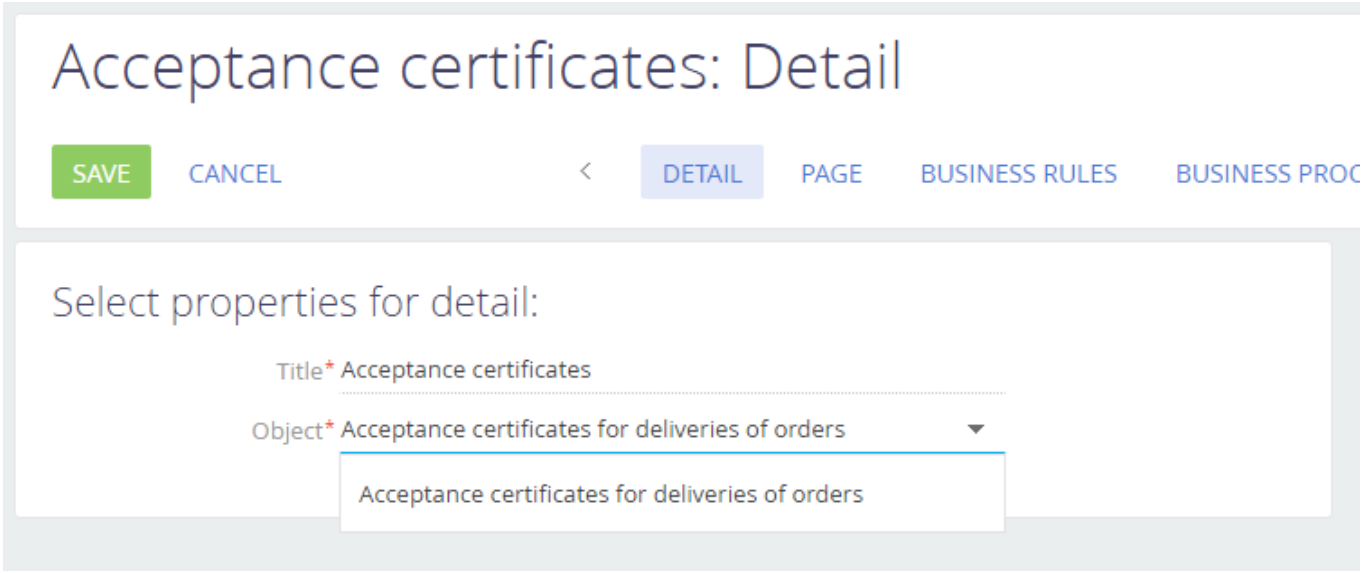
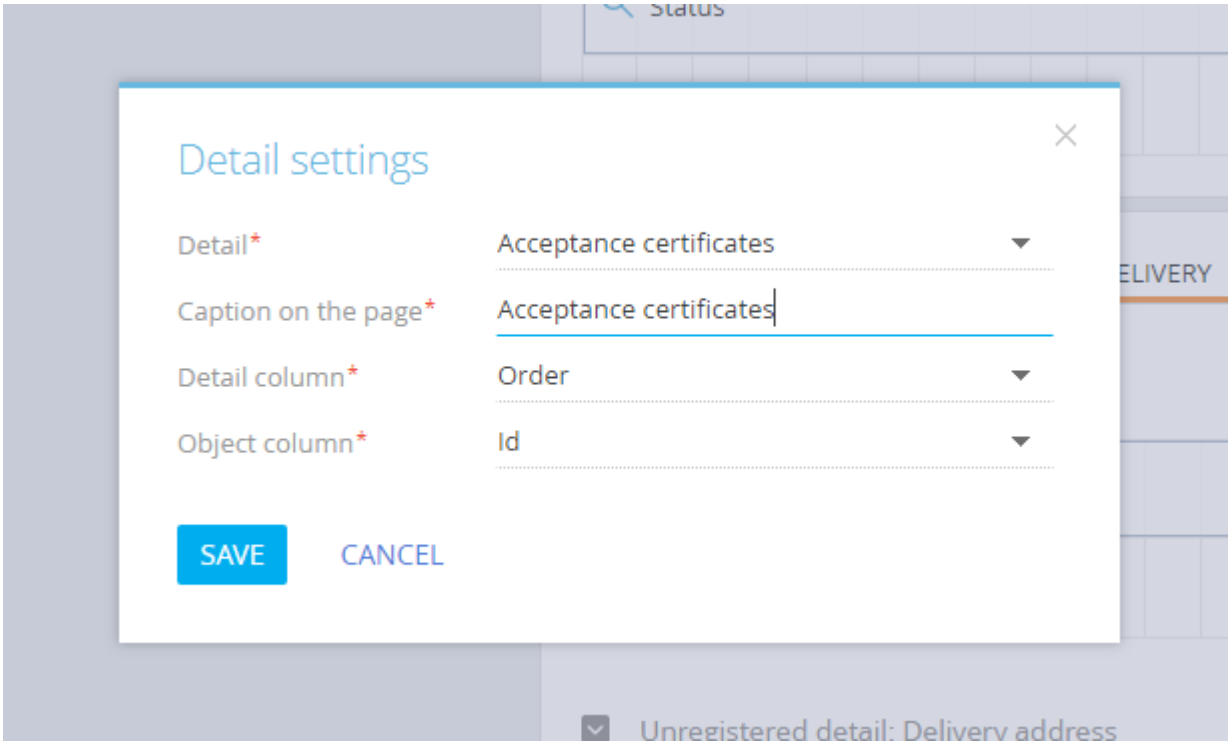


Fig. 6. Adding the detail to the section



As a result, the *UsrSchema1Detail* detail schema and the *OrderPageV2* section replacing schema will be created in the development package.

> ATTENTION
>
> We recommend changing the name of the detail schema to a more unique name, since the auto generated name can be duplicated in a different development package. We replaced the name for *UsrCourierCertDetail* in the current case.
>
> Change the detail schema name for a new one in the *OrderPageV2* section replacing schema.

3. Configuring the detail

Change the source code of the created detail schema:

1. Add dependency from the *ConfigurationEnums* module.
2. Add the *onDocumentInsert()*, *onCardSaved()* event handler-methods, the *openDocumentLookup()* method of calling the modal lookup window and additional data control methods.
3. Add the necessary configuration objects to the *diff* modification array.

The Terrasoft.EntitySchemaQuery class is used for executing database queries in the current case. You can learn more about using EntitySchemaQuery in the "**The use of EntitySchemaQuery for creation of queries in database**" article. To simplify the case, we blocked the detail menu options that enable editing and copying detail list records.

Source code of the detail schema:

```
// Defining the shcema and setting up its dependencies from other modules.
define("UsrCourierCertDetail", ["ConfigurationEnums"],
    function(configurationEnums) {
        return {
            // Name of the detail object schema.
            entitySchemaName: "UsrCourierCertInOrder",
            // Detail schema methods.
            methods: {
                //Returns columns selected by query.
                getGridDataColumns: function() {
                    return {
```

```
                        "Id": {path: "Id"},
                        "Document": {path: "UsrDocument"},
                        "Document.Number": {path: "UsrDocument.Number"}
                    };
                },

                //Configures and displays modal lookup window.
                openDocumentLookup: function() {
                    //Configuration object
                    var config = {
                        // Name of the object schema whose records will be displayed
in the lookup.
                        entitySchemaName: "Document",
                        // Multiple selection option.
                        multiSelect: true,
                        // Columns used in the lookup, e.g., for sorting.
                        columns: ["Number", "Date", "Type"]
                    };
                    var OrderId = this.get("MasterRecordId");
                    if (this.Ext.isEmpty(OrderId)) {
                        return;
                    }
                    // The [EntitySchemaQuery] class instance.
                    var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
                        // Setting up the root schema.
                        rootSchemaName: this.entitySchemaName
                    });
                    // Adding the [Id] column.
                    esq.addColumn("Id");
                    // Adding the [Id] column for the [Document] schema.
                    esq.addColumn("Document.Id", "DocumentId");
                    // Creating and adding filters to query collection.
                    esq.filters.add("filterOrder",
this.Terrasoft.createColumnFilterWithParameter(
                        this.Terrasoft.ComparisonType.EQUAL, "UsrOrder", OrderId));
                    // Receiving the whole record collection and its display in the
modal lookup window.
                    esq.getEntityCollection(function(result) {
                        var existsDocumentsCollection = [];
                        if (result.success) {
                            result.collection.each(function(item) {

existsDocumentsCollection.push(item.get("DocumentId"));
                            });
                        }
                        // Adding filter to the configuration object.
                        if (existsDocumentsCollection.length > 0) {
                            var existsFilter =
this.Terrasoft.createColumnInFilterWithParameters("Id",
                                existsDocumentsCollection);
                            existsFilter.comparisonType =
this.Terrasoft.ComparisonType.NOT_EQUAL;
                            existsFilter.Name = "existsFilter";
                            config.filters = existsFilter;
                        }
                        // Call of the modal lookup window
                        this.openLookup(config, this.addCallBack, this);
                    }, this);
                },

                // Event handler of saving the edit page.
                onCardSaved: function() {
```

```
                    this.openDocumentLookup();
                },

                //Opens the document lookup if the order edit page has been saved.
                addRecord: function() {
                    var masterCardState = this.sandbox.publish("GetCardState", null,
[this.sandbox.id]);
                    var isNewRecord = (masterCardState.state ===
configurationEnums.CardStateV2.ADD ||
                    masterCardState.state === configurationEnums.CardStateV2.COPY);
                    if (isNewRecord === true) {
                        var args = {
                            isSilent: true,
                            messageTags: [this.sandbox.id]
                        };
                        this.sandbox.publish("SaveRecord", args, [this.sandbox.id]);
                        return;
                    }
                    this.openDocumentLookup();
                },

                // Adding the selected products.
                addCallBack: function(args) {
                    // Class instance of the BatchQuery package query.
                    var bq = this.Ext.create("Terrasoft.BatchQuery");
                    var OrderId = this.get("MasterRecordId");
                    // Collection of the selected documents from the lookup.
                    this.selectedRows = args.selectedRows.getItems();
                    // Collection passed over to query.
                    this.selectedItems = [];
                    // Copying the necessary data.
                    this.selectedRows.forEach(function(item) {
                        item.OrderId = OrderId;
                        item.DocumentId = item.value;
                        bq.add(this.getDocumentInsertQuery(item));
                        this.selectedItems.push(item.value);
                    }, this);
                    // Executing the package query if it is not empty.
                    if (bq.queries.length) {
                        this.showBodyMask.call(this);
                        bq.execute(this.onDocumentInsert, this);
                    }
                },

                //Returns query for adding the current object.
                getDocumentInsertQuery: function(item) {
                    var insert = Ext.create("Terrasoft.InsertQuery", {
                        rootSchemaName: this.entitySchemaName
                    });
                    insert.setParameterValue("UsrOrder", item.OrderId,
this.Terrasoft.DataValueType.GUID);
                    insert.setParameterValue("UsrDocument", item.DocumentId,
this.Terrasoft.DataValueType.GUID);
                    return insert;
                },

                //Method called when adding records to the detail record list.
                onDocumentInsert: function(response) {
                    this.hideBodyMask.call(this);
                    this.beforeLoadGridData();
                    var filterCollection = [];
                    response.queryResults.forEach(function(item) {
```

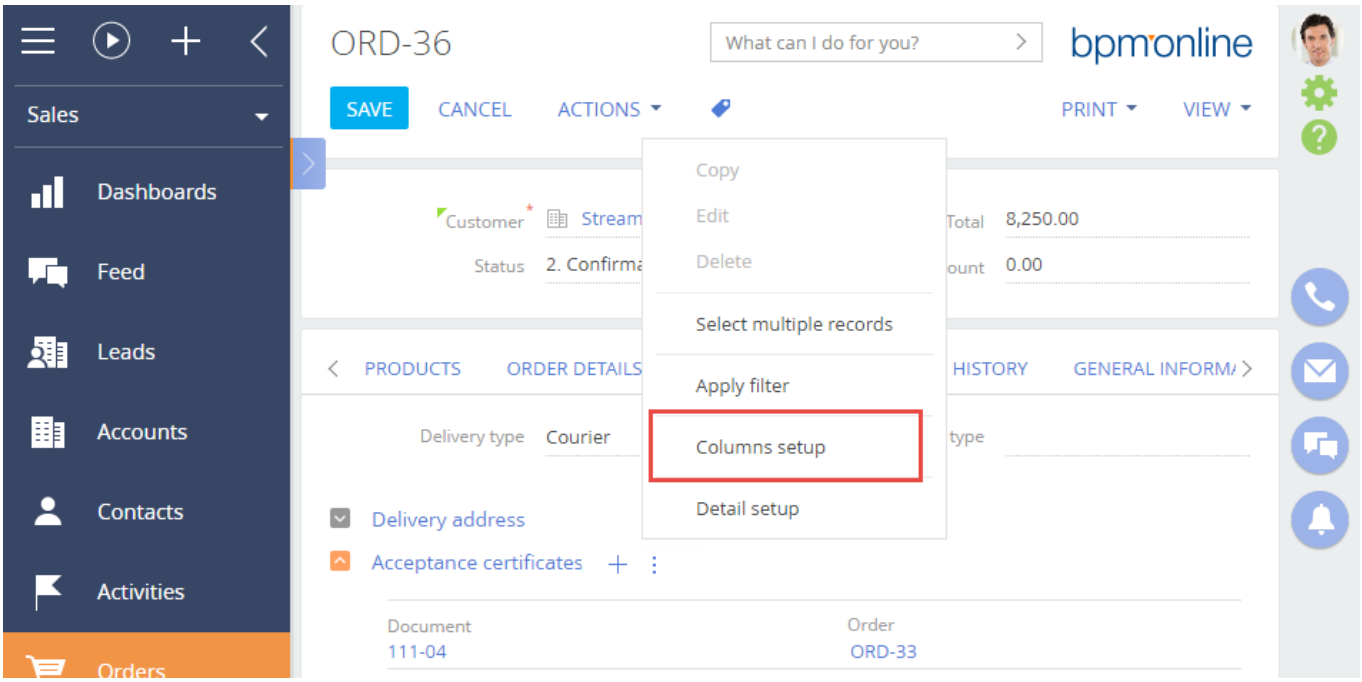
```
                        filterCollection.push(item.id);
                    });
                    var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
                        rootSchemaName: this.entitySchemaName
                    });
                    this.initQueryColumns(esq);
                    esq.filters.add("recordId",
Terrasoft.createColumnInFilterWithParameters("Id", filterCollection));
                    esq.getEntityCollection(function(response) {
                        this.afterLoadGridData();
                        if (response.success) {
                            var responseCollection = response.collection;
                            this.prepareResponseCollection(responseCollection);
                            this.getGridData().loadAll(responseCollection);
                        }
                    }, this);
                },

                // Method called when deleting records from the detail record list.
                deleteRecords: function() {
                    var selectedRows = this.getSelectedItems();
                    if (selectedRows.length > 0) {
                        this.set("SelectedRows", selectedRows);
                        this.callParent(arguments);
                    }
                },

                // Hide the [Copy] menu option.
                getCopyRecordMenuItem: Terrasoft.emptyFn,
                 // Hide the [Edit] menu option.
                getEditRecordMenuItem: Terrasoft.emptyFn,
                // Returns the default filter column name.
                getFilterDefaultColumnName: function() {
                    return "UsrDocument";
                }
            },
            // Modification array.
            diff: /**SCHEMA_DIFF*/[
                {
                    // Operation type - merging.
                    "operation": "merge",
                    // Name of the schema element under operation.
                    "name": "DataGrid",
                    // The object, whose properties will be combined with the schema
element properties.
                    "values": {
                        "rowDataItemMarkerColumnName": "UsrDocument"
                    }
                },
                {
                    // Operation type - merging.
                    "operation": "merge",
                    // Name of the schema element under operation.
                    "name": "AddRecordButton",
                    // The object, whose properties will be combined with the schema
element properties.
                    "values": {
                        "visible": {"bindTo": "getToolsVisible"}
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
```

```
        }
);
```

4. Setting up detail list columns

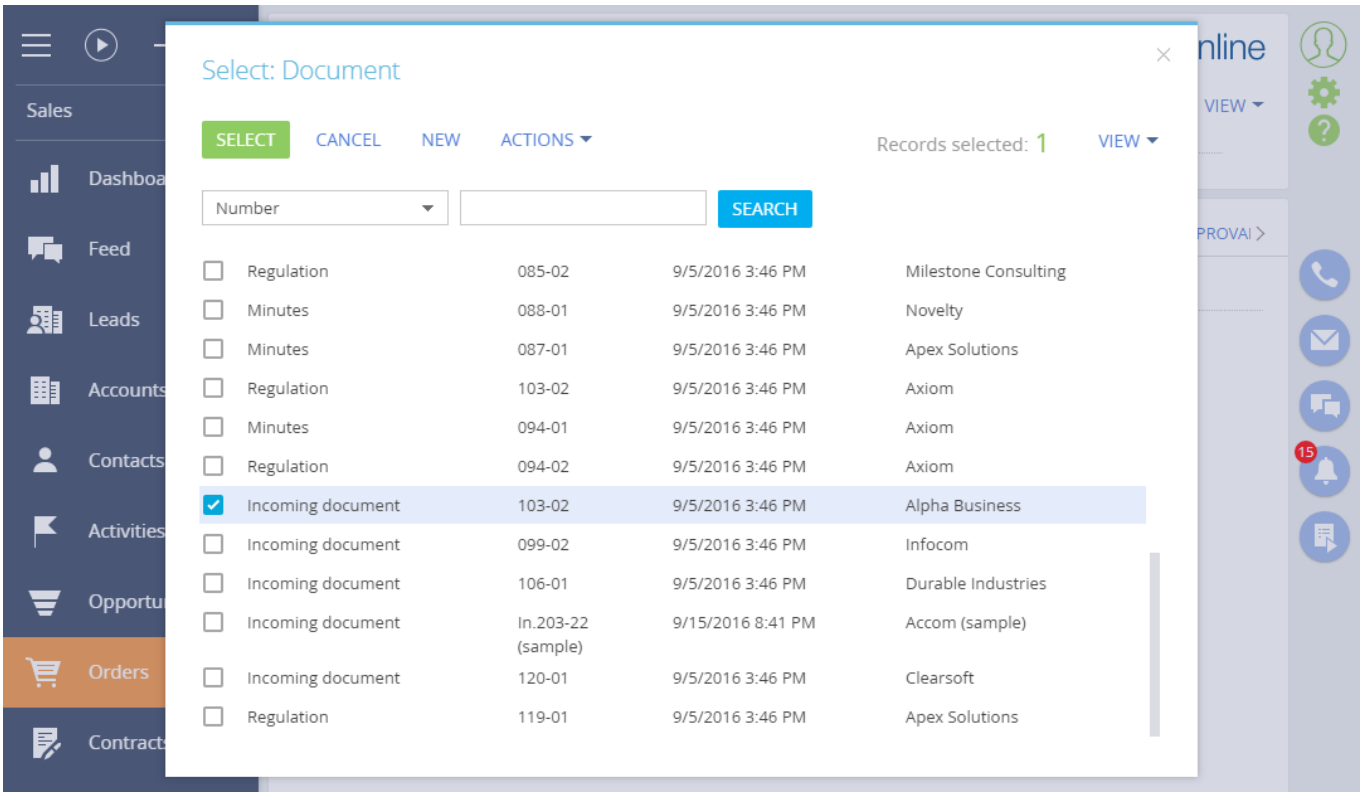
At this stage the detail is completely operable but contact records are not displayed on the detail as the display columns are not specified. Call the detail action menu and set up the column display (fig. 7).

Fig. 7. Detail action menu



As a result, the new detail will enable adding records from the document lookup via the modal window (Fig. 8).

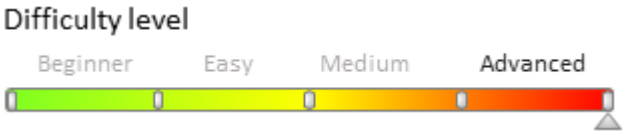
Fig. 8. Case result



See also

- **Creating a detail in wizards**
- **Adding a detail with an editable list**
- **Adding an edit page detail**
- **Creating a custom detail with fields**

Adding multiple records to a detail



Introduction

Normally, a detail allows you to only add one record. Adding multiple entries to a detail is done through the *LookupMultiAddMixin* mixin.

A mixin is a class designed to extend the functions of other classes. Learn more about mixins in the “**Mixins. The "mixins" property**”.

The *LookupMultiAddMixin* is designed to extend the detail schemas. It provides an opportunity to add multiple lookup records to the detail at the same time.

The sequence of adding the multiple records selection functionality to a detail:

1. Create a replacing schema of the detail.

2. Use mixin methods instead of base detail methods.

Case description

Implement the possibility of multiple records selection in the [Contacts] detail on the [Sales] section records edit page.

Source code

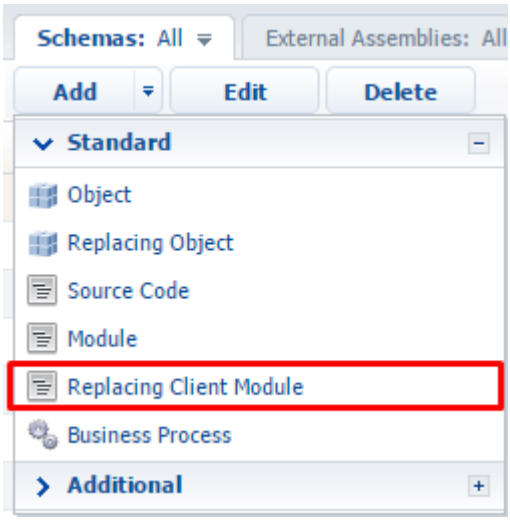
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Create a replacing view model schema of a detail

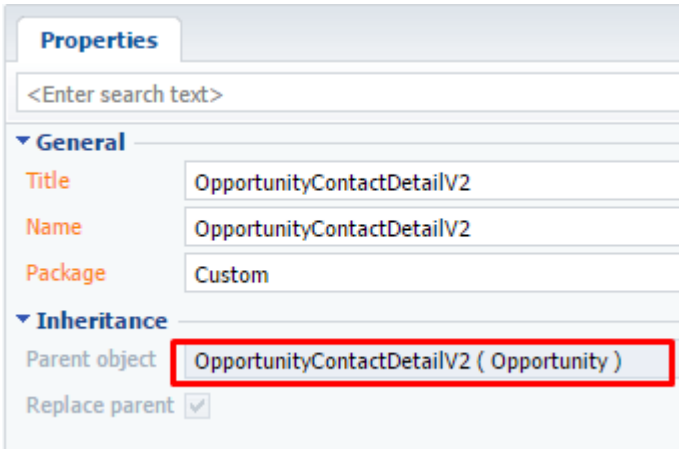
Create a detail replacing schema in the development package (Fig. 1). The procedure for creating a replacing schema is covered in the “**Creating a custom client module schema**”.

Fig. 1. Adding replacing client module



Select the *OpportunityContactDetailV2* schema as the parent object (Fig. 2).

Fig. 2. Properties of the replacing client module



2. Use mixin methods instead of base detail methods.

To use the *LookupMultiAddMixin* mixin in a schema, add it to the *mixins* property and initialize it in the pre-defined *init()* schema method. Learn more about pre-defining the *init()* method in the " **Modular development principles in bpm'online**" article.

To implement the required functionality, you need to pre-define the "add" button displaying methods (*getAddRecordButtonVisible()*), saving detail page methods (*onCardSaved()*), and adding a detail record methods (*addRecord()*).

Saving a detail page and adding record methods include the method of calling the help window for multiple selection *openLookupWithMultiSelect()* and the associated method *getMultiSelectLookupConfig()* which configures the help window, is used. The description and parameters of these methods are given in Table 1.

Table 1. Methods for calling and configuring the help window

Methods.	Description
openLookupWithMultiSelect(isNeedCheckOfNew)	Opens a lookup window with a multiple selection option The *isNeedCheckOfNew {bool}* parameter indicates the need to check if the record is new.
getMultiSelectLookupConfig()	Returns the configuration object for the help window. Object properties: *rootEntitySchemaName* – root object schema; *rootColumnName* – a connected column that indicates the

Methods.	Description
	root schema record; *relatedEntitySchemaName* – connected schema; *relatedColumnName* – a column that indicates the connected schema record.

In this case the help window will pull data from the *OpportunityContact* table using the *Opportunity* and *Contact* columns.

Source code of the detail schema:

```
define("OpportunityContactDetailV2", ["LookupMultiAddMixin"], function() {
    return {
        mixins: {
            // Connecting the mixin to the schema.
            LookupMultiAddMixin: "Terrasoft.LookupMultiAddMixin"
        },
        methods: {
            // Overriding the base method for initializing the schema.
            init: function() {
                this.callParent(arguments);
                //Initializing the mixin.
                this.mixins.LookupMultiAddMixin.init.call(this);
            },
            // Overriding the base method for displaying the "Add" button.
            getAddRecordButtonVisible: function() {
                //Displaying the "add" button if the detail is maximized, even if the
detail edit page is not implemented.
                return this.getToolsVisible();
            },
            // Overriding the base method.
            // The save event handler for the detail edit page.
            onCardSaved: function() {
                // Opens the window for multiple record selection.
                this.openLookupWithMultiSelect();
            },
            // Overriding the base method of adding a detail record.
            addRecord: function() {
                // Opens the window for multiple records selection.
                this.openLookupWithMultiSelect(true);
            },
            // A method that returns a window configuration object.
            getMultiSelectLookupConfig: function() {
                return {
                    // Root schema — [Opportunities].
                    rootEntitySchemaName: "Opportunity",
                    // Root schema column.
                    rootColumnName: "Opportunity",
                    // Connected schema — [Contact].
                    relatedEntitySchemaName: "Contact",
                    // Root schema column.
                    relatedColumnName: "Contact"
                };
            }
        }
    };
});
```

After saving the schema and reloading the application page, the user will be able to select multiple records from the lookup (Fig. 4) by clicking the "add" detail record button (Fig. 3). All selected records will be added to the [Contacts]

detail of the [Opportunities] section record edit page (Fig. 5).

Fig. 3. Adding multiple records to a detail

Fig. 4. Selecting necessary records from a lookup

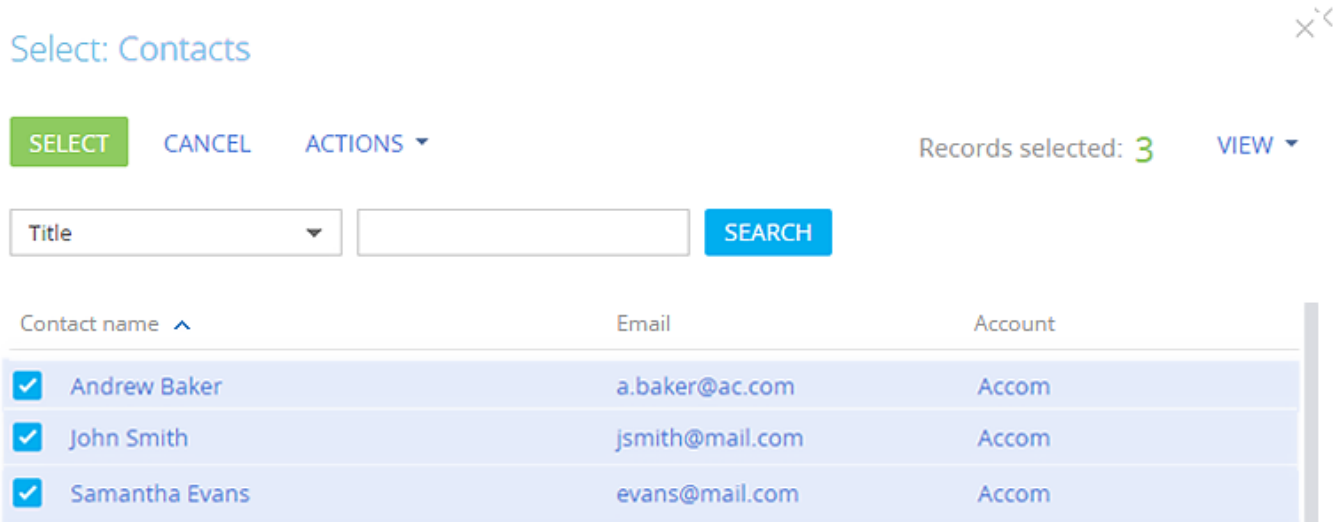
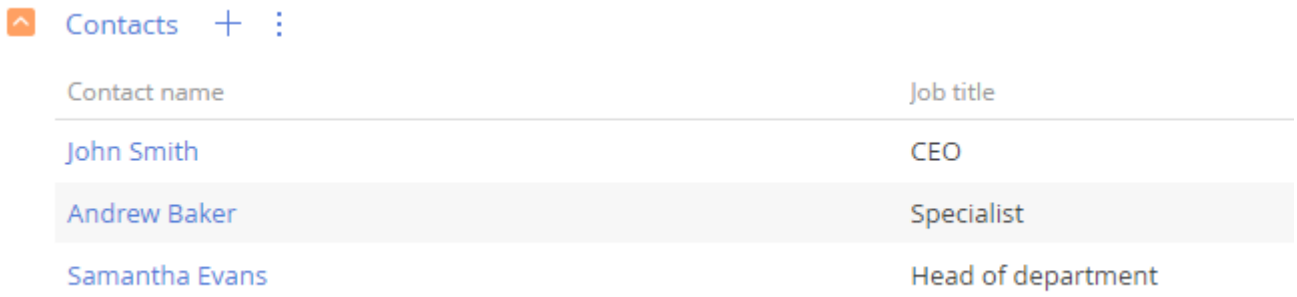
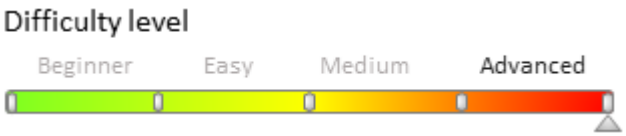


Fig. 5. Case result: All selected records are added to a detail



Creating a custom detail with fields



Introduction

A detail with fields can include multiple field groups. The base detail with fields is implemented in the *BaseFieldsDetail* schema of the *BaseFinance* package, which is available in bpm'online bank customer journey, bank sales and lending. The detail record view model is implemented in the *BaseFieldRowViewModel* schema.

Base detail with fields enables you to:

- Add detail records without saving a page.

- Work with a detail like you would with an edit page.
- Use the base field validation with the ability to add a custom one.
- Add a virtual record.
- Expand record behavior logic.

A custom detail with fields can be **created** with the help of the base detail (a custom detail schema should be inherited from the base detail schema).

Case description

Implement a custom detail with fields for document registration. The detail should be populated with records that include the document's [Number] and [Series] fields. The detail should be located on the [History] tab of the contact edit page.

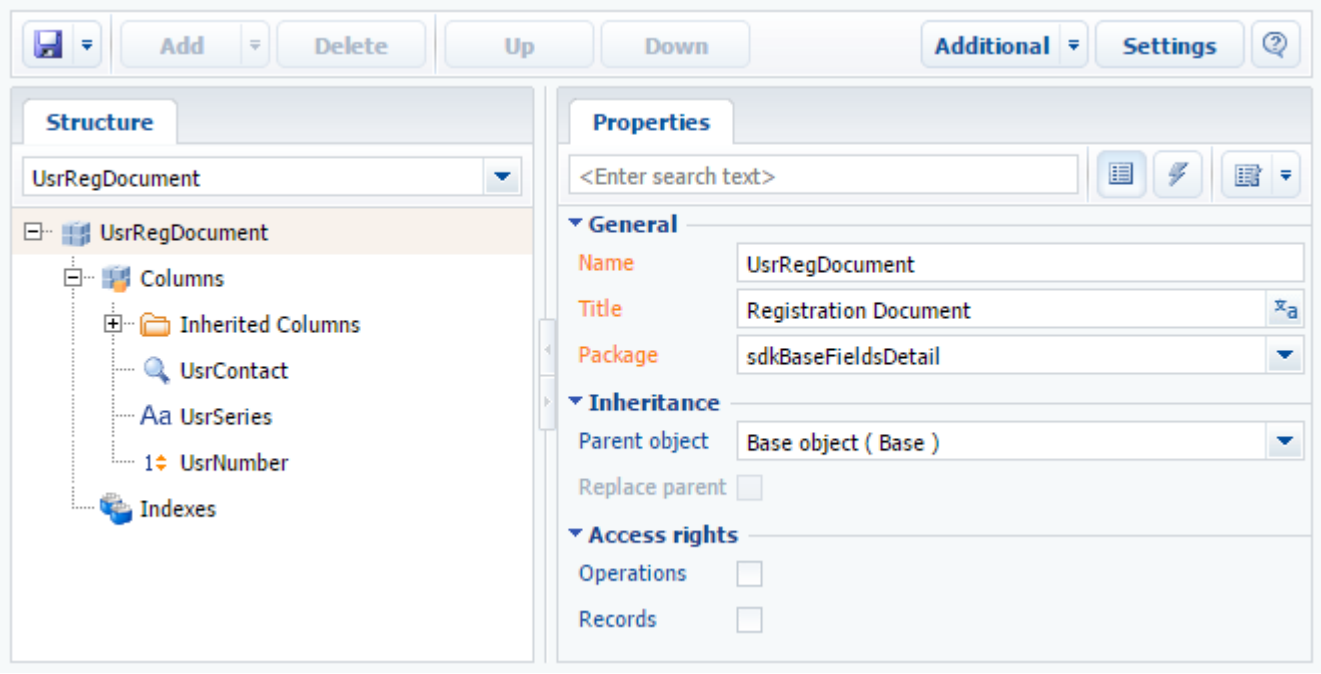
Case implementation algorithm

1. Create a detail object schema

Create a new object schema in a custom package with the following property values:

- [Title] – "Registration document".
- [Name] – "UsrRegDocument".
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down list.
- [Parent object] – "Base object", implemented in the Base package.

Fig. 1. Object schema properties



Add three columns in the object structure. Column properties are listed in Table 1. Learn more about adding object schema columns in the "**Creating the entity schema**" article.

Table 1. — Column properties of the UsrRegDocument detail object schema

Title	Name	Data Type
Contact	UsrContact	Lookup
Series	UsrSeries	Text (50 characters)

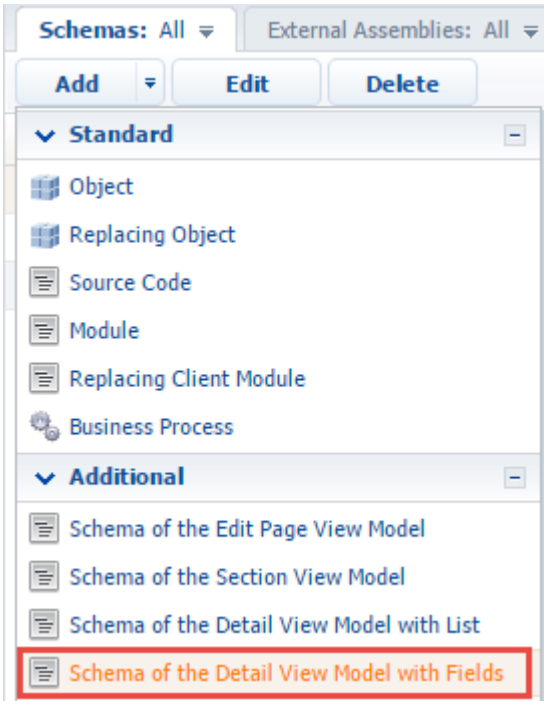
Number	UsrNumber	Integer

Publish the schema to apply changes.

2. Create a view model schema for the custom detail with fields.

Add a custom schema([Schema of the Detail View Model with Fields]) in a custom package (Fig. 2).

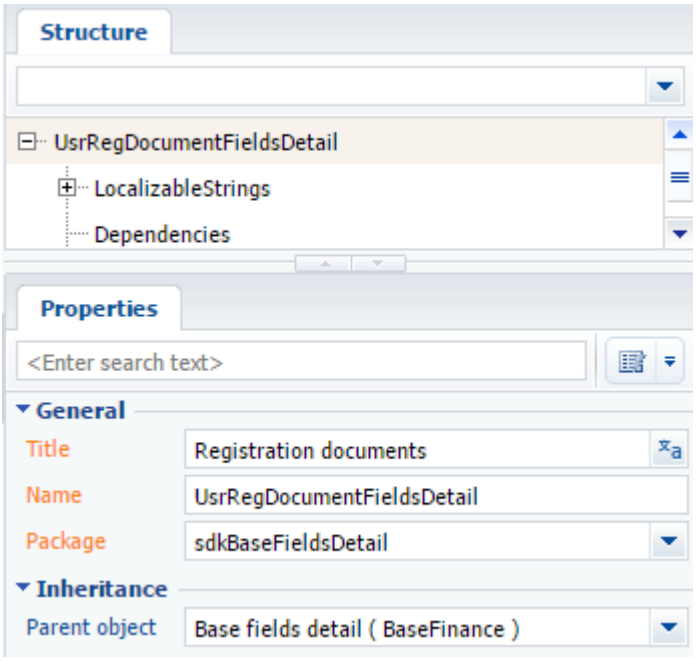
Fig. 2. Adding a custom view model schema for the custom detail with fields



Property values for the created schema (Fig. 3):

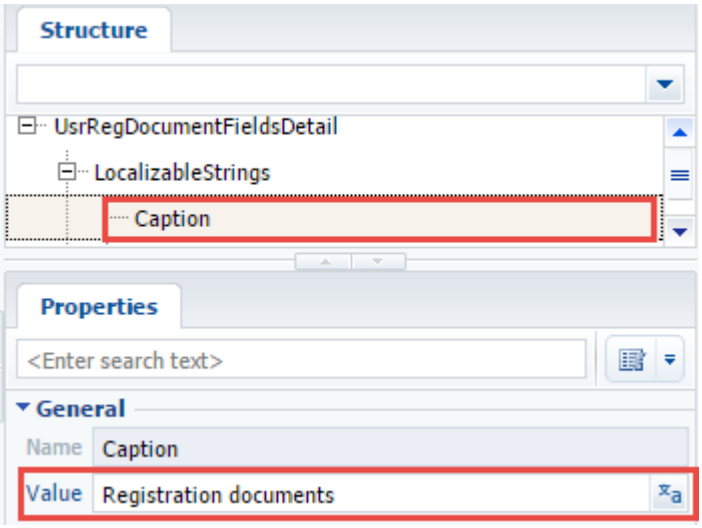
- [Title] – “Registration documents”.
- [Name] – “UsrRegDocumentFieldsDetail”.
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down list.
- [Parent object] – "Base fields detail", implemented in the BaseFinance package.

Fig. 3. UsrRegDocumentFieldsDetail client schema properties



Assign the “Registration documents” value to the localizable [Caption] string of the [Value] property.

Fig. 4. Localizable string properties



Create a module description and redefine the base *getDisplayColumns()* method, which returns column names that are displayed as detail fields. By default, this method returns all required columns, as well as the column with the set [Displayed value] checkbox in the object schema.

Schema source code:

```
define("UsrRegDocumentFieldsDetail", [],
    function() {
        return {
            entitySchemaName: "UsrRegDocument",
            diff: /**SCHEMA_DIFF*/ [], /**SCHEMA_DIFF*/
            methods: {
                getDisplayColumns: function() {
                    return ["UsrSeries", "UsrNumber"];
                }
            }
        };
    });
```

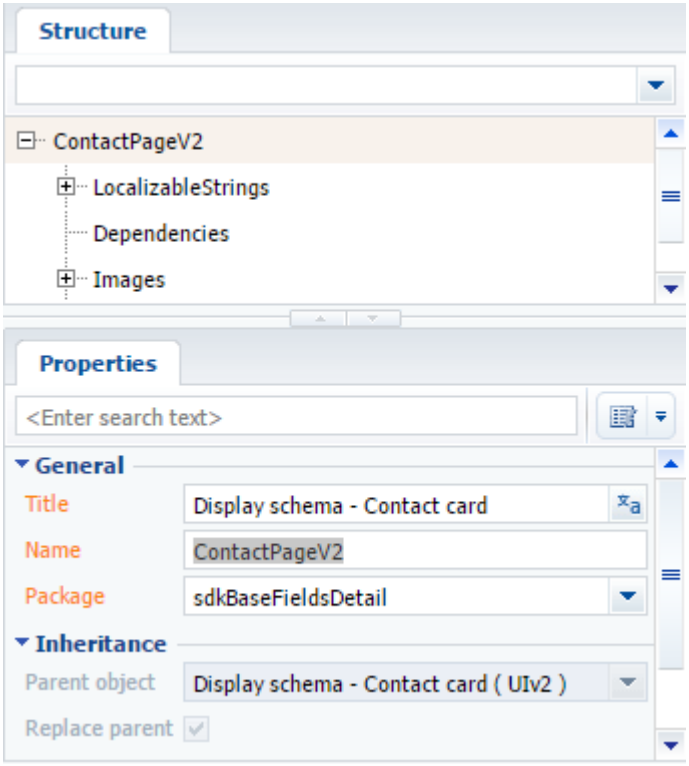
Save the schema to apply changes.

3. Create a replacing schema for the edit page

To do this, create a **replacing schema** of the contact edit page (*ContactPageV2*). Main replacing schema properties (Fig. 5):

- [Title] – "Display schema - Contact card".
- [Name] – "ContactPageV2".
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down list.
- [Parent object] – "Display schema - Contact card", implemented in the *UIv2* package.

Fig. 5. The ContactPageV2 replacing schema properties



Make the following adjustments to the source code:

- **Create a module description**
- Add a detail in the **"details" property**
- Add a configuration object of the detail's view model to the **"diff" modification array**.

Schema source code:

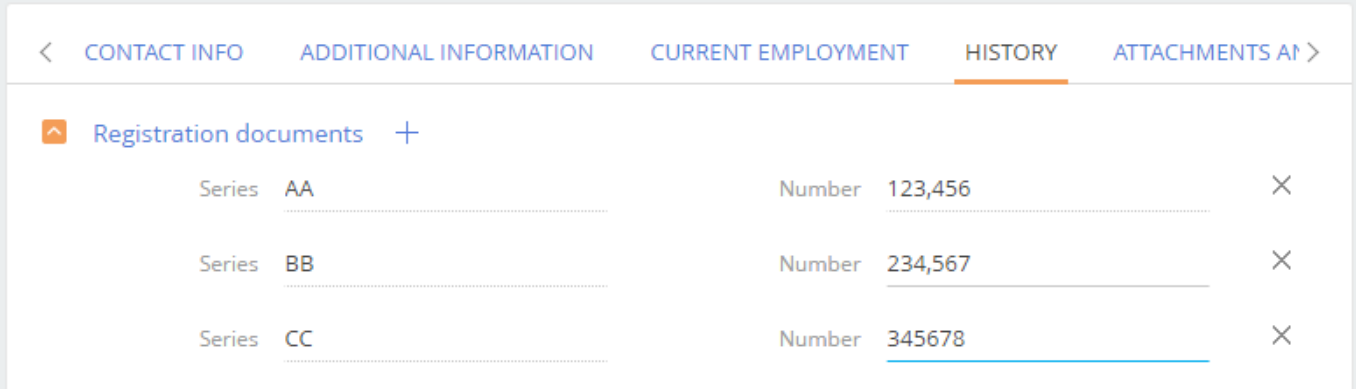
```
define("ContactPageV2", [], function() {
    return {
        entitySchemaName: "Contact",
        details: /**SCHEMA_DETAILS*/ {
            // Adding a field with details.
            "UsrRegDocumentFieldsDetail": {
                // Name of the custom detail schema.
                "schemaName": "UsrRegDocumentFieldsDetail",
                // Filtering current contact's record details.
                "filter": {
                    // Detail object column.
                    "detailColumn": "UsrContact",
                    // Contact's Id column.
```

```
                    "masterColumn": "Id"
                }
            }
        } /**SCHEMA_DETAILS*/ ,
        diff: /**SCHEMA_DIFF*/ [{
            // Adding a new element.
            "operation": "insert",
            // Element name.
            "name": "UsrRegDocumentFieldsDetail",
            // Value configuration object.
            "values": {
                // Element type.
                "itemType": Terrasoft.ViewItemType.DETAIL
            },
            // Container element name.
            "parentName": "HistoryTab",
            // Property name of the container element with the collection of nested
elements.
            "propertyName": "items",
            // The index of the element added to the collection.
            "index": 0
        }] /**SCHEMA_DIFF*/
    };
});
```

Save the schema to apply changes.

The [History] tab of the contact edit page should now have the custom detail with the [Registration documents] fields (Fig. 6).

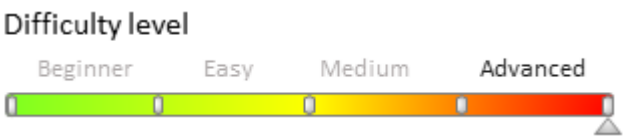
Fig. 6. Case result



ATTENTION

Register the detail with fields (similarly to the **detail with the editable grid**) in the system to make it visible in detail and section wizards.

Advanced settings of a custom detail with fields



Introduction

A detail with fields can include multiple field groups. The base detail with fields is implemented in the

BaseFieldsDetail schema of the *BaseFinance* package, which is available in bpm'online bank customer journey, bank sales and lending. The detail record view model is implemented in the *BaseFieldRowViewModel* schema.

The process of creating a custom detail with fields is described in a separate **Creating a custom detail with fields**.

Adding custom styles

Case description

Redefine the field signature style for a detail implemented in the “**Creating a custom detail with fields**” article. The field signatures should be displayed in blue.

NOTE

You can override the basic CSS style classes for displaying detail records by using the *getLeftRowContainerWrapClass()* and *getRightRowContainerWrapClass()* methods.

Case implementation algorithm

1. Create a module schema and define record view styles

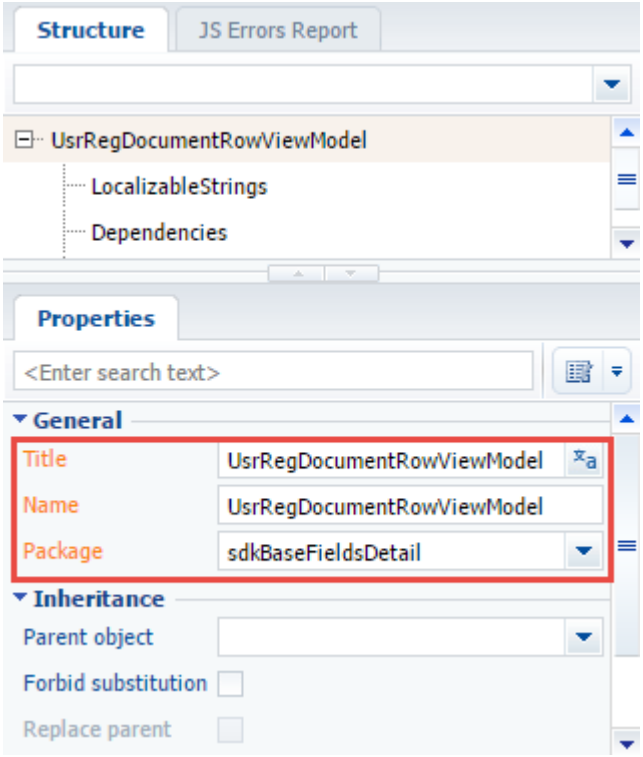
ATTENTION

You can not set the styles in a view model schema of the edit page. It is necessary to create a new module schema, define the styles and add the created module to module dependencies of a detail.

Create a new module schema in a custom package with the following property values:

- [Title] – “UsrRegDocumentRowViewModel”.
- [Name] – “UsrRegDocumentRowViewModel”.
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down list.

Fig. 1. UsrRegDocumentFieldsDetail custom schema properties



Create a module description, and define the *Terrasoft.configuration.UsrRegDocumentRowViewModel* class which is inherited from *Terrasoft.configuration.BaseFieldRowViewModel*.

The source code of the schema:

```
define("UsrRegDocumentRowViewModel", ["BaseFieldRowViewModel"], function() {
    Ext.define("Terrasoft.configuration.UsrRegDocumentRowViewModel", {
        extend: "Terrasoft.BaseFieldRowViewModel",
        alternateClassName: "Terrasoft.UsrRegDocumentRowViewModel"
    });
    return Terrasoft.UsrRegDocumentRowViewModel;
});
```

Define the CSS view classes for the correct display of detail records. To do this, add the following CSS classes to the LESS tab of the module designer:

```
.reg-document-left-row-container {
    .t-label {
        color: blue;
    }
}
.field-detail-row {
    width: 100%;
    display: inline-flex;
    margin-bottom: 10px;

    .field-detail-row-left {
        display: flex;
        flex-wrap: wrap;
        width: 100%;

        .control-width-15 {
            min-width: 300px;
            width: 50%;
            margin-bottom: 5px;
        }
        .control-width-15:only-child {
            width: 100% !important;
        }
    }
    .field-detail-row-left.singlecolumn {
        width: 50%;
    }
}
```

Save the schema to apply changes.

2. Modifying a replacing view model schema of a detail

To use the created module and its styles in a detail schema, add it to the dependency of the module defined in the detail schema.

Additionally, add the following methods to a detail schema module:

- *getRowViewModelClassName()* – returns the name of the record view model class to the detail.
- *getLeftRowContainerWrapClass()* – returns the string array with CSS class names, used to generate the views of record signature field containers.

The source code of the modified schema:

```
define("UsrRegDocumentFieldsDetail", ["UsrRegDocumentRowViewModel",
"css!UsrRegDocumentRowViewModel"],
    function() {
        return {
```

```
            entitySchemaName: "UsrRegDocument",
            diff: /**SCHEMA_DIFF*/ [], /**SCHEMA_DIFF*/
            methods: {
                getDisplayColumns: function() {
                    return ["UsrSeries", "UsrNumber"];
                },
                getRowViewModelClassName: function() {
                    return "Terrasoft.UsrRegDocumentRowViewModel";
                },
                getLeftRowContainerWrapClass: function() {
                    return ["reg-document-left-row-container", "field-detail-row"];
                }
            }
        };
    });
```

Save the schema to apply changes.

On the [History] tab of the contact edit page, the detail names will be displayed in blue (Fig. 2).

Fig. 2. Case result

Adding additional custom logic for detail records

Case description

Add the field validation to the [Number] field of the detail, implemented in the “**Creating a custom detail with fields**” article. The field value can not be negative.

Case implementation algorithm

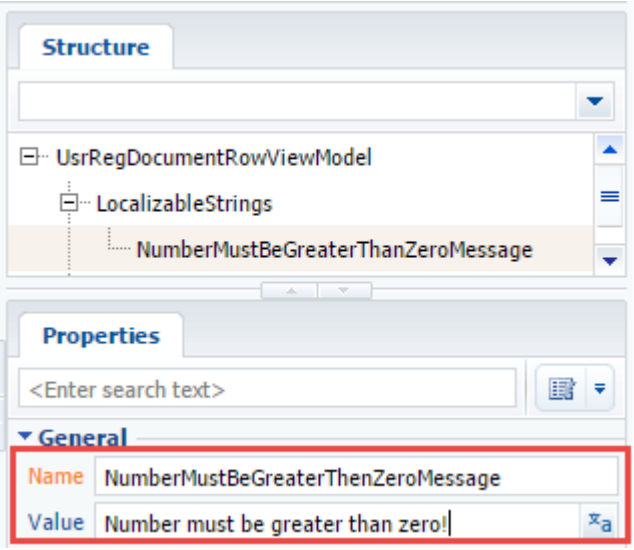
1. Adding a localizable string with the error message

In the module designer, add the localizable string on the [Structure] tab of the opened *UsrRegDocumentRowViewModel* schema with the following property values

(Fig. 3):

- [Name] – “NumberMustBeGreaterThenZeroMessage”.
- [Value] – “Number must be greater than zero!”.

Fig. 3. Case result



NOTE

A localized string is a schema resource. In order for its values to appear in the client part of the application, add the *UsrRegDocumentRowViewModelResources* resource module to the dependencies of the *UsrRegDocumentRowViewModel* module.

Save the schema to apply changes.

2. Adding the validation program logic

Add the following methods to the *UsrRegDocumentRowViewModel* module to implement the validation program logic:

- *validateNumberMoreThenZero()* – contains the validation logic of the field value.
- *setValidationConfig()* – connects the [Number] column and the validateNumberMoreThenZero() validation method.
- *Init()* – an overridden base method that calls the base logic and the *setValidationConfig()* method.

Source code of the modified schema:

```
define("UsrRegDocumentRowViewModel", ["UsrRegDocumentRowViewModelResources",
"BaseFieldRowViewModel"],
    function(resources) {
        Ext.define("Terrasoft.configuration.UsrRegDocumentRowViewModel", {
            extend: "Terrasoft.BaseFieldRowViewModel",
            alternateClassName: "Terrasoft.UsrRegDocumentRowViewModel",
            validateNumberMoreThenZero: function(columnValue) {
                var invalidMessage;
                if (columnValue < 0) {
                    invalidMessage =
resources.localizableStrings.NumberMustBeGreaterThanZeroMessage;
                }
                return {
                    fullInvalidMessage: invalidMessage,
                    invalidMessage: invalidMessage
                };
            },
            setValidationConfig: function() {
                this.addColumnValidator("UsrNumber",
this.validateNumberMoreThenZero);
            },
            init: function() {
                this.callParent(arguments);
```

```
                this.setValidationConfig();
            }
        });
        return Terrasoft.UsrRegDocumentRowViewModel;
    });
```

Save the schema to apply changes.

When a negative value is entered in the [Number] field on the [History] tab of the contact edit page, a warning message will be displayed (Fig. 4).

Fig. 4. Case result

Adding a virtual record

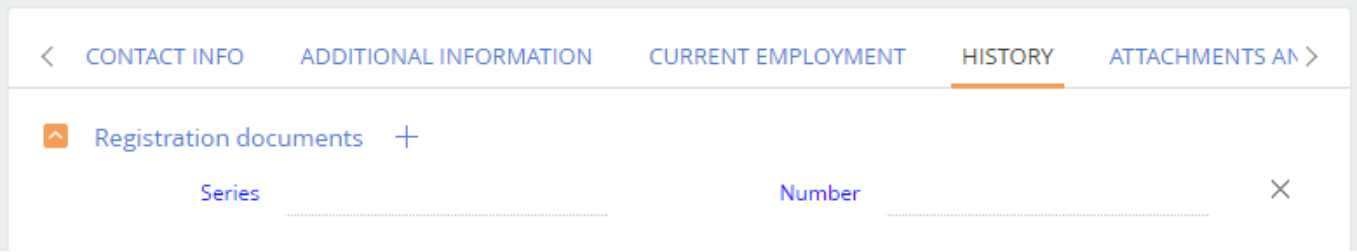
When a detail is loaded, adding virtual records enables you to display a field edit card immediately, without pressing the [Add] button.

That requires defining the *useVirtualRecord()* method (returns *true*) in the *UsrRegDocumentFieldsDetail* detail schema:

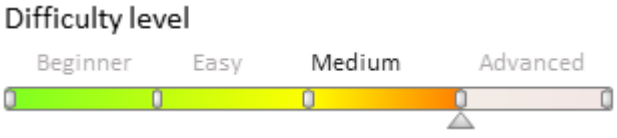
```
useVirtualRecord: function() {
  return true;
}
```

When opening a tab with a detail, a virtual record will be displayed (Fig. 5).

Fig. 5. Displaying a virtual record



Creating a detail in wizards



Introduction

Detail is an element of the section record edit page, designed to display additional data related to the main section object. Details are displayed on edit page tabs. The difference between details and section records is that details are stored in a separate object, and the records in this database object are usually associated with the main section record entity with the "many-to-one" ratio. Please refer to the “**Details**” article in the “**Application interface and structure**” section for more information.

Details are standard bpm'online elements and can be added to the section by using a detail wizard or a section wizard.

The general outline for adding a detail with an “add” page to an existing system section:

1. Create a detail object schema
2. Create a schema of a client detail list module and the schema of a detail edit page
3. Implement the detail on the section record edit page using the detail wizard.

Case description

Create a custom [Contact’s ID] detail in the [Contacts] section. The detail must display the contact’s ID number and the document number. One contact may have several ID’s.

Case implementation algorithm

1. Creating a detail object schema

Learn more about adding object schema columns in the “**Creating the entity schema**” article.

Create an object schema with the following parameters (Fig. 1):

- [Title] – “Contact Identity Card”.
- [Name] — “UsrContactIdentityCard”.
- [Package] — "Custom" (or a different custom package).
- [Parent object] – “Base object”, implemented in the *Base* package.

Add three columns in the object structure. Column properties are listed in Table 1.

Table 1. – Column properties of the UsrRegDocument detail object schema

Title	Name	Data Type	Lookup
Series	UsrSeries	Text (50 characters) Text (50 characters)	–
Number	UsrNumber	Text (50 characters) Text (50 characters)	–
Contact	UsrContact	Lookup	Contact

Publish the schema to apply changes.

> NOTE
>
> Add columns in the detail wizard.

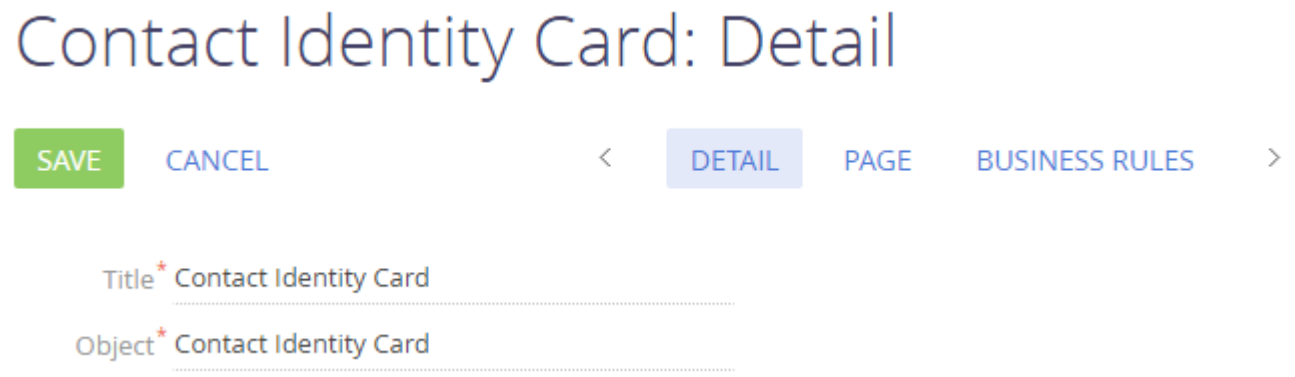
2. Creating a schema of the detail list client module and a schema of the detail edit page.

> ATTENTION
>
> If the development needs to be carried out in a custom package, it needs to be specified in the [Current package] system setting. Otherwise, the detail wizard will not be able to save the changes to the package used for development.

To create a new detail in a wizard, go to the [Detail wizard] section in the [System setup] group of the system designer.

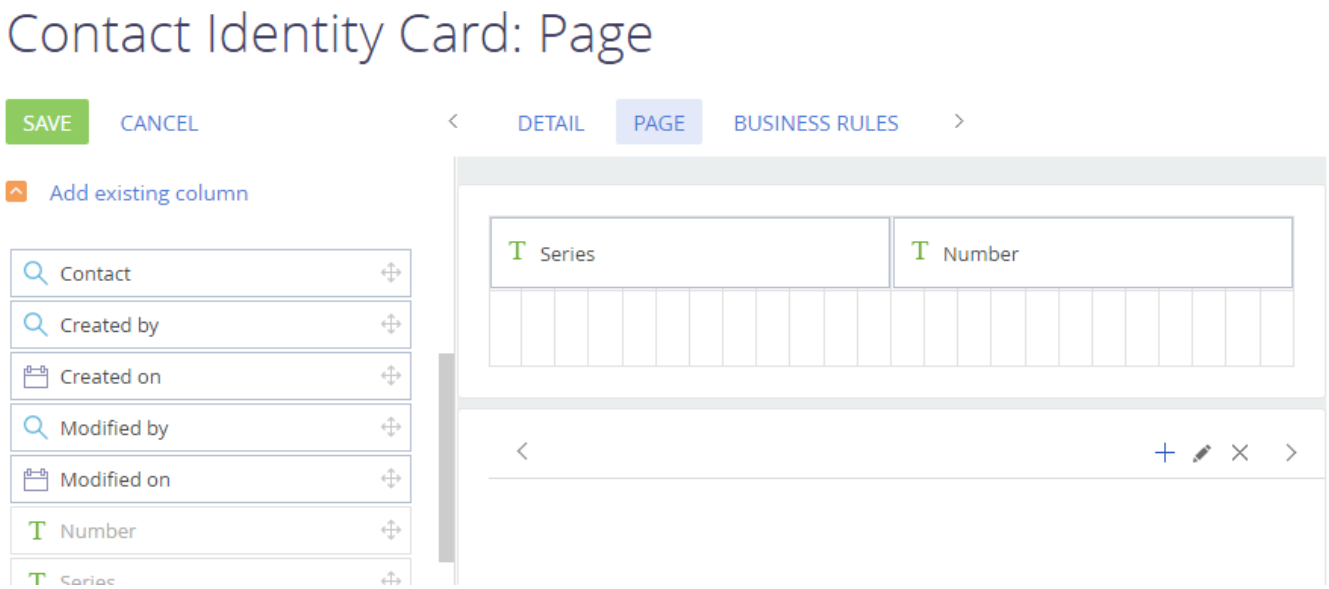
On the [DETAIL] step, specify the title of the detail and select the main detail object (Fig. 1).

Fig. 1. The [DETAIL] step in the detail wizard



Arrange the required detail columns on the [PAGE] step.

Fig. 2. The [PAGE] step in the detail wizard



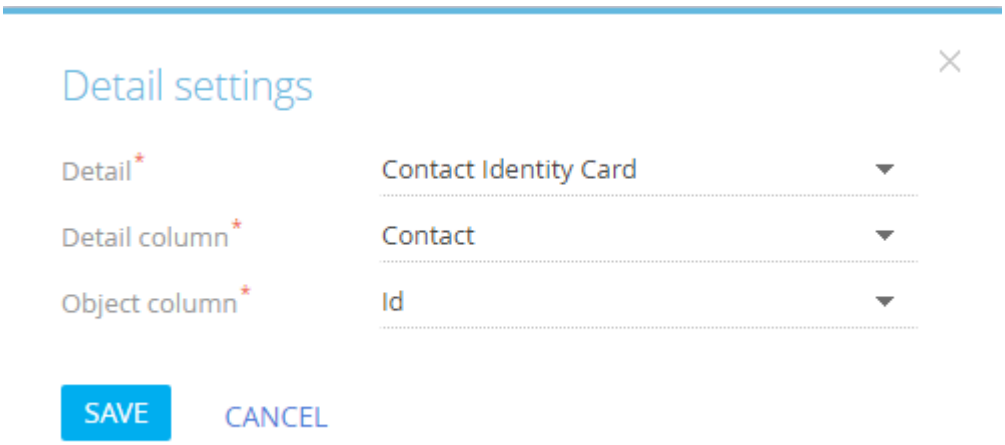
Save the detail when the setup is done.

As a result, the custom package will have a schema of the detail list client module and a schema of the detail edit page.

3. Implement the detail on the section record edit page using the detail wizard

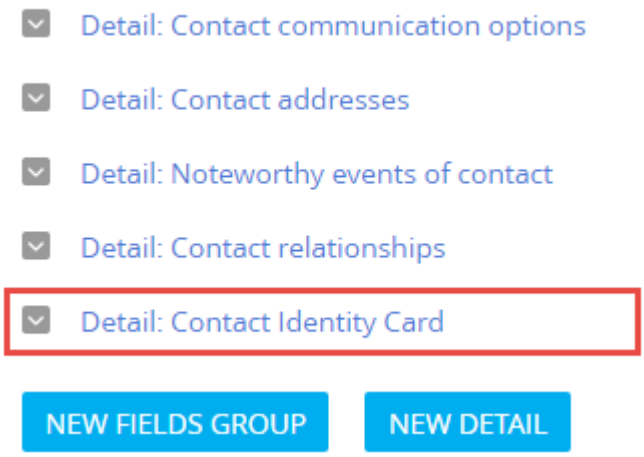
Open the detail wizard in the [Contacts] section, and select [NEW DETAIL] on the [PAGE] step. In the opened window, select the [Contact’s ID] detail and configure the connection between detail object columns and the section object (Fig. 3).

Fig. 3. Detail properties setup



The detail will be displayed in the section record page constructor (Fig. 4).

Fig. 4. A detail in the section record page constructor

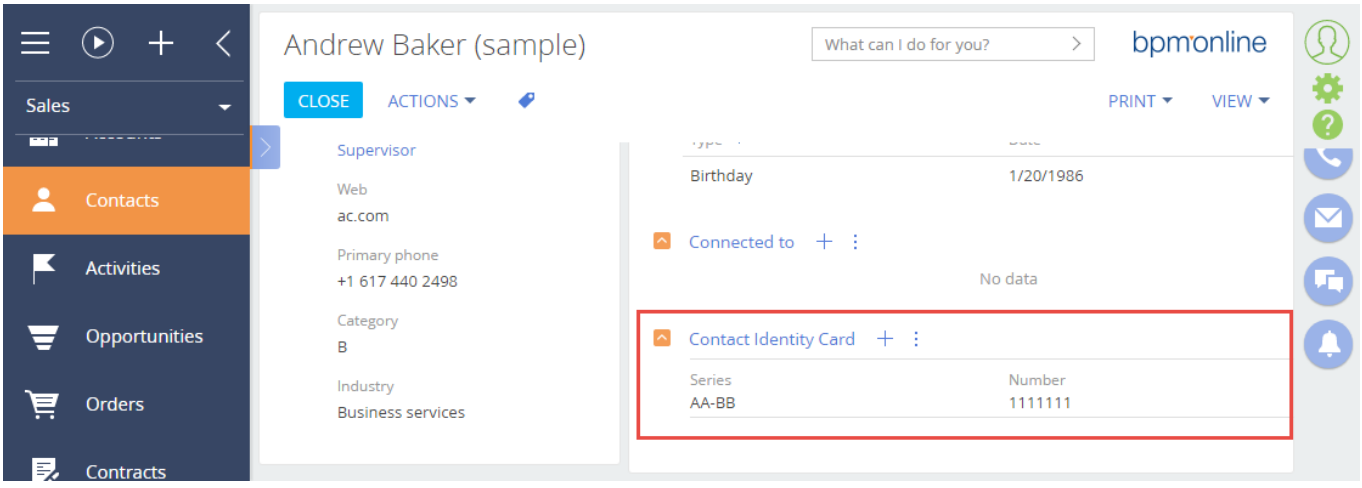


Save the changes when the section record page setup is done.

As a result, the custom package will have a replacing client module of a section page and a schema of the section record edit page.

Upon refreshing, the detail will be displayed on a record edit page.

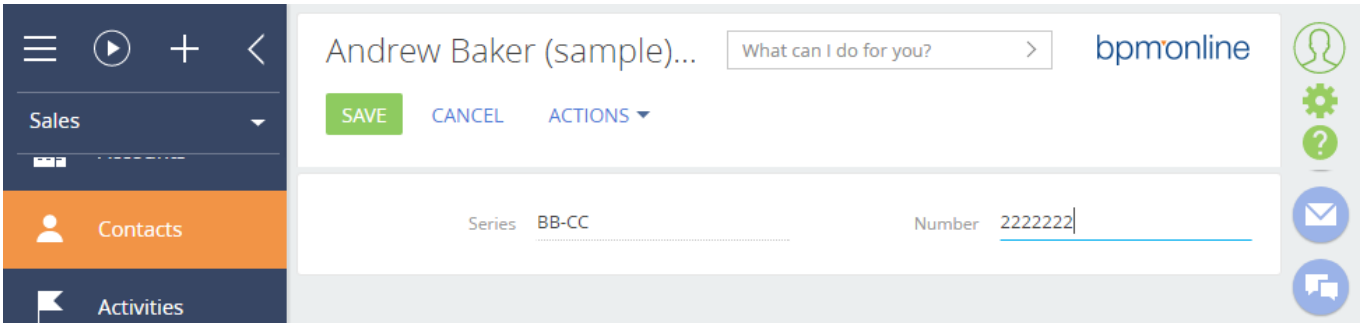
Fig. 5. Case result



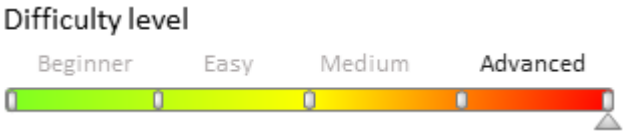
ATTENTION

It is necessary to configure the columns in a detail menu, and add a few records to see the result.

Fig. 6. Adding a record to a detail



Adding the [Attachments] detail



Introduction

The [Attachments] detail is designed for storing files and links to web resources and knowledge base articles related to a section record. The detail is available in all bpm'online sections (see: "Attachments"). The main functionality of the detail is implemented in the *FileDetailV2* schema of the *UIv2* package.

To add a detail to the edit page of a custom section record, do the following:

1. Create a regular detail in the detail wizard using the object schema of the *[Section object name]File* section (see: "**Creating a new section**").

2. Change the parent for the detail list schema.

3. In the wizard, add the detail to the the edit page of a custom section record.

4. Perform additional detail configurations.

Case description

Add the [Attachments] details to the record edit page of the custom [Photos] section. All schemas of the [Photos] section and the [Attachments] details should be stored in a custom package (for example, *sdkDetailAttachment*).

NOTE

To create a section, use the section wizard (see: "Creating a new section" and "Section wizard").

ATTENTION

If the development needs to be carried out in a **custom package**, it needs to be specified in the [Current package] system setting. Otherwise, the wizard will save the changes to the [Custom] package.

Case implementation algorithm

1. Create a detail, using the [Section object name]File section object schema.

As a result (Fig. 1), a number of client modules and object schemas will be created in the custom package. The name

of the schema of the main section object is *UsrPhotos* (Fig. 2). The schema name of the section object that you want to use for the details is *UsrPhotosFile*.

Fig. 1. The [Photos] section properties in the section wizard

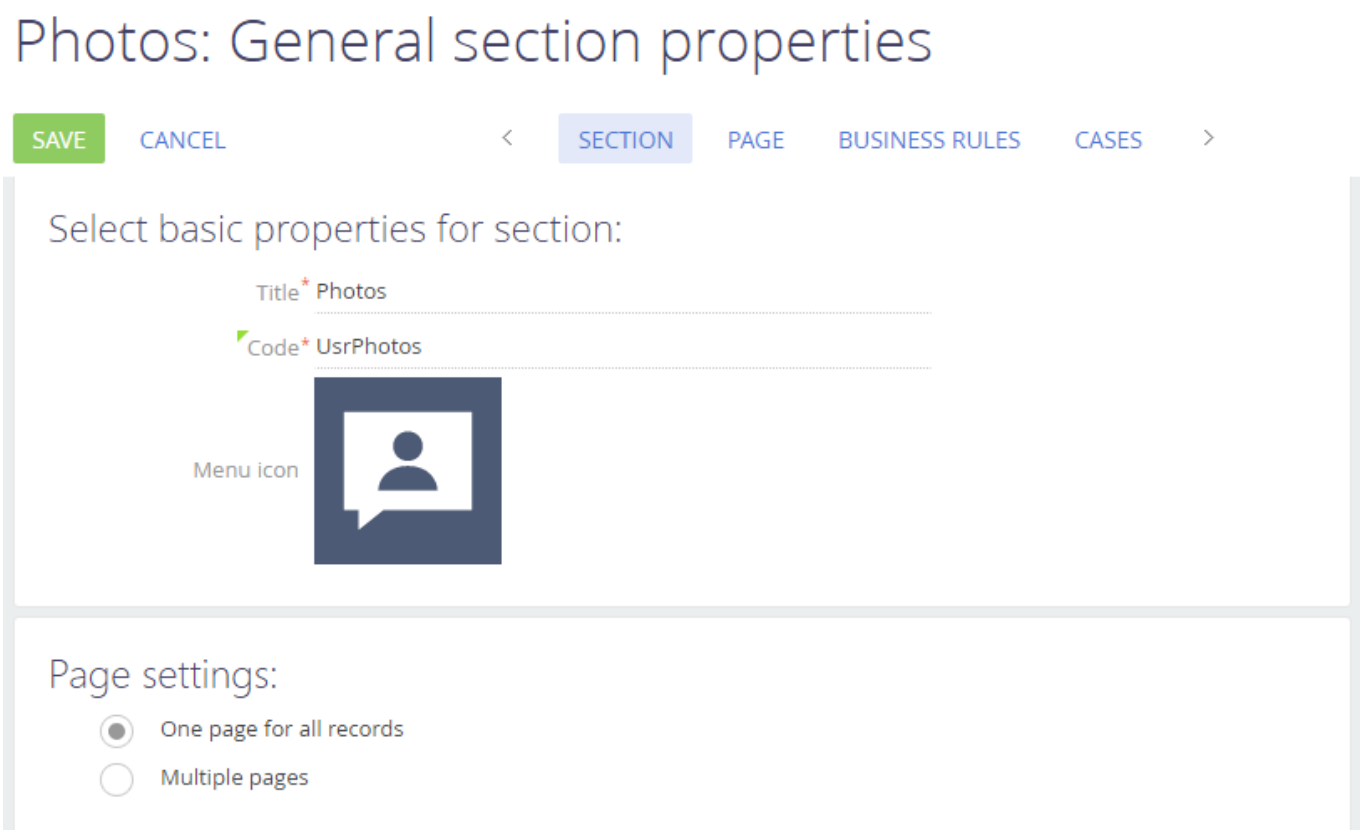
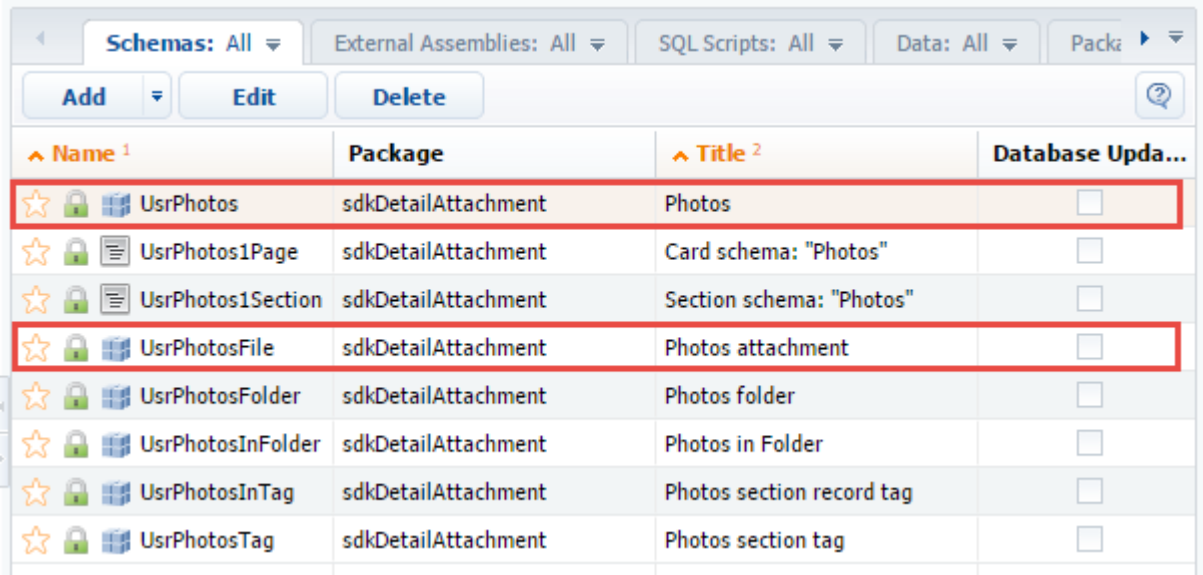
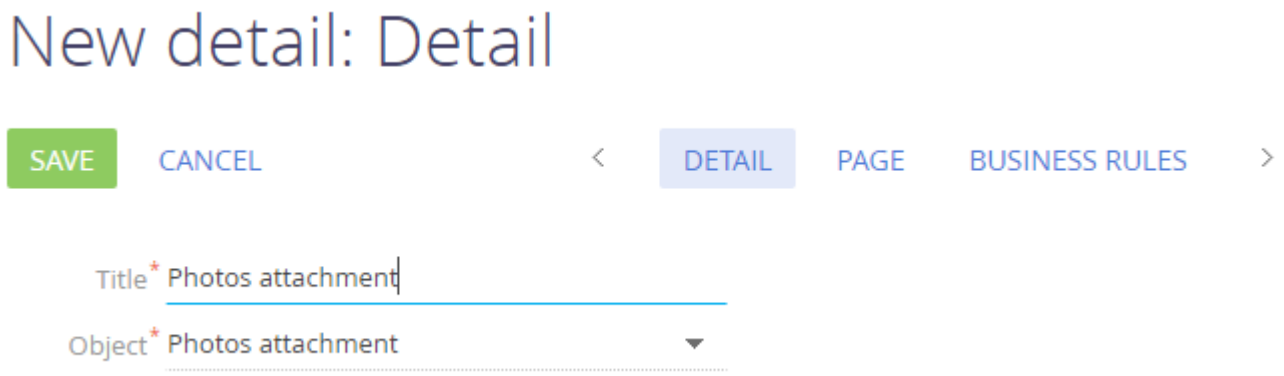


Fig. 2. Schemas created by the section wizard in the user package



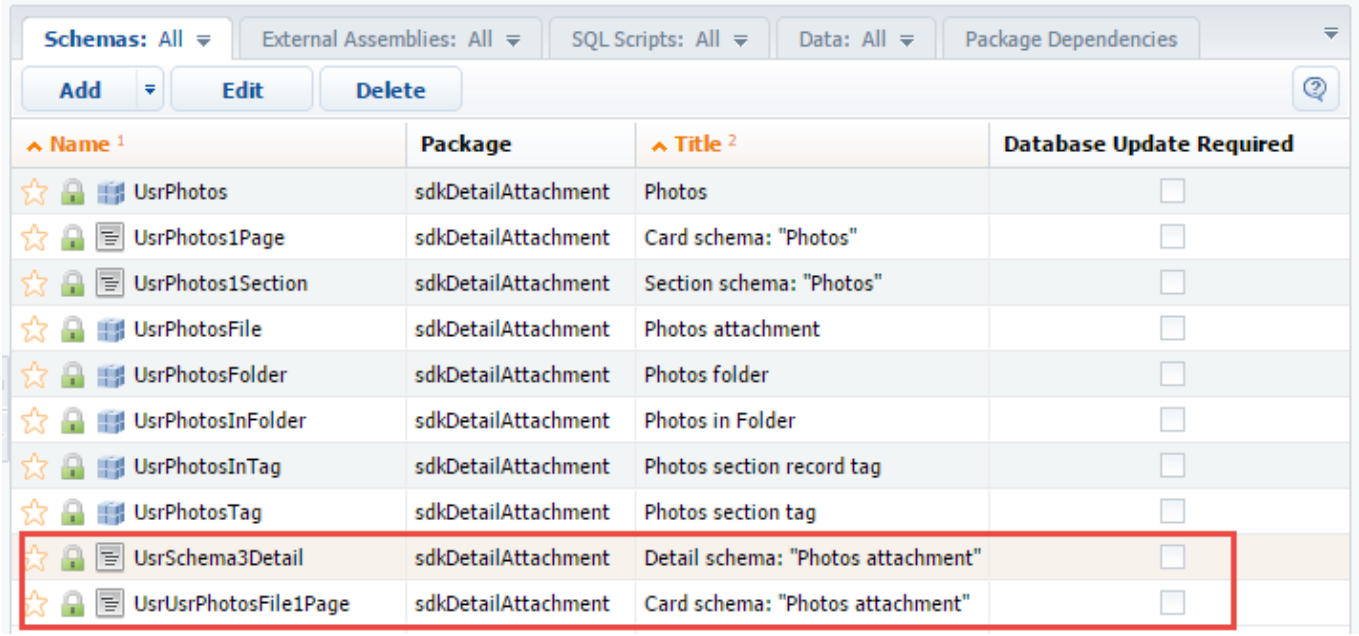
Creating details in a wizard is described in more detail in the "**Creating a detail in wizards**" article. Choose the detail title and the [Photos attachment] object as the default one in the detail wizard (Fig. 3). The transition to the second step in the wizard is optional.

Fig. 3. Detail properties in the wizard



As a result, the custom package will have a schema of the detail list client module and a schema of the detail edit page (Fig. 4).

Fig. 4. Schemas created by the section wizard in the user package



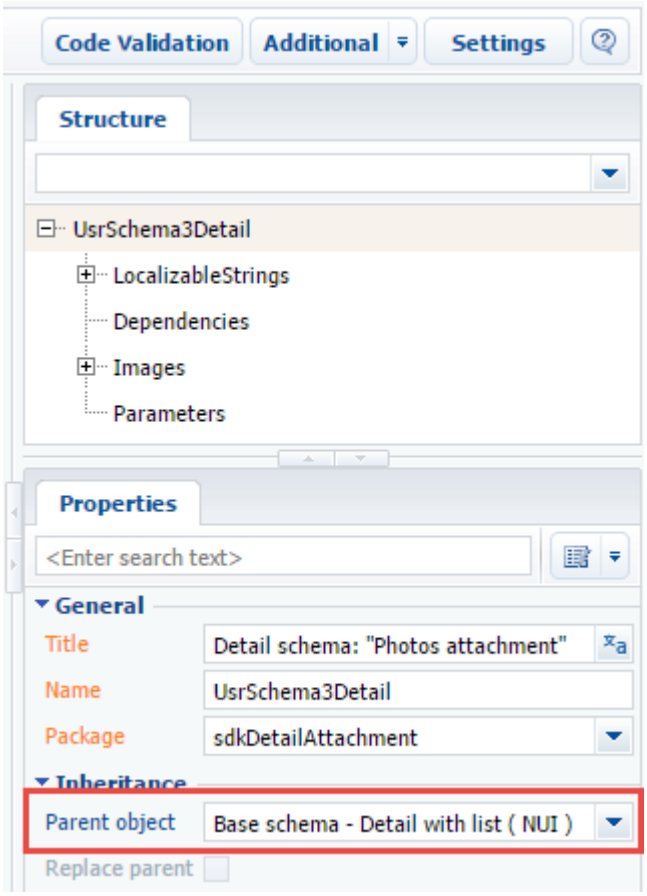
NOTE

Schema names are generated automatically and may be different from those shown on Fig. 4.

2. Change the parent for the detail list schema.

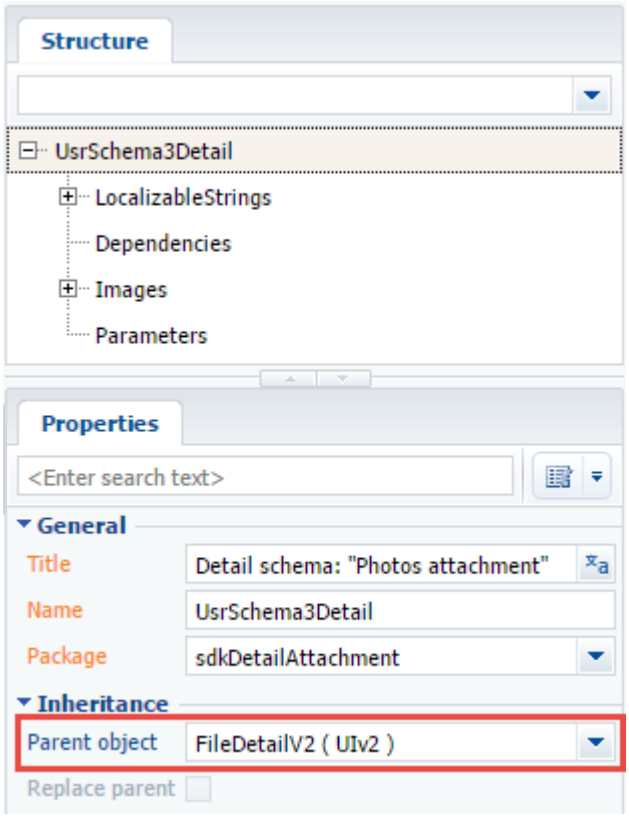
The detail with **an edit page** is created. The [Base schema - Detail with list] schema of the *NUI* package is the parent object of the client module schema of the *UsrSchema3Detail* detail list (Fig. 5).

Fig. 5. A default parent object



To implement the [Files and Links] detail functionality in the custom detail, specify the *FileDetailV2* schema as the parent object of the *UsrSchema3Detail* schema (Fig. 6).

Fig. 6. The parent object – the FileDetailV2 schema



NOTE

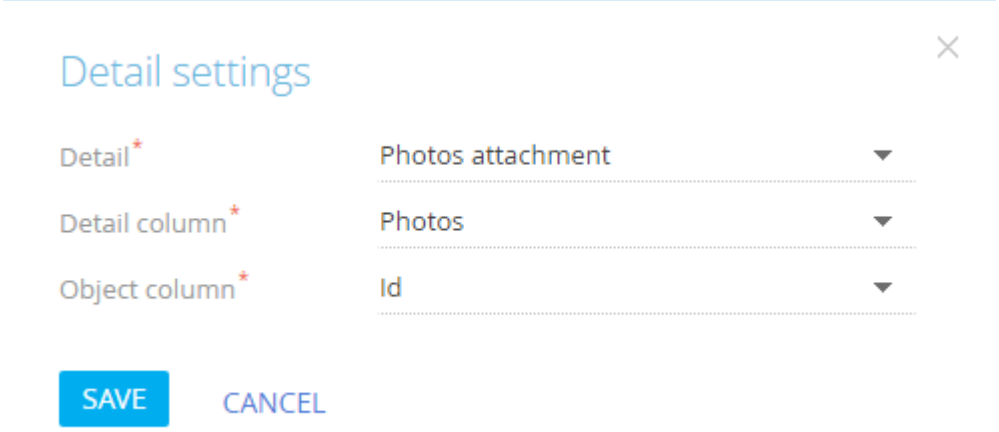
The procedure of specifying a parent object is covered in the "**Creating a custom client module schema**".

Save the schema to apply changes.

3. In the wizard, add the detail to the the edit page of a custom section record.

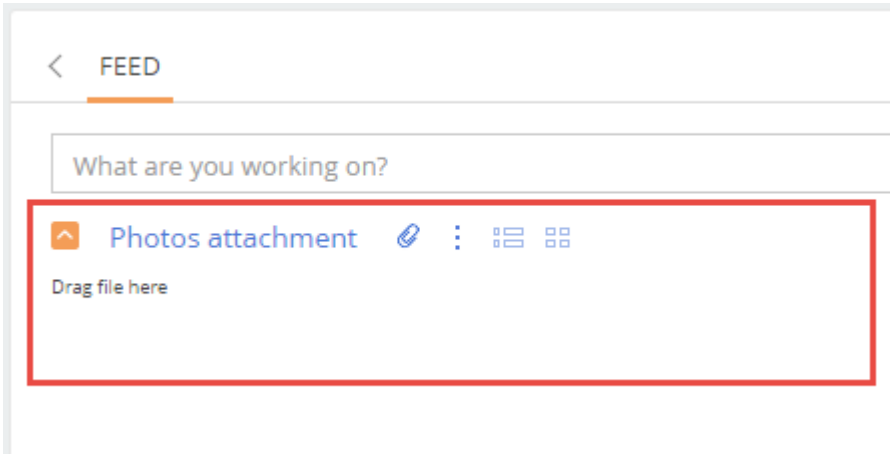
Adding a detail to the record edit page using the detail wizard is described in the "**Creating a detail in wizards**" article. When you add a detail in the wizard, configure the link between the detail columns and the main section object (Fig. 7).

Fig. 7. Configuring the link between the detail columns and the main section object



Upon refreshing, the detail will be displayed on a record edit page (Fig. 8).

Fig. 8. A detail on an edit page



NOTE

You can add details to any edit page tab. Additionally, you can create a separate [Attachments] tab for the detail.

4. Perform additional detail configurations

After completing the previous steps, the detail is fully functional, but looks different from the [Attachments] detail of the base application sections. To make the custom detail look similar to the standard one, you need to define CSS styles for it.

ATTENTION

The client module schema of the *UsrSchema3Detail* detail list is a schema of the view model. Defining styles in

it is impossible. It is necessary to create a new module schema, define the styles and add the created module to module dependencies of a detail.

Create a new module schema in a custom package with the following property values:

- [Title] – “UsrSchema3DetailCSS”.
- [Name] – “UsrSchema3DetailCSS”.
- [Package] – “sdkDetailAttachment”.

Add the following CSS selectors to the LESS tab of the module designer:

```
div[id*="UsrSchema3Detail"] {
    .grid-status-message-empty {
        display: none;
    }
    .grid-empty > .grid-bottom-spinner-space {
        height: 5px;
    }
    .dropzone {
        height: 35px;
        width: 100%;
        border: 1px dashed #999999;
        text-align: center;
        line-height: 35px;
    }
    .dropzone-hover {
        border: 1px dashed #4b7fc7;
    }
    .DragAndDropLabel {
        font-size: 1.8em;
        color: rgb(110, 110, 112);
    }
}

div[data-item-marker*="added-detail"] {
    div[data-item-marker*="tiled"], div[data-item-marker*="listed"] {
        .entity-image-class {
            width: 165px;
        }
        .entity-image-container-class {
            float: right;
            width: 128px;
            height: 128px;
            text-align: center;
            line-height: 128px;
        }
        .entity-image-view-class {
            max-width: 128px;
            max-height: 128px;
            vertical-align: middle;
        }
        .images-list-class {
            min-height: 0.5em;
        }
        .images-list-class > .selectable {
            margin-right: 10px;
            display: inline-block;
        }
        .entity-label {
            display: block;
            max-width: 128px;
            margin-bottom: 10px;
```

```
            text-align: center;
        }
        .entity-link-container-class > a {
            font-size: 1.4em;
            line-height: 1.5em;
            display: block;
            max-width: 128px;
            margin-bottom: 10px;
            color: #444;
            text-decoration: none;
            text-overflow: ellipsis;
            overflow: hidden;
            white-space: nowrap;
        }
        .entity-link-container-class > a:hover {
            color: #0e84cf;
        }
        .entity-link-container-class {
            float: right;
            width: 128px;
            text-align: center;
        }
        .select-entity-container-class {
            float: left;
            width: 2em;
        }
        .listed-mode-button {
            border-top-right-radius: 1px;
            border-bottom-right-radius: 1px;
        }
        .tiled-mode-button {
            border-top-left-radius: 1px;
            border-bottom-left-radius: 1px;
        }
        .tiled-mode-button, .listed-mode-button {
            padding-left: 0.308em;
            padding-right: 0.462em;
        }
    }
    .button-pressed {
        background: #fff;

        .t-btn-image {
            background-position: 0 16px !important;
        }
    }
    div[data-item-marker*="tiled"] {
        .tiled-mode-button {
            .button-pressed;
        }
    }
    div[data-item-marker*="listed"] {
        .listed-mode-button {
            .button-pressed;
        }
    }
}
```

NOTE

The styles defined in the source code above almost completely coincide with the styles defined in the schema of the *FileDetailCssModule* module in the *UIv2* package. They are intended for the *FileDetailV2* schema used

as the parent object.

You can not use the *FileDetailCssModule* module directly, because the markers and identifiers of the HTML elements of the standard detail and the detail created by the wizard are different.

Save the schema to apply changes.

To use the created module and its styles in a detail *UsrSchema3Detail* detail schema, add it to the dependency of the module defined in the detail schema.

The source code of the modified schema:

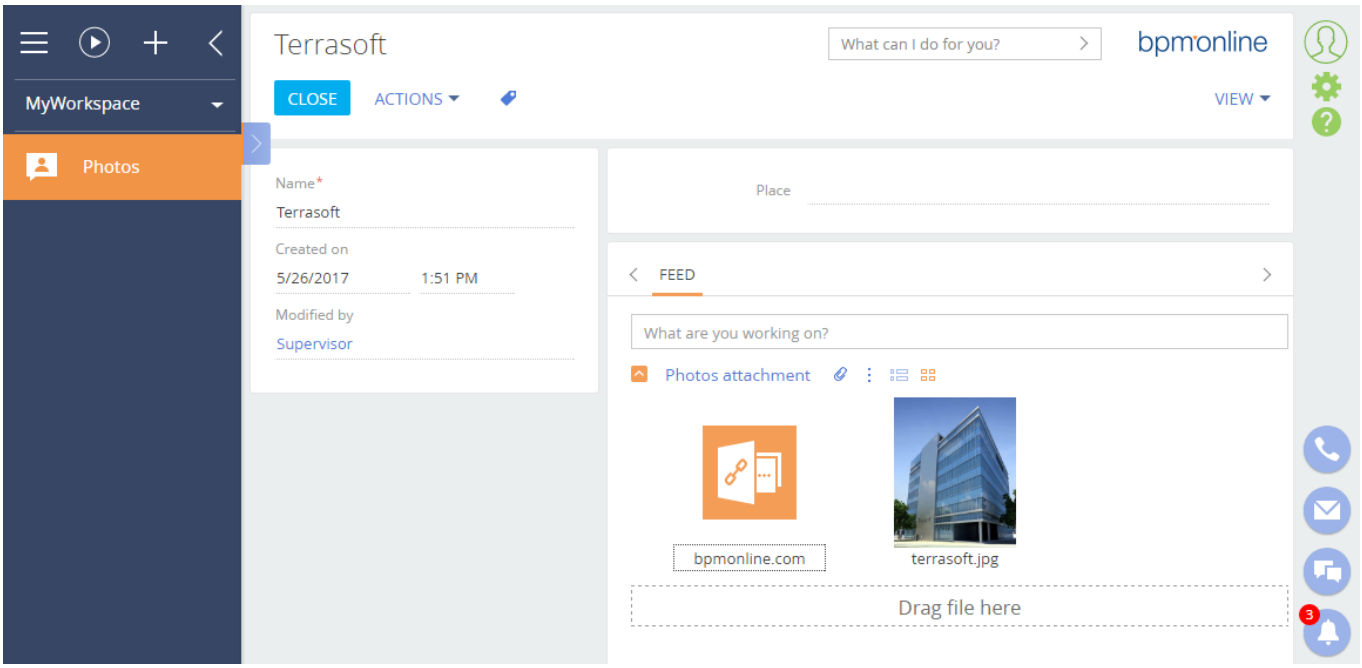
```
div[id*="UsrSchema3Detail"] {
    .grid-status-message-empty {
        display: none;
    }
    .grid-empty > .grid-bottom-spinner-space {
        height: 5px;
    }
    .dropzone {
        height: 35px;
        width: 100%;
        border: 1px dashed #999999;
        text-align: center;
        line-height: 35px;
    }
    .dropzone-hover {
        border: 1px dashed #4b7fc7;
    }
    .DragAndDropLabel {
        font-size: 1.8em;
        color: rgb(110, 110, 112);
    }
}

div[data-item-marker*="added-detail"] {
    div[data-item-marker*="tiled"], div[data-item-marker*="listed"] {
        .entity-image-class {
            width: 165px;
        }
        .entity-image-container-class {
            float: right;
            width: 128px;
            height: 128px;
            text-align: center;
            line-height: 128px;
        }
        .entity-image-view-class {
            max-width: 128px;
            max-height: 128px;
            vertical-align: middle;
        }
        .images-list-class {
            min-height: 0.5em;
        }
        .images-list-class > .selectable {
            margin-right: 10px;
            display: inline-block;
        }
        .entity-label {
            display: block;
            max-width: 128px;
            margin-bottom: 10px;
```

```
            text-align: center;
        }
        .entity-link-container-class > a {
            font-size: 1.4em;
            line-height: 1.5em;
            display: block;
            max-width: 128px;
            margin-bottom: 10px;
            color: #444;
            text-decoration: none;
            text-overflow: ellipsis;
            overflow: hidden;
            white-space: nowrap;
        }
        .entity-link-container-class > a:hover {
            color: #0e84cf;
        }
        .entity-link-container-class {
            float: right;
            width: 128px;
            text-align: center;
        }
        .select-entity-container-class {
            float: left;
            width: 2em;
        }
        .listed-mode-button {
            border-top-right-radius: 1px;
            border-bottom-right-radius: 1px;
        }
        .tiled-mode-button {
            border-top-left-radius: 1px;
            border-bottom-left-radius: 1px;
        }
        .tiled-mode-button, .listed-mode-button {
            padding-left: 0.308em;
            padding-right: 0.462em;
        }
    }
    .button-pressed {
        background: #fff;

        .t-btn-image {
            background-position: 0 16px !important;
        }
    }
    div[data-item-marker*="tiled"] {
        .tiled-mode-button {
            .button-pressed;
        }
    }
    div[data-item-marker*="listed"] {
        .listed-mode-button {
            .button-pressed;
        }
    }
}
```

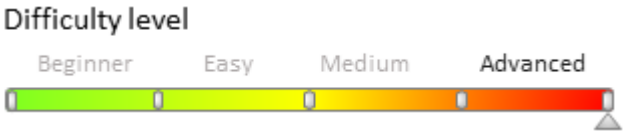
Save the schema to apply changes.

As a result, the edit page of the custom [Photos] section will display an [Attachments] detail, almost identical to the base one (Fig. 9).

Fig. 9. Case result



Displaying additional columns on the [Attachments] tab

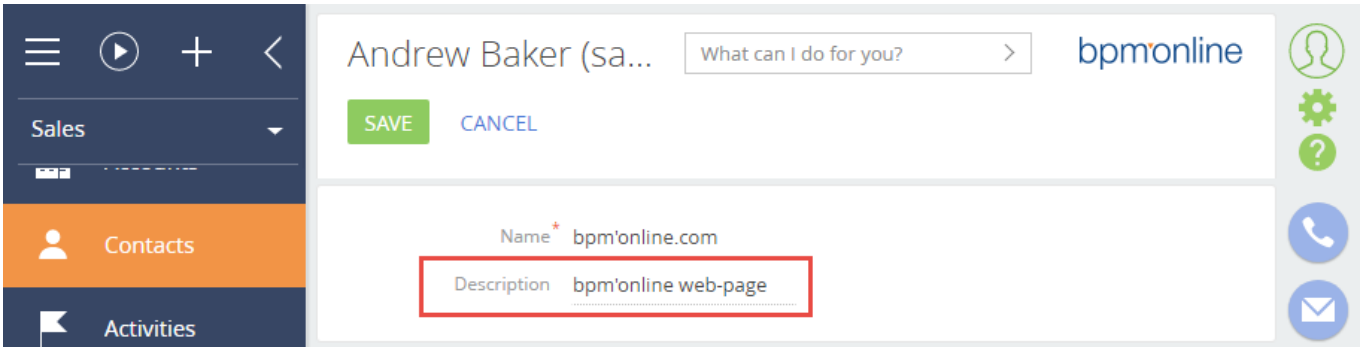


Introduction

The [Attachments] detail is designed for storing files and links to web resources and knowledge base articles related to the section record. The detail is available in all bpm’online section (see: “How to work with attachments and notes”). The main functionality of the detail is implemented in the *FileDetailV2* scheme of the *UIv2* package.

By default, the [Attachments] detail is set to have only the [Name] and [Version] columns in the list view. The [Description] column is available while adding a new link. However, it is not displayed in the list view.

Fig. 1. The [Description] field



NOTE

The tile view of the [Attachments] detail only features the [Name] column and a file or a link.

Use the columns setup page to set up the detail’s list columns (see: “Setting up columns”). However, the [Attachments] detail does not have this configuration option by default.

To add this configuration option, do the following:

- Create a replacing schema of the *FileDetailV2* detail.
- Call the method of opening the column configuration page in the replacing schema.

Case description

Add a new [Columns setup] command to the [Actions] menu for the [Attachments] detail.

Case implementation algorithm

1. Replace the FileDetailV2 detail schema

The procedure for creating a replacing client schema is covered in the “**Creating a custom client module schema**”. Create a replacement schema with the following properties in a custom package:

- [Title]— “FileDetailV2”.
- [Name] — “FileDetailV2”.
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down list.
- [Parent object] – “FileDetailV2”.

2. Call the method of opening the column configuration page in the replacing schema.

To do this, go to the the **module description** in the source code of the replacement schema, and redefine the base *getGridSettingsMenuItem()* method, which returns the detail menu item, associated with the call to the base *openGridSettings()* method defined in the *GridUtilities* mixin.

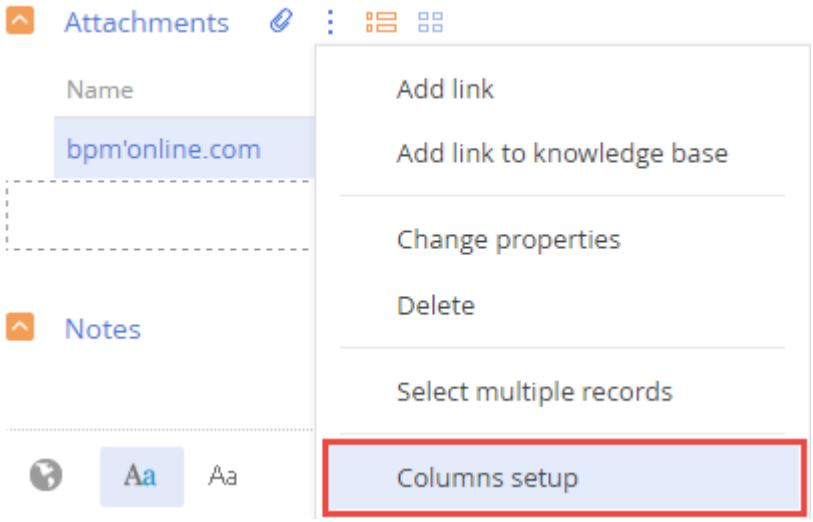
Schema source code:

```
define("FileDetailV2", [], function() {
    return {
        methods: {
            getGridSettingsMenuItem: function() {
                return this.getButtonMenuItem({
                    Caption: {"bindTo": "Resources.Strings.SetupGridMenuCaption"},
                    Click: {"bindTo": "openGridSettings"}
                });
            }
        }
    };
});
```

Save the schema to apply changes.

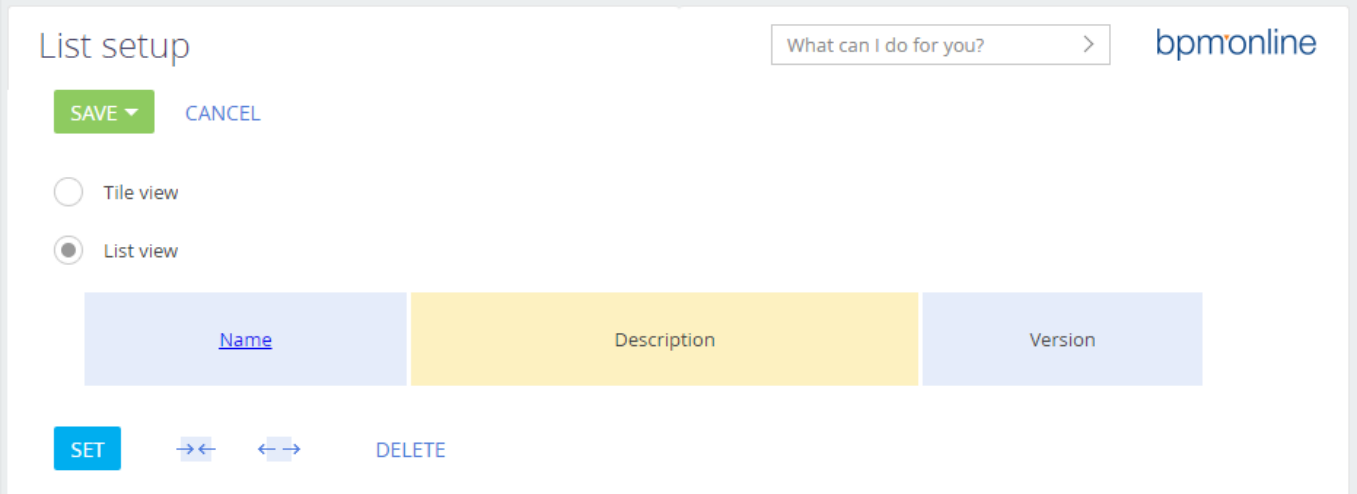
As a result, a new [Columns setup] command is displayed in the [Actions] menu (Fig. 2).

Fig. 2. The [Columns setup] command



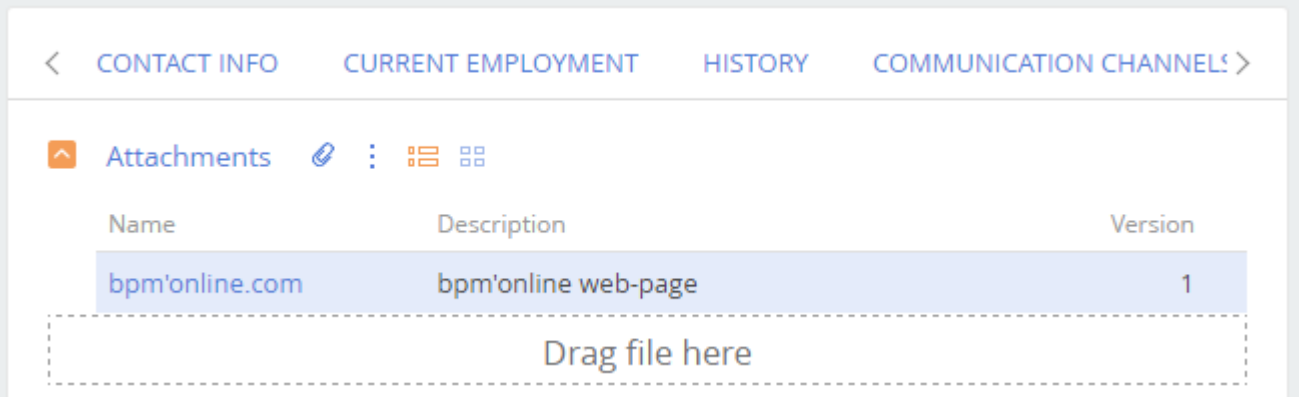
This command enables the user to configure the list column view, and add the [Description] column to the detail.

Fig. 3. The column configuration page

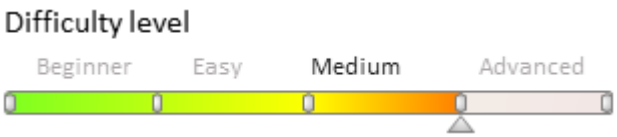


After saving the settings in the detail list view, the column will be displayed (Fig. 4).

Fig. 4. Case result

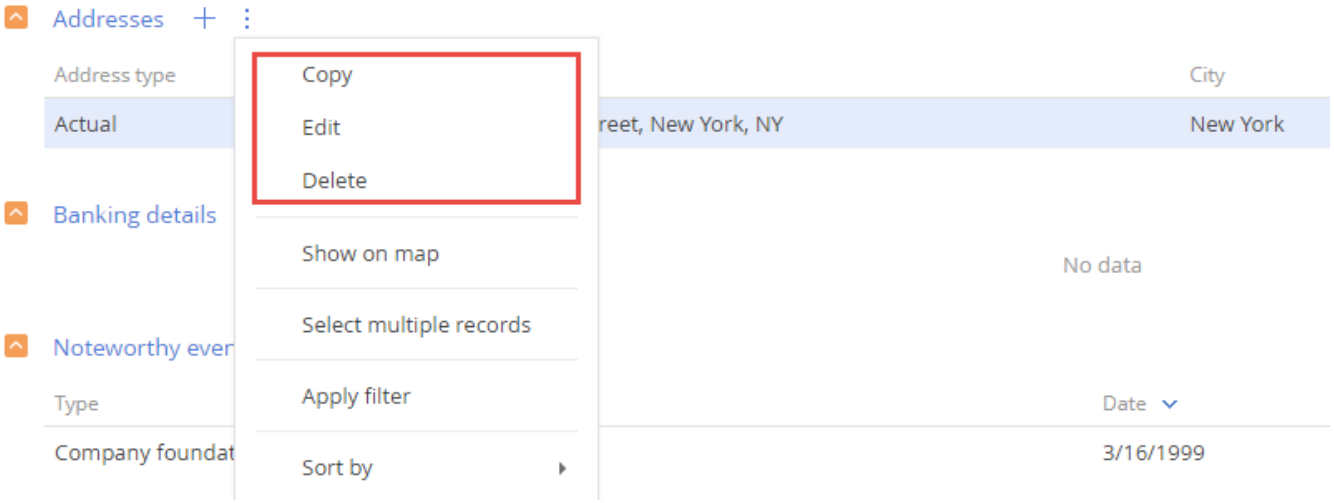


How to hide menu commands of the detail with list



The [Copy], [Edit], and [Delete] commands in a detail menu are used to manage records in the detail list (Fig. 1).

Fig. 1. The [Addresses] detail menu



To hide menu detail commands:

1. Create a replacing schema of the detail list. For example, for the [Addresses] detail of the account edit page it will be the [Account addresses detail]. The procedure for creating a replacing client schema is covered in the “**Creating a custom client module schema**” article.

2. Add the following source code to the schema:

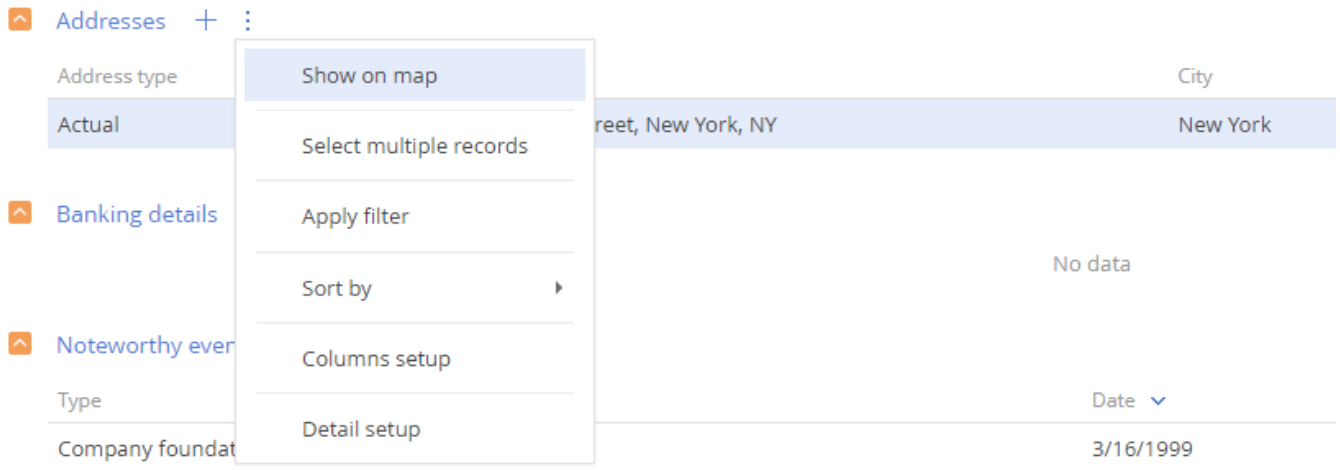
```
define("AccountAddressDetailV2", [], function() {
    return {
        entitySchemaName: "AccountAddress",
        methods: {
            // Disabling the [Copy] command
            getCopyRecordMenuItem: Terrasoft.emptyFn,
            // Disabling the [Edit] command
            getEditRecordMenuItem: Terrasoft.emptyFn,
            // Disabling the [Delete] command
            getDeleteRecordMenuItem: Terrasoft.emptyFn
        },
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    };
});
```

3. Save the changes.

4. Refresh the browser page.

As a result, commands will be disabled in the detail menu (Fig.2).

Fig. 2. [Addresses] detail menu without menu commands

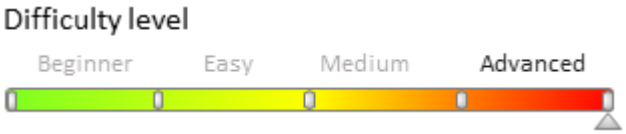


Business processes

Contents

- **How to add auto-numbering to the edit page field**
- **Process launch from a client module**
- **Creating custom [User task] process element**
- **How to customize notifications for the [User task] process element**
- **How to run bpm'online processes via web service**
- **How to save the record without closing the edit page which is opened by the business process**

How to add auto-numbering to the edit page field



Introduction

You can add auto numbering for an object column. For instance, auto numbering is applied in the [Documents], [Invoices] and [Contracts] sections where a preformatted number is automatically generated when you add a new record.

There are two ways of implementing auto numbering:

- Client-side implementation.
- Server-side implementation.

To implement auto numbering ***on the client side***, override the base virtual method *onEntityInitialized()* and call the *getIncrementCode()* method of the edit page base schema *BasePageV2*.

The *getIncrementCode()* method accepts two parameters:

- *callback* – the function that will run after receiving service response. The response must be passed to the corresponding column (attribute);
- *scope* – the context where the *callback* function will be run (optional parameter).

To implement auto numbering ***on the server side*** , add the event handler [Before record adding] to the object, whose column will be auto numbered. Set up number generating parameters in the business process, namely:

- Indicate the schema of the object for which generation will be performed.
- Call the [Generate ordinal number] action.
- Pass the generated value to the necessary object column.

NOTE

This is not the only way to implement auto numbering on the server side. It can be implemented via custom means, for instance, by creating a **custom service**.

Regardless of the chosen solution, add two system settings to use auto numbering:

- *[Entity]CodeMask* – object number mask.
- *[Entity]LastNumber* – current object number.

[Entity] – is the name of the object, whose column will be auto numbered. For example, *InvoiceCodeMask* (Invoice number mask) and *InvoiceLastNumber* (Current invoice number).

Case description

Set up auto numbering for the [Code] field in the [Products] section. The product code format must be as follows: ART_00001, ART_00002 and so on.

ATTENTION

We covered two alternative ways of case implementation: client- and server-side.

Source code

You can download the package with case implementation using the following link.

ATTENTION

The package does not contain the bound system settings *ProductCodeMask* and *ProductLastNumber*. You will need to add them manually.

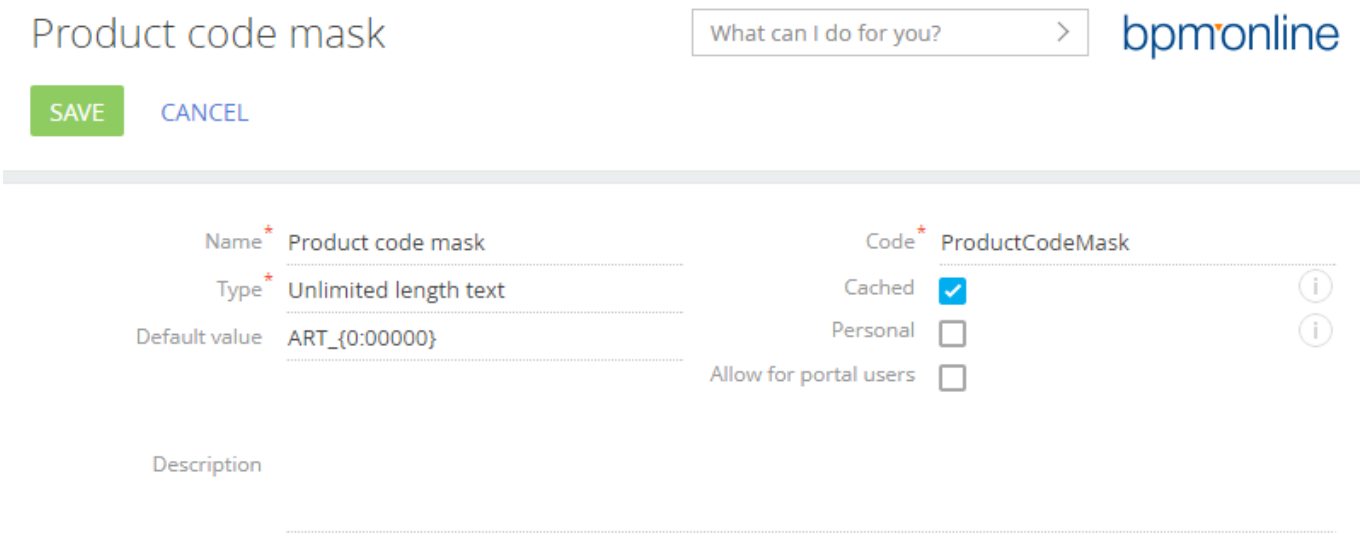
Case implementation algorithm: client-side

1. Create two system settings

Create the [Product code mask] system setting with the following number mask: “ART_{0:00000}” (Fig.1) Populate the following fields:

- [Name] – “Product code mask”.
- [Code] – “ProductCodeMask”.
- [Type] – a string, whose length depends on the number of characters in the mask. In most cases 50 characters are enough. In this example, we use a string of unlimited length.
- [Default value] – "ART_{0:00000}".

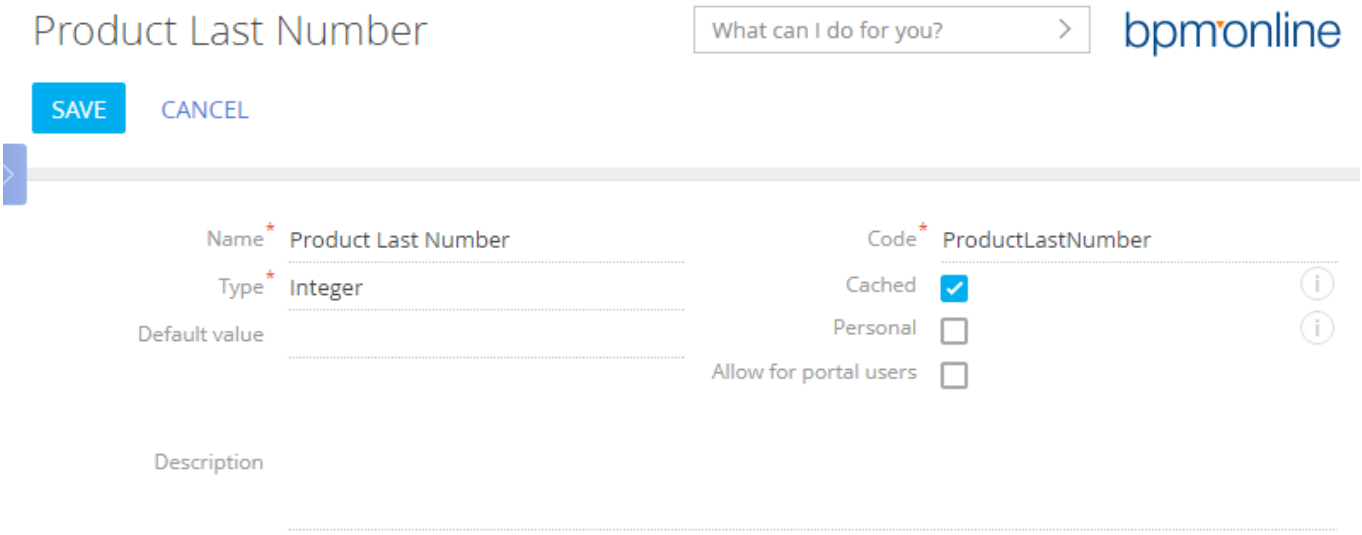
Fig. 1. The [Product code mask] system setting



Create the [Product last number] system setting (Fig.2). Populate its properties:

- [Name] – “Product last number”.
- [Code] – “ProductLastNumber”.
- [Type] – “Integer”.

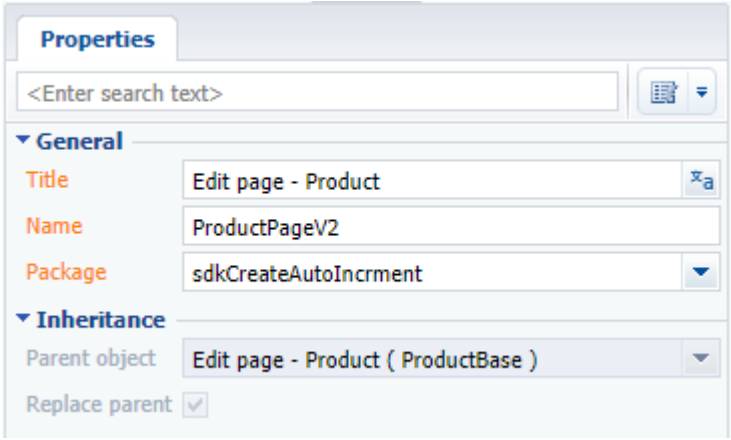
Fig. 2. The [Product last number] system setting



2. Create a replacing schema in the custom package

Create a replacing client module and specify the *ProductPageV2* schema as parent object (Fig. 3). The procedure for creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 3. Properties of the product edit page replacing schema



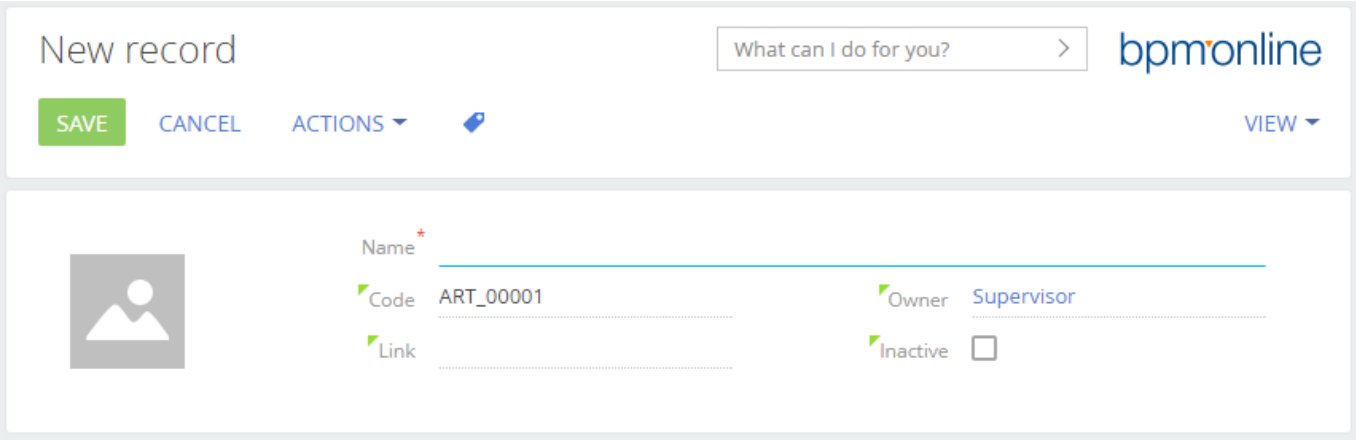
3. Override the onEntityInitialized() method

In the collection of edit page view model methods, override the *onEntityInitialized()* method. In *onEntityInitialized()* method, call the *getIncrementCode()* method and fill in the generated number in the [Code] column of its callback function. The replacing schema source code is as follows:

```
define("ProductPageV2", [], function() {
    return {
        // The name of edit page object schema.
        entitySchemaName: "Product",
        //  The collection of edit page view model methods.
        methods: {
            // Overriding Terrasoft.BasePageV2.onEntityInitialized base method, that
            // is triggered when the initialization of the edit page object schema is
finished.
            onEntityInitialized: function() {
                // onEntityInitialized method parent realization is called.
                this.callParent(arguments);
                // The code is generated only in case we create a new element or a
copy of the existing element.
                if (this.isAddMode() || this.isCopyMode()) {
                    //  Call of the Terrasoft.BasePageV2.getIncrementCode base
method, that generates the number
                    // according to the previously set mask.
                    this.getIncrementCode(function(response) {
                        // The generated number is stored in [Code] column.
                        this.set("Code", response);
                    });
                }
            }
        }
    };
});
```

After saving the schema, clearing the browser cache and updating the application page, the automatically generated product code will be displayed when you add a new product (Fig.4).

Fig. 4. The result of case implementation on the client side



Case implementation algorithm: server-side

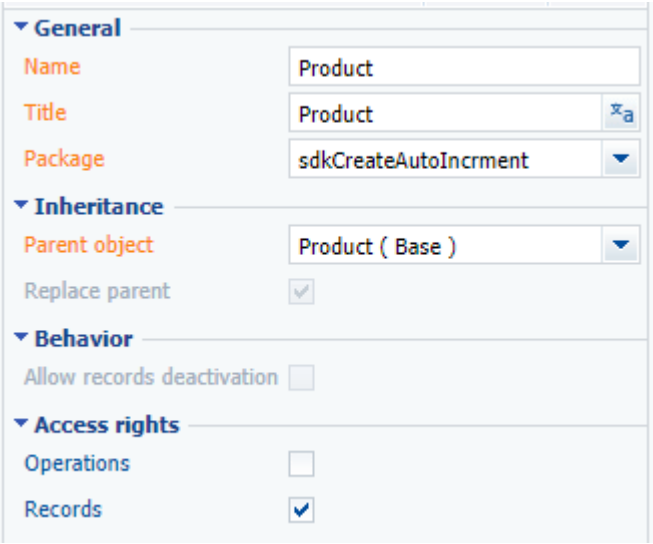
1. Create two system settings

This step is absolutely identical to the first step of case implementation algorithm on the client side.

2. Create a replacing schema of the [Product] object

Select a custom package and execute the [Add] – [Replacing object] menu command on the [Schemas] tab. Specify the [Product] object as the parent object in the new object properties (Fig. 5).

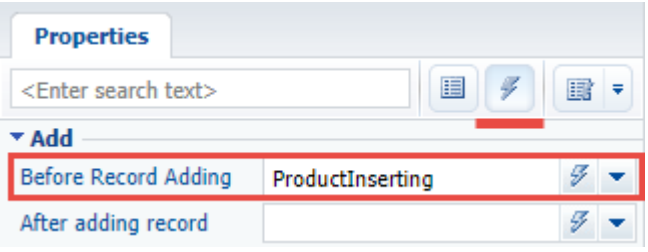
Fig. 5. Properties of the product replacing schema



3. Add the [Before record adding] event handler to the object schema

Add a new event handler in the object properties displayed in the **object designer**. To do this, go to the event tab and double-click the [Before Record Adding] field or click the event icon in this field (Fig.6).

Fig. 6. The [Before record adding] product event handler

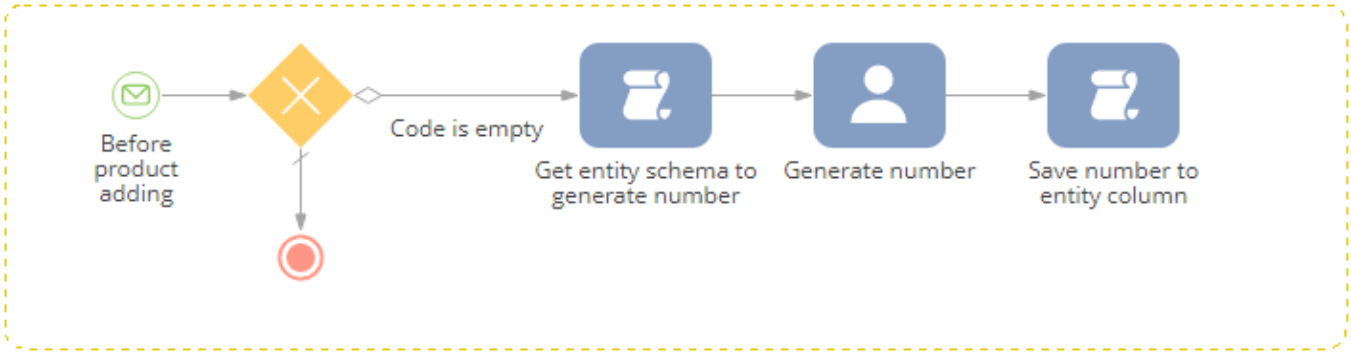


The object's **process designer** will open.

4. Add an event sub-process

To implement the [Before Record Adding] event handler, add event sub-process to the working area of the object process designer. Set up a business process for number generation (Fig.7).

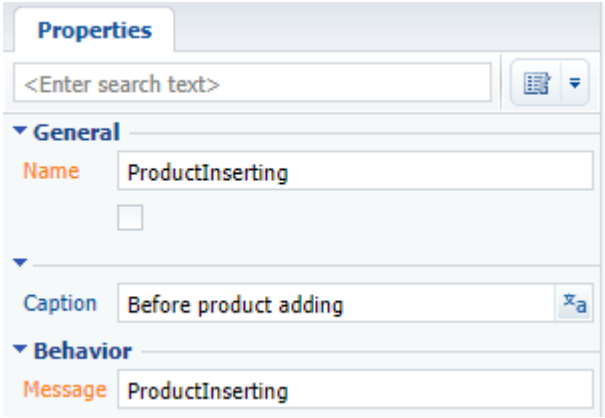
Fig. 7. The sub-process for the [Before Record Adding] event handler



Event sub-process elements

1. Initial message [Before product adding] (Fig.8) – the sub-process will be run upon receiving the *ProductInserting* message added at step3.

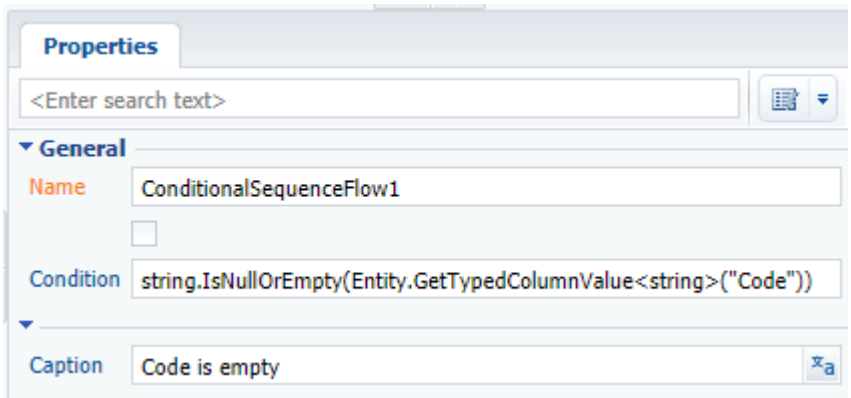
Fig. 8. The [Before product adding] initial message properties



2. [Exclusive gateway (OR)], which branches the process into two flows:

- Default flow – the transition down this flow will occur if the condition flow cannot be implemented. This branch finishes with the [Terminate] event.
- Condition flow [Code is empty] – checks whether the [Code] column is populated (Fig.9). The further execution of the sub-process can only be possible if the column is not populated.

Fig. 9. The [Code is empty] condition flow properties

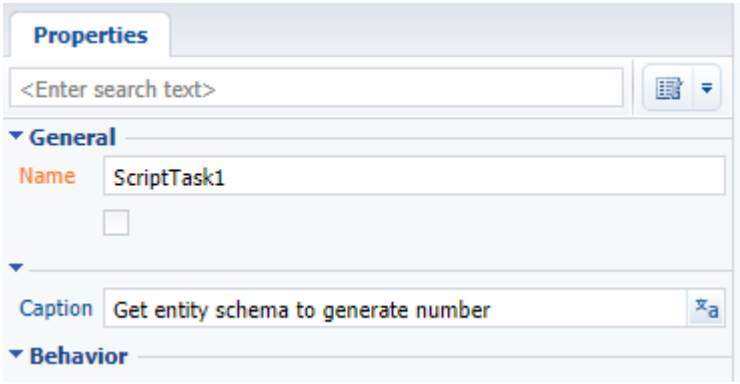


Add the following code to the [Condition] field of the condition flow:

```
string.IsNullOrEmpty(Entity.GetTypedColumnValue<string>("Code"))
```

3. Script task [Get entity schema to generate number] (Fig.10).

Fig. 10. [Get entity schema to generate number] script task properties



Program code C# script is executed in this element. To add it double-click the element. Add the following source code in the opened window:

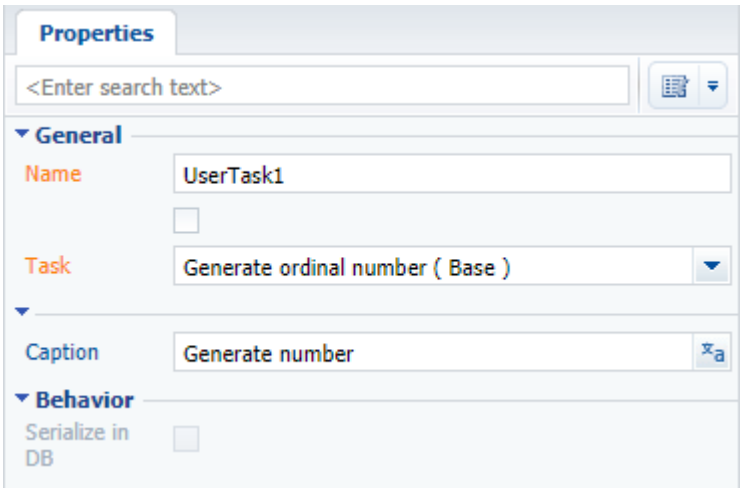
```
//Setting the schema for number generation.
UserTask1.EntitySchema = Entity.Schema;
return true;
```

Note that "UserTask1" here is the name of the [Generate number] user action.

> ATTENTION
>
> Save the script after adding the source code. To do this, select the [Save] menu action.

4. User task [Generate number] (Fig.11).

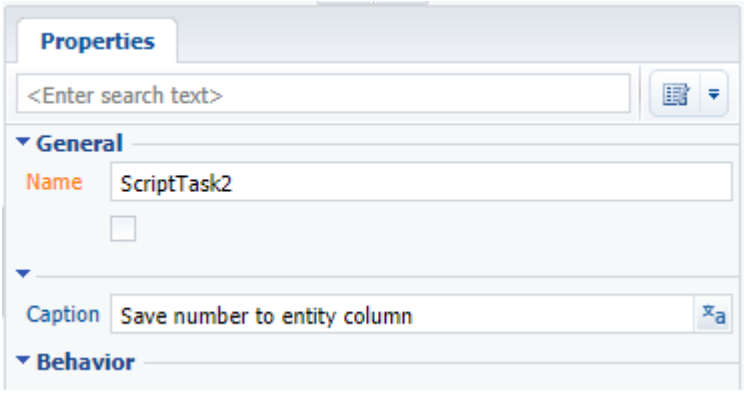
Fig. 11. The [Generate number] user task properties



This element performs the [Generate ordinal number] system action. It is the [Generate ordinal number] system action, which generates the current ordinal number in accordance with the *ProductCodeMask* mask set in the system settings.

5. Script task [Save number to entity column] (Fig.12).

Fig. 12. The [Save number to entity column] script task properties



Program code C# script is executed in this element. The value generated by the UserTask1 user action is stored in the [Code] column of the [Product] created object. The source code of the schema is available below:

```
Entity.SetColumnValue("Code", UserTask1.ResultCode);
return true;
```

Save and close the default process designer and publish the [Product] object schema. As a result, after saving the new product, the [Code] field will be automatically populated on the product page (Fig.13, Fig.14).

> ATTENTION
>
> Since code auto generation and saving in the column is performed on the server side when the [Before saving record] event occurs, it is impossible to view the code value on the product page immediately. This is because the [Before saving record] event occurs on the server side after sending the request to add a record from the application client part.

Fig. 13. The code is not displayed when creating a product

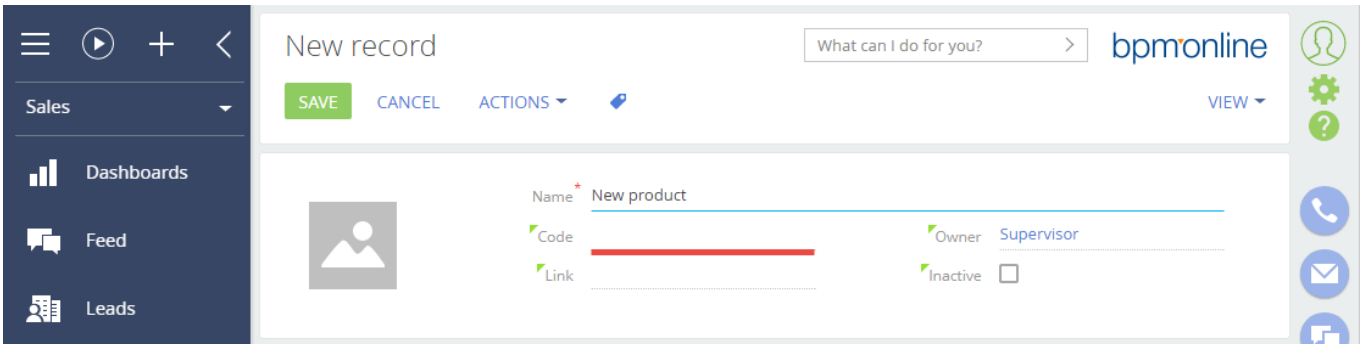
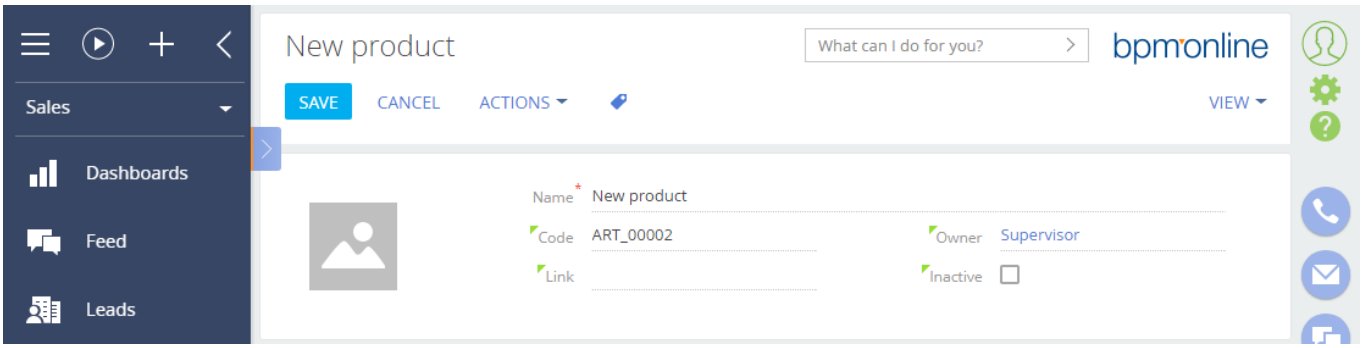
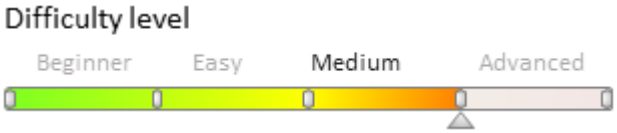


Fig. 14. The code is displayed in the saved product



Process launch from a client module



Introduction

To launch a process from the JavaScript code client schema:

1. Add the *ProcessModuleUtilities* module as a dependency to the module of the page that was used for calling the service. This module provides a convenient interface for executing queries to the ProcessEngineService.svc sevice.
2. Call the *executeProcess(args)* method of the *ProcessModuleUtilities* module by passing the *args* object over to it as a parameter with the following properties (table 1):

Table 1. Properties of the args object

Property	Details
sysProcessName	The name of the called process (not required if the *sysProcessId* property is defined).
sysProcessId	The unique identifier of the called process (not required if the *sysProcessName* property is defined).
parameters	The object whose properties are the same as the properties of the called process incoming parameters.

Case description

Add an action that will launch the "Conducting a meeting" business process to the account edit page. Pass the primary contact of the account as a parameter to the business process.

Source code

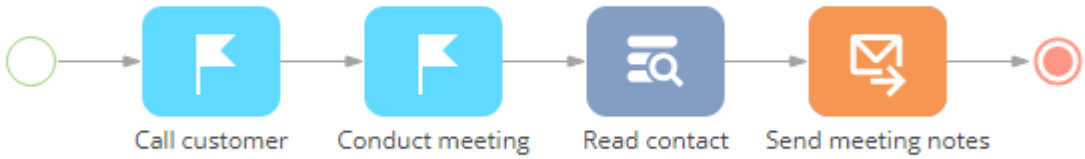
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Creating the "Conducting a meeting" custom business process

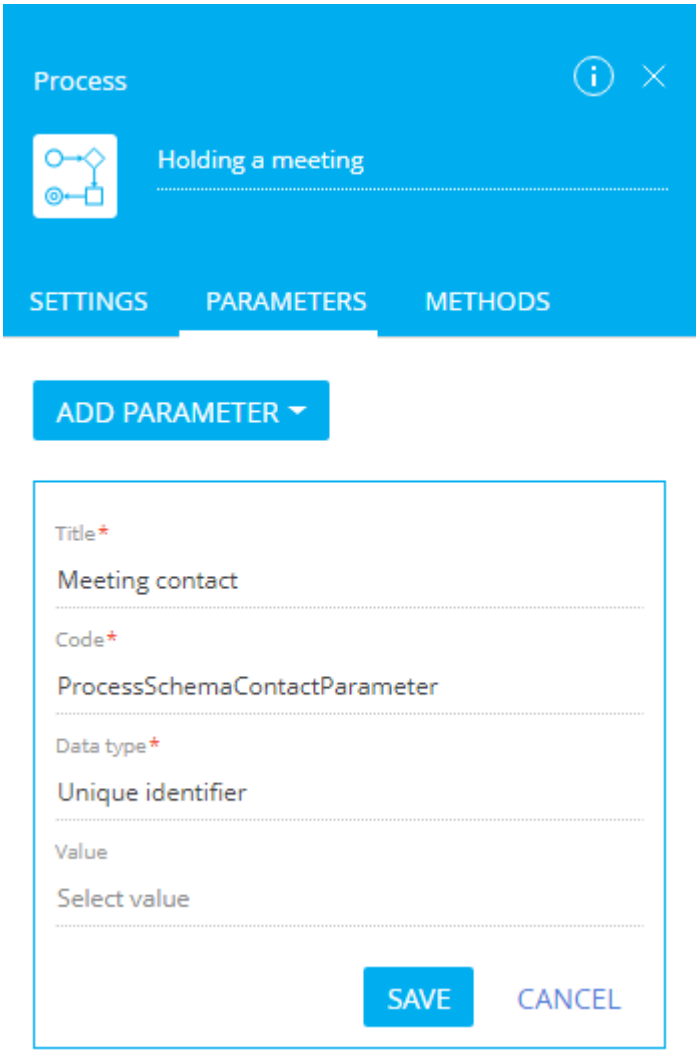
The case uses the "Conduct a meeting" business process described in the "Designing a linear process" and "How to work with emails" sections (fig. 1).

Fig. 1. Creating the "Conducting a meeting" source business process



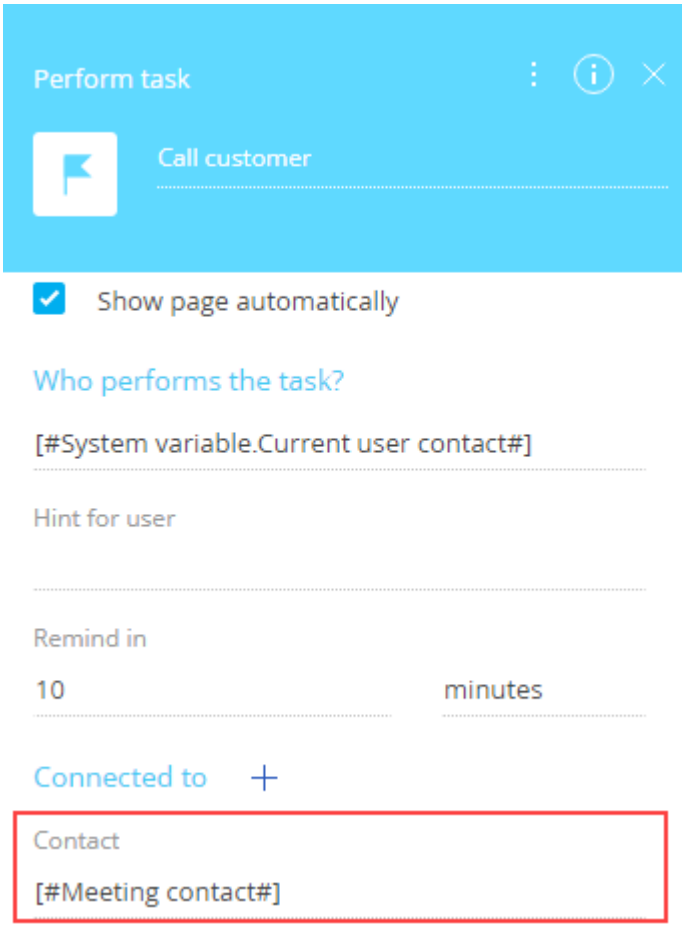
After you create the business process, add the *ProcessSchemaContactParameter* incoming parameter to it. Specify "Unique identifier" in the [Data type] field of the parameter properties (fig. 2).

Fig. 2. Adding the incoming parameter



In the properties of the [Call customer] action (which is the first business process action), populate the [Contact] field with the business process incoming parameter (fig. 3).

Fig. 3. Passing the parameter to the process element



2. Creating the replacing edit page of the account and adding the action.

Adding action to the edit page is covered in the "**Adding an action to the edit page**" article.

Add the *CallProcessCaption* localizable string with an action caption (for example, "Schedule a meeting") to the replacing module schema of the edit page and the account section schema.

Add the *ProcessModuleUtilities* module as a dependency to declaring the edit page module.

The source codes of the section schema and the section edit page are below.

3. Adding the necessary methods to schemas

Use the *executeProcess()* method of the *ProcessModuleUtilities* module to launch the process. As a parameter, pass the object with the following properties: the created business process name, the object with initialized incoming process parameters.

In the below source code, it is implemented in the *callCustomProcess()* method. The *isAccountPrimaryContactSet()* method of verifying the availability of the primary contact and the *getActions()* method of adding action menu options are also implemented.

The source code of the edit page replacing module:

```
define("AccountPageV2", ["ProcessModuleUtilities"], function(ProcessModuleUtilities)
{
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Account",
        // Methods of the edit page view model.
        methods: {
            // Verifies if the [Primary contact] page field is populated.
            isAccountPrimaryContactSet: function() {
                return this.get("PrimaryContact") ? true : false;
```

```
            },
            // Overriding the base virtual method that returns edit page action
collection.
            getActions: function() {
                // Parent method implementation is called to receive
                // the collection of initialized actions of the base page.
                var actionMenuItems = this.callParent(arguments);
                // Adding a separator line.
                actionMenuItems.addItem(this.getActionsMenuItem({
                    Type: "Terrasoft.MenuSeparator",
                    Caption: ""
                }));
                // Adding the [Conducting a meeting] menu option to the edit page
action list.
                actionMenuItems.addItem(this.getActionsMenuItem({
                    // Binding the caption of the menu option to localizable string
of the schema.
                    "Caption": { bindTo: "Resources.Strings.CallProcessCaption" },
                    // Binding the action handler method.
                    "Tag": "callCustomProcess",
                    // Binding the visibility property of the menu option to the
value, which returns the isAccountPrimaryContactSet() method.
                    "Visible": { bindTo: "isAccountPrimaryContactSet" }
                }));
                return actionMenuItems;
            },
            // Action handler method.
            callCustomProcess: function() {
                // Receiving the identifier of the account primary contact.
                var contactParameter = this.get("PrimaryContact");
                // The object that will be transferred to the executeProcess() method
as an argument.
                var args = {
                    // The name of the process that needs to be launched.
                    sysProcessName: "UsrCustomProcess",
                    // The object with the ContactParameter incoming parameter value
for the CustomProcess process.
                    parameters: {
                        ProcessSchemaContactParameter: contactParameter.value
                    }
                };
                // Launch of the custom business process.
                ProcessModuleUtilities.executeProcess(args);
            }
        }
    };
});
```

Add the *isAccountPrimaryContactSet()* method implementation to the section schema for the correct action display in the menu when displaying the page with the vertical list in the combined mode.

The source code of the section schema replacing module:

```
define("AccountSectionV2", [], function() {
    return {
        // Name of the section schema.
        entitySchemaName: "Account",
        methods: {
            // Verifies if the [Primary contact] field of the selected record is
populated.
            isAccountPrimaryContactSet: function() {
                // Defining the active record.
```

```
                var activeRowId = this.get("ActiveRow");
                if (!activeRowId) {
                    return false;
                }
                // Receiving the data collection of the section record list view.
                // Receiving the model of the selected account by the set value of
the primary column.
                var selectedAccount = this.get("GridData").get(activeRowId);
                if (selectedAccount) {
                    // Receiving the model property — availability of the primary
contact.
                    var selectedPrimaryContact =
selectedAccount.get("PrimaryContact");
                    // The method returns true if the primary contact is established.
Otherwise, it returns false.
                    return selectedPrimaryContact ? true : false;
                }
                return false;
            }
        }
    };
});
```

After you save the schemas and update the application page with clearing the cache, the new [Schedule a meeting] action will appear in the account page action menu (fig. 4). This action is available only if there exists a primary contact for the active list record. When executing the action, the "Conducting a meeting" custom business process will be launched. The primary contact of the account will be passed to the business process parameter (fig. 5)

Fig. 4. Launch of the business process by action on the edit page

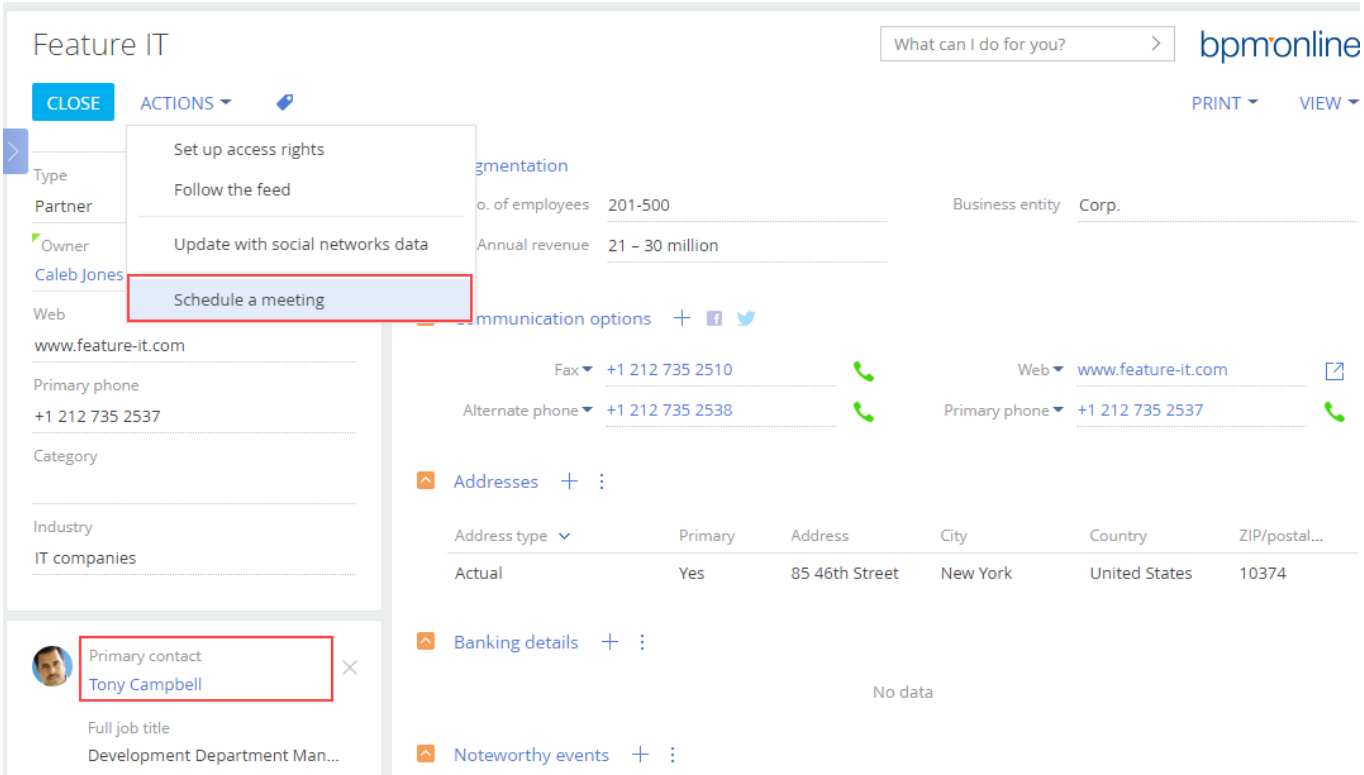
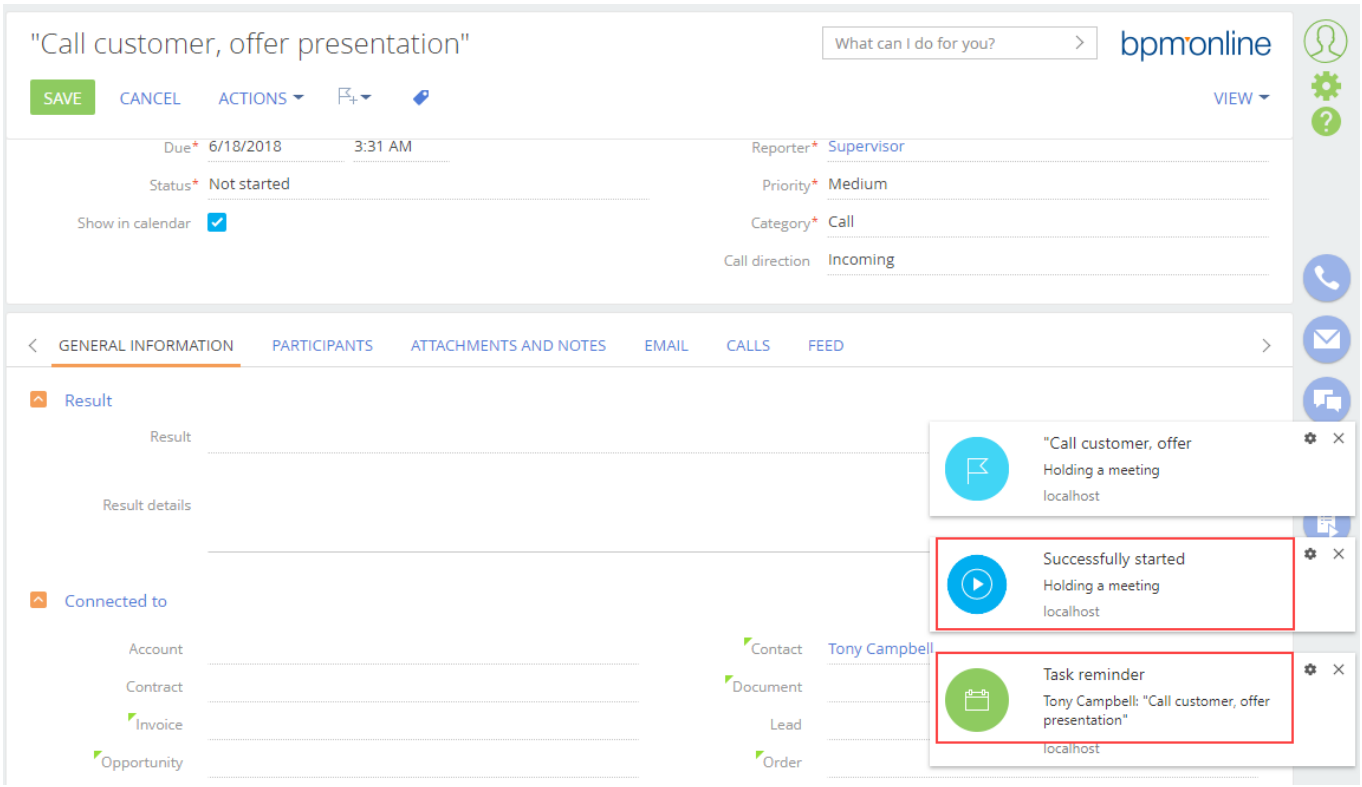


Fig. 5. Result of the business process launch. Passing the parameter from the account edit page to the business process



Creating custom [User task] process element

Introduction

It is often necessary to perform similar operations repeatedly while working with business processes in bpm'online. The [User task] process element is best suited for these operations. Learn more about the [User task] element in the “[User task] process element” article.

By default, a number of user tasks is already available in the system. You can add new user tasks if needed.

The “User task” configuration schema type is used to create new user tasks. The process task partially replicates the logic of the [Script task] process element. However, a user task can be reused in different processes. Any changes to the task will be immediately applied to all processes that contain the mentioned task.

Case description

Create a simple process user task that would calculate the sum of two numbers. Use two numbers (specified as task parameters) to calculate the sum.

Source code

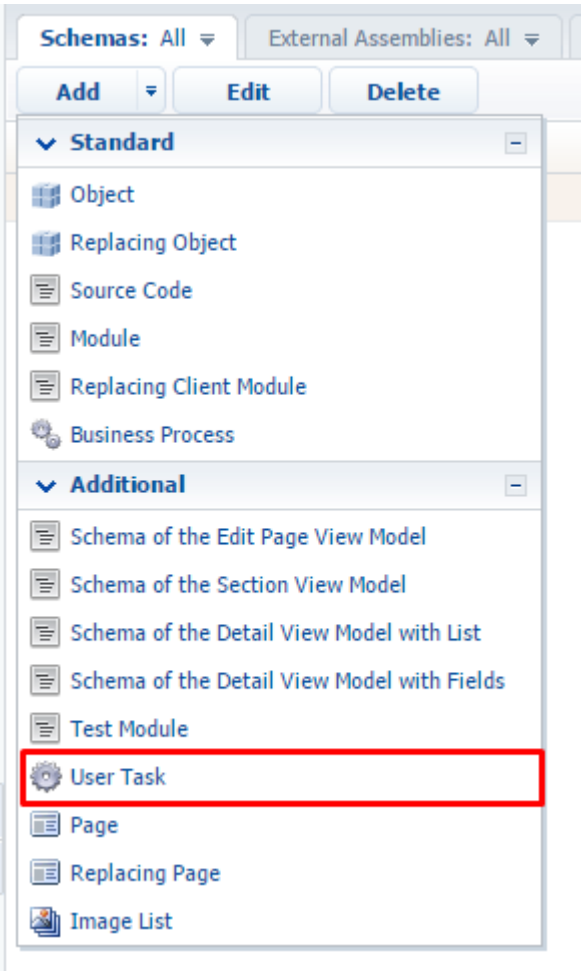
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Creating a user task schema

Go to the [Configuration] section of the system designer, select a custom package and execute the menu command [Add] - [User Task] on the [Schemas] tab (Fig. 1).

Fig. 1. Creating a user task schema



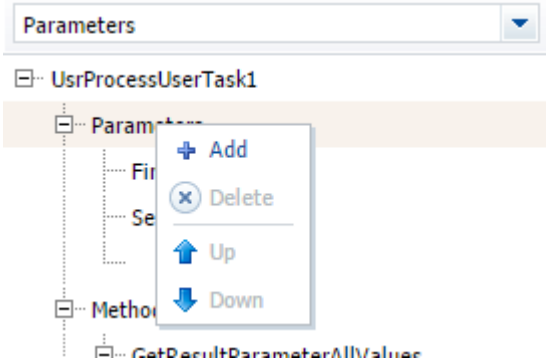
A user task designer window will open.

The default values for [Name] and [Title] are "UsrProcessUserTask1" and "User Task 1", respectively.

2. Adding task parameters

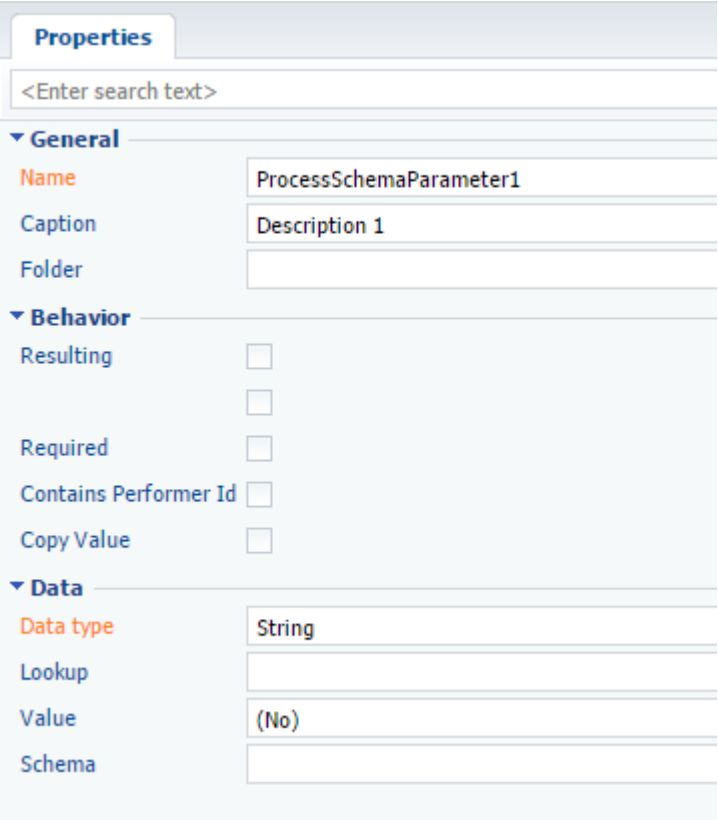
On the [Activity] tab of the process task designer, open the context menu on the [Parameters] element to add task result parameters. Execute the [Add] context menu command (Fig. 2).

Fig. 2. Creating a process task parameter



A new parameter will be added as a result, its main properties are displayed in Fig. 3.

Fig. 3. Default properties of a user task parameter



Add three parameters to create the process task (main properties are shown in table 1).

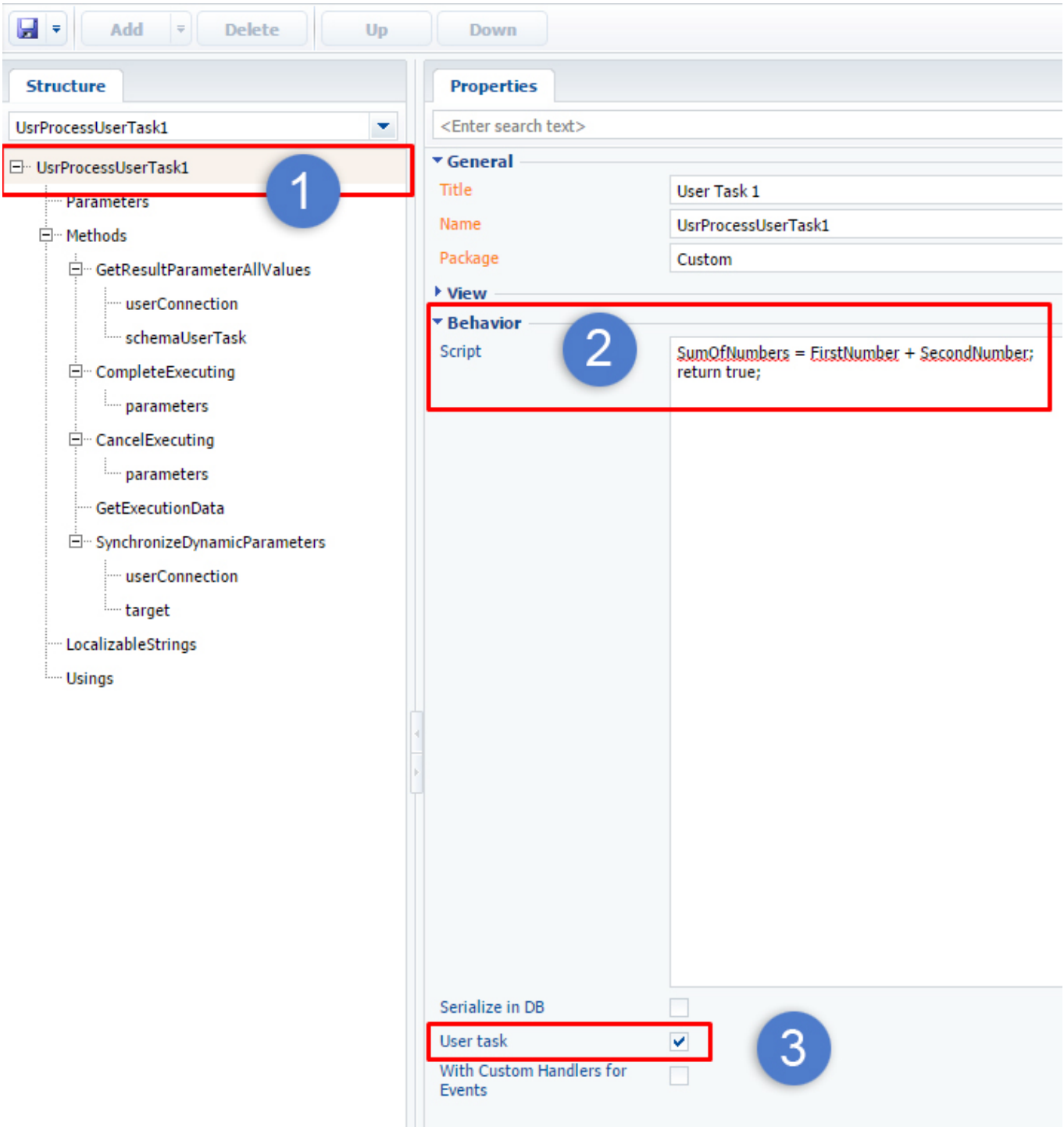
Table 1. Main parameters of the created process task

Name	Type	Description
FirstNumber	Integer	First number
SecondNumber	Integer	Second number
SumOfNumbers	Integer	Sum of numbers

3. Adding task logic

The task logic is set via a script. The task script is a process task parameter which contains the C# program code used to implement the necessary task logic.

Fig. 4. Adding user task logic



To add the task script program code, select the root element of the structure (Fig. 4, 1) and add the program code to the text field of the [Script] property (Fig. 4, 2):

```
// Performing operations with task parameters
SumOfNumbers = FirstNumber + SecondNumber;
// Indicates that the task script execution was successful.
return true;
```

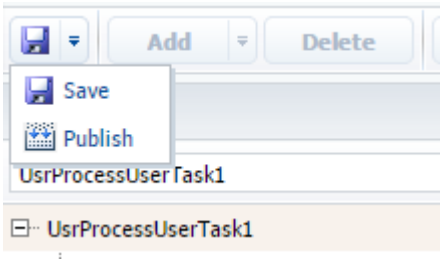
The parameters must be addressed directly by name. Return the *true* value at the end of the script to signal the successful execution of the element and continue the process.

Select the [User task] checkbox (located under the script input field (Fig. 4, 3)) to enable using the custom user task in business processes.

4. Saving and publishing the schema

After assigning values to the necessary properties of the created process task schema, save the schema and publish it (Fig. 5).

Fig. 5. Saving and publishing the schema



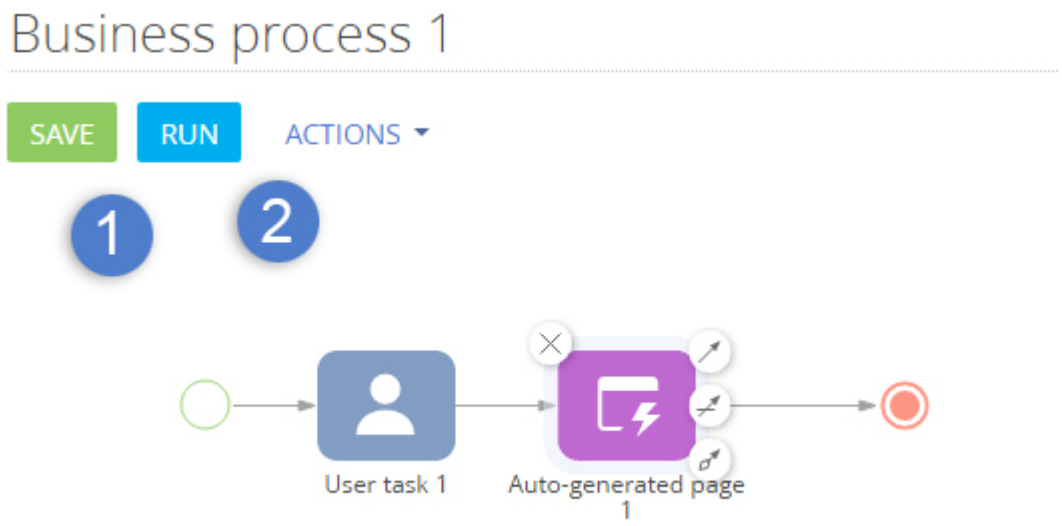
You can use the user task to create business processes after successfully publishing the schema.

5. Testing

Create a new business process to test the user task. Learn more about creating business processes in the “How to create business processes” article.

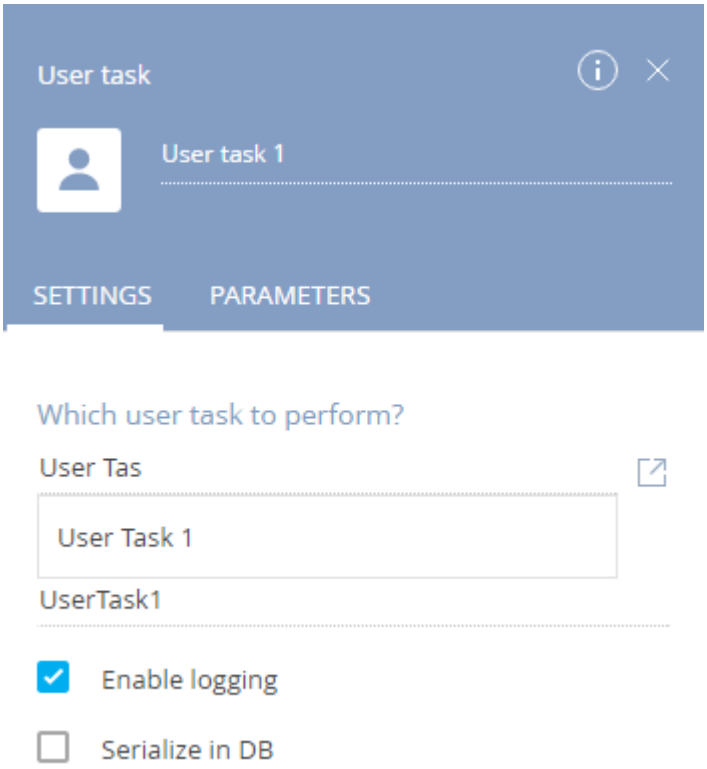
In the process designer, add the [User task] and the [Auto-generated page] elements to the process diagram (Fig. 6).

Fig. 6. Business process diagram.



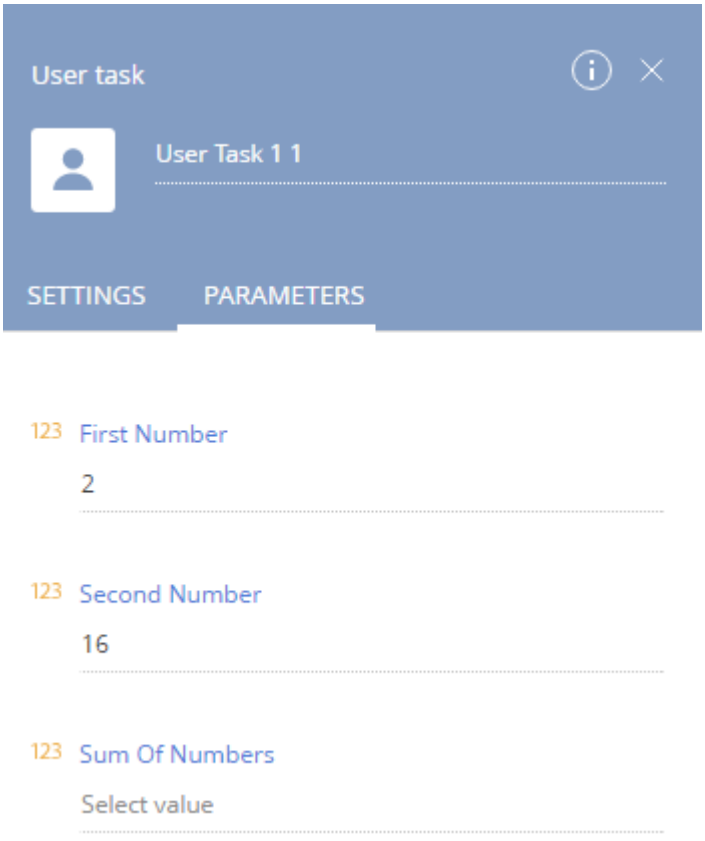
Specify the schema title (see step 1) in the [Which user task to perform?] property of the [User task 1] element (Fig. 7).

Fig. 7. Selecting a custom process task



Set the values of the parameters whose sum will be calculated (Fig. 8). The [Sum of numbers] parameter value will be determined by the script (see step 3), so the value entry field for this parameter can be left unpopulated.

Fig. 8. User task parameter values



The [Auto-generated page 1] element displays the task result, i.e. the sum of parameter values of the user task. To

display the sum of parameters, add an integer to the page element collection (Fig. 9) and set the title of the auto-generated page element (Fig. 10, 1).To set the displayed value, call the value formula dialog window by clicking the lightning icon in the [Value] field (Fig.10, 2).

Fig. 9. Adding page elements

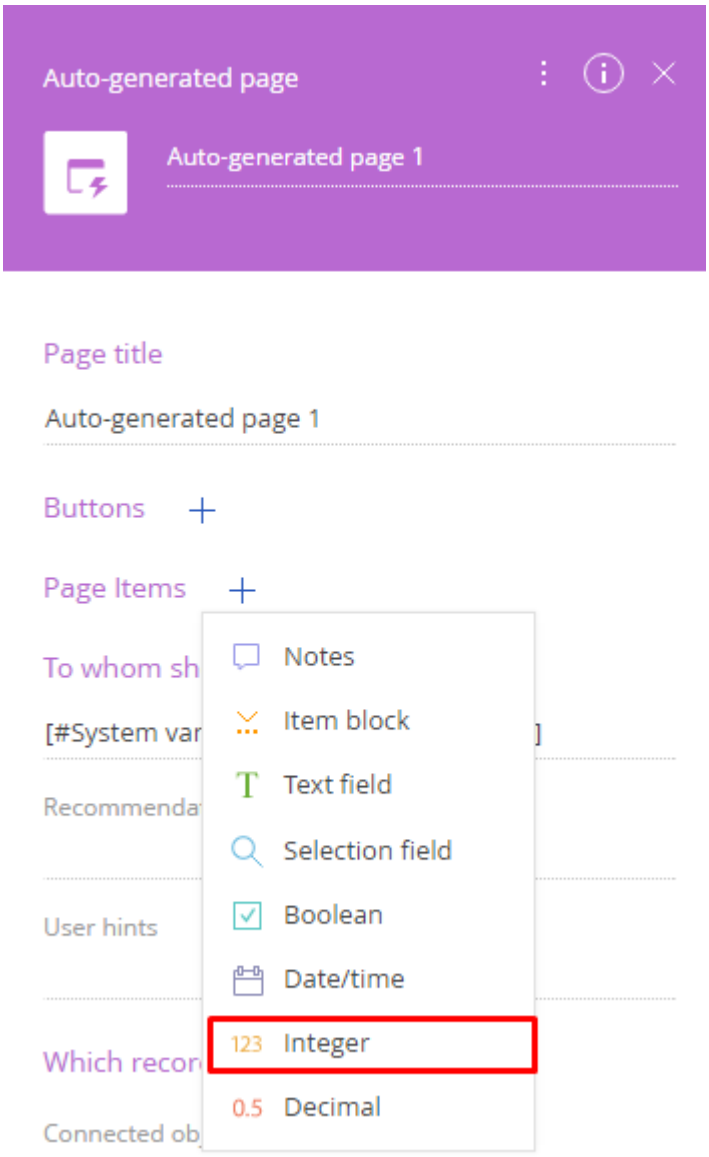
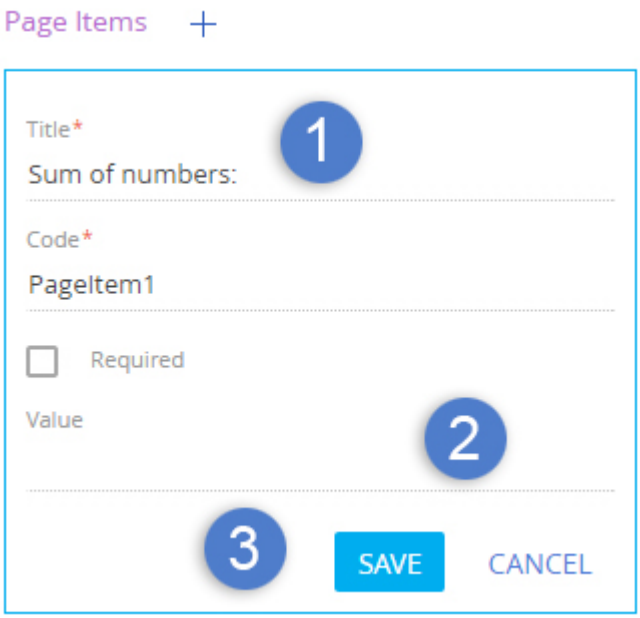
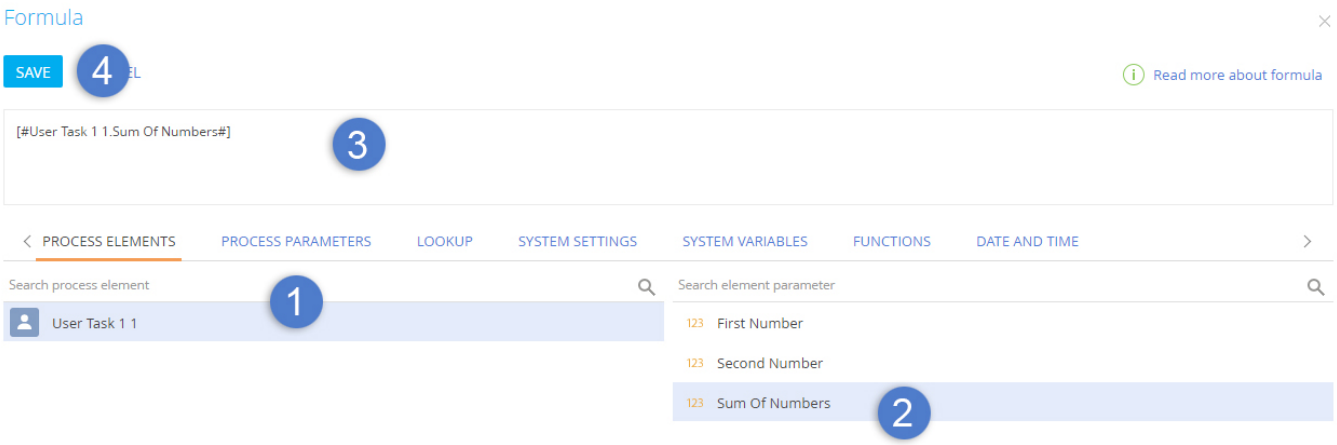


Fig. 10. Page element parameters



In the [Formula] dialog window that appears on the [Process Elements] tab, select the [User task 1] element (Fig. 11, 1) and double-click the [Sum of Numbers] element parameter (Fig. 11, 2). The formula used to calculate the auto-generated page value will be displayed (Fig. 11, 3).

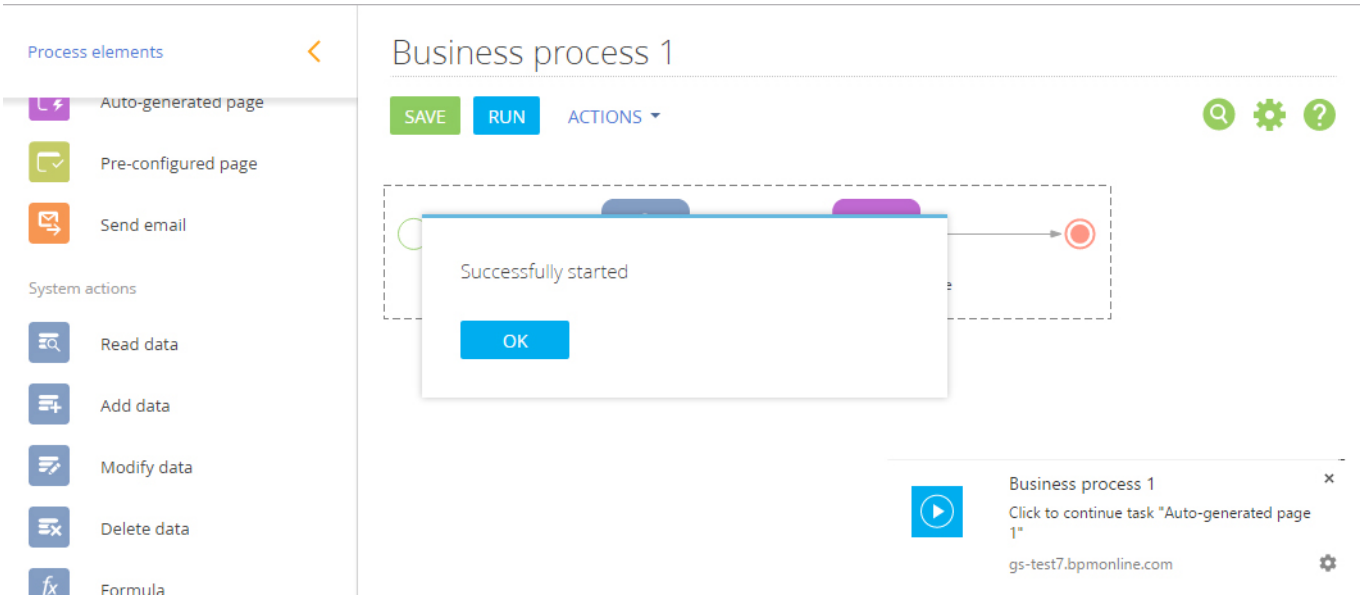
Fig. 11. The [Formula] dialog window



Save the formula (Fig. 11, 4) and the added auto-generated page element (Fig. 10, 3).

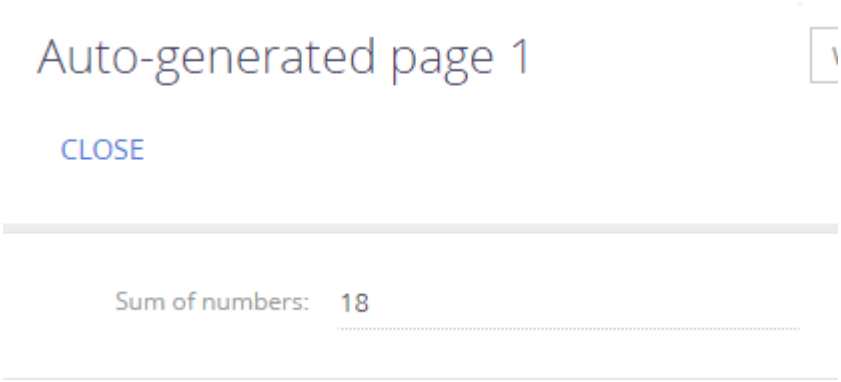
Run the process (Fig. 6, 2) after saving it (Fig. 6, 1). The message will alert that the process has started. A notification will enable you to display the business process log (Fig. 12).

Fig. 12. Business process start message



The business process result is shown in Fig. 13.

Fig. 13. Result of the business process execution.



> **Note**
>
> After changing the parameter values of the custom user task, change the currently displayed page before starting the business process again (go to any bpm'online section, for example). If you do not leave the auto-generated business process page, it will display the previous result.

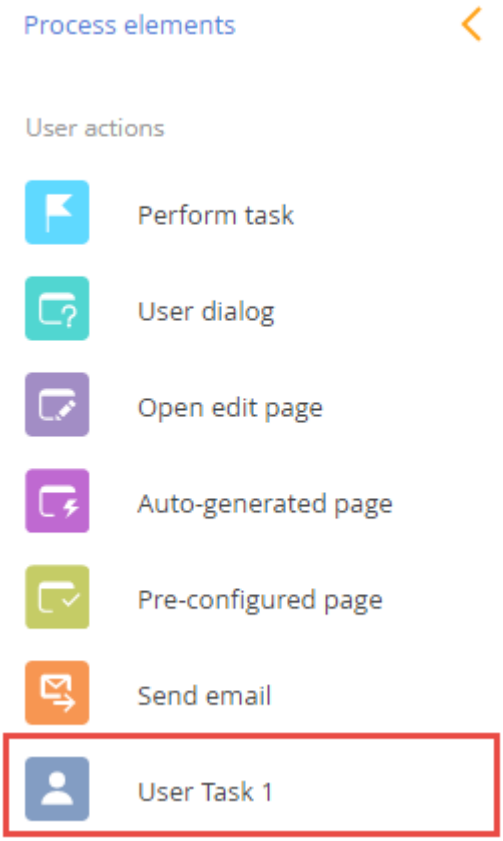
Adding the user process task to the [Process elements] tab

If the created user process task element is planned for regular use, it can be added to the [Process elements] tab in the process designer. To do so, execute the following SQL script in the database:

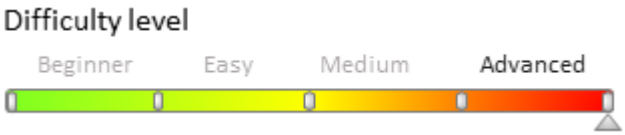
```
-- UsrProcessUserTask1 - name of the process task schema.
insert into SysProcessUserTask(SysUserTaskSchemaUId, Caption)
select s.UId, s.Caption from SysSchema s
where s.Name = 'UsrProcessUserTask1'
```

After restarting (or compilation) the application, the element will be displayed on the tab (Fig. 14).

Fig. 14. User task element on the tab



How to customize notifications for the [User task] process element



Introduction

The [User task] process element can create notification on the [Business Process Tasks] panel, just like any other process action. To activate the notification mechanism, select the [Serialize in DB] checkbox in the process properties and define the *ShowExecutionPage* logical parameter.

Default process actions are able to create notifications for process steps. Custom process actions must be manually assigned this parameter.

Case of creating a user task with a notification

Case description

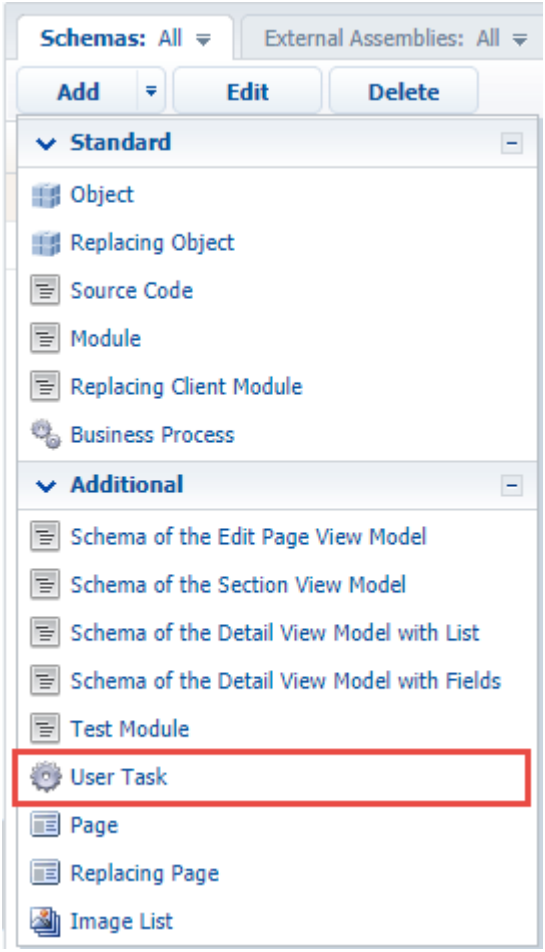
Create a simple custom process action ([User task]) that would automatically add a notification with “Attention” in its title and “Very important!” as its text. The notification must be displayed on the [Business process task] panel

Case implementation algorithm

1. Create a user task schema

To do this, go to the [Configuration] section of the system designer, select a custom package and on the [Schemas] tab, execute the menu command [Add] - [User Task] (Fig.1).

Fig. 1. Creating a user task schema



Set the property values specified in table 1 for the created schema.

Table 1. Custom process task properties

Property	Value
Title	Customized User Task
Name	UsrCustomizedUserTask
Small vector image	Scalable Vector Graphics The notification will be displayed on the [Business process task] panel. For this particular case we will use the SVG image available under the link.
Notification Icon (54x54 px)	Image for the notification pop-up window in PNG (Portable network graphics) format, 54x54 px. In the current case we will use the following image:  .
Serialize in DB	Select the checkbox.
User task	Select the checkbox.

After making the changes, save the schema's meta data.

2. Add the ShowExecutionPage parameter

To enable automatic adding of the notification when the user task is run, add a new parameter (ShowExecutionPage) to the Parameters section of the custom user task (Fig. 2). Its primary properties are listed in table 2.

Fig. 2. Adding a schema parameter

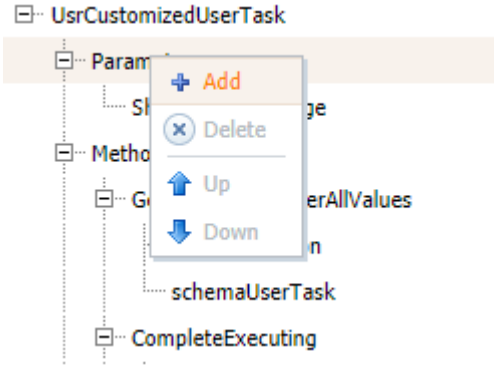


Table 2. Properties of the ShowExecutionPage parameter

Property	Value
Title	ShowExecutionPage
Name	ShowExecutionPage
Data type	Boolean

The value of the *ShowExecutionPage* parameter does not affect the notification mechanism. If the specified parameter exists in the process action, then before any process step implemented by this User task is executed, an automatic notification will be created for this step.

After making the changes, save the schema's meta data.

3. Override the GetNotificationData() method

The contents of the business process step notification is generated via the *GetNotificationData()* method that can be overridden.

The method must return an instance of the *Terrasoft.Core.Process.ProcessElementNotification* class that contains data for the business process step notification. We recommend calling the base method first, which will return instance of the *ProcessElementNotification*, populated with default values, and then customize this instance. Full description of the *ProcessElementNotification* class properties is available in the **.NET class libraries of platform core (on-line documentation)**.

The properties that are most useful for customization are available in table 3.

Table 3. Primary properties of the Terrasoft.Core.Process.ProcessElementNotification class

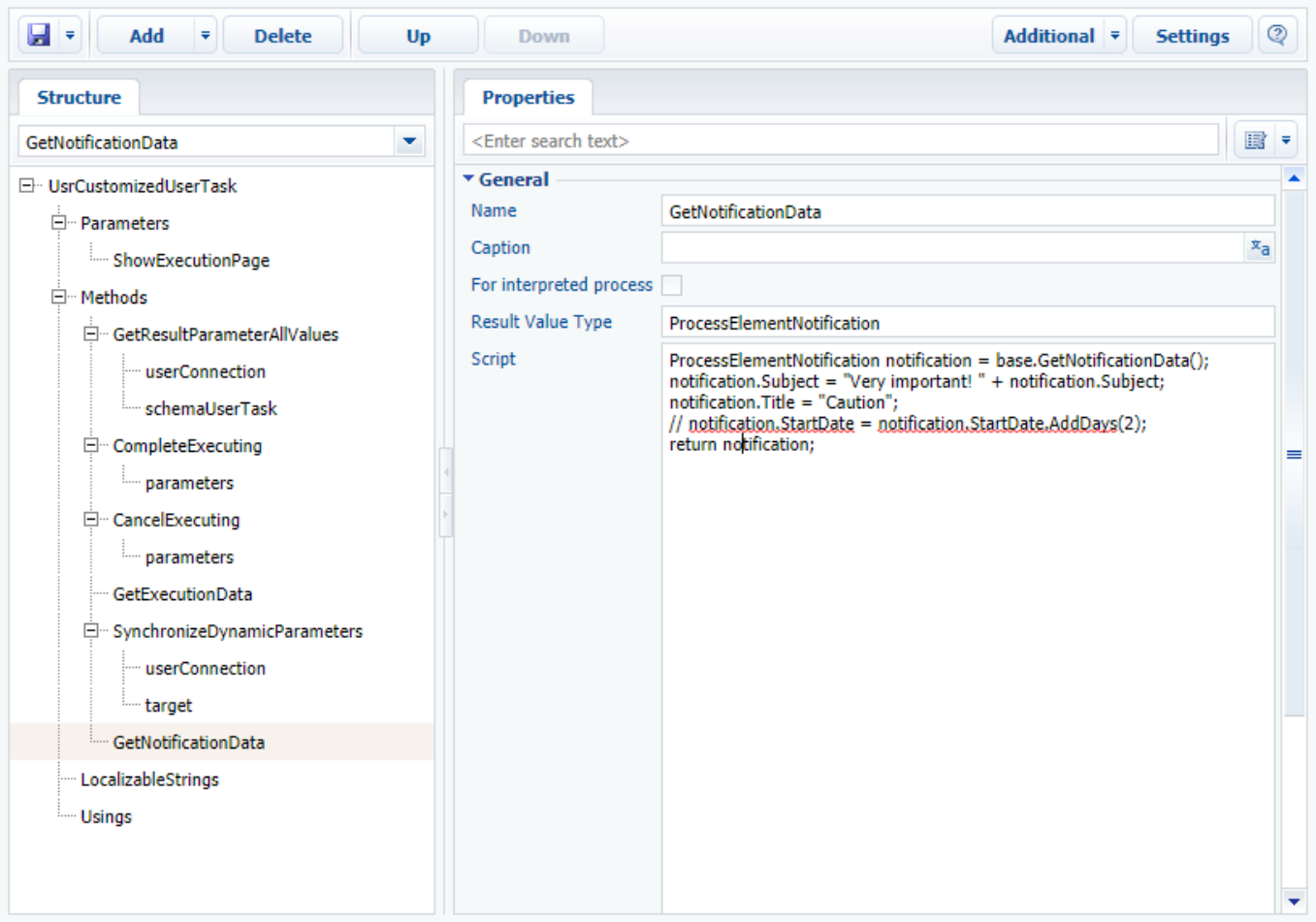
Property	Value
Title	Heading of the business process step notification. Default value – business process element name on the diagram.
Subject	Text of the business process step notification. Provide any process element specifics that are relevant to the notification recipient user here. Default value is the *NotificationCaption* process parameter value of the corresponding process. Thus, all steps of a business process will have the save value of the *Subject* parameter.
StartDate	Date and time of notification for system user. Default value – moment when the notification about the business process step has been created. This notification will be displayed for the user immediately after the process step is activated.

To execute the case conditions, override the *GetNotificationData()* method of the created schema. To do this, select the GetNotificationData node in the schema structure. Add the following code in the in the [Script] field (Fig. 3):

```
// Executing base method and getting an instance of the ProcessElementNotification
class generated by default.
```

```
ProcessElementNotification notification = base.GetNotificationData();
// Customizing the notification element.
notification.Subject = "Very important! " + notification.Subject;
notification.Title = "Attention";
// You can postpone date and time of the notification.
// notification.StartDate = notification.StartDate.AddDays(2);
// The method returns customized instance of ProcessElementNotification
return notification;
```

Fig. 3. Overriding the *GetNotificationData()* method

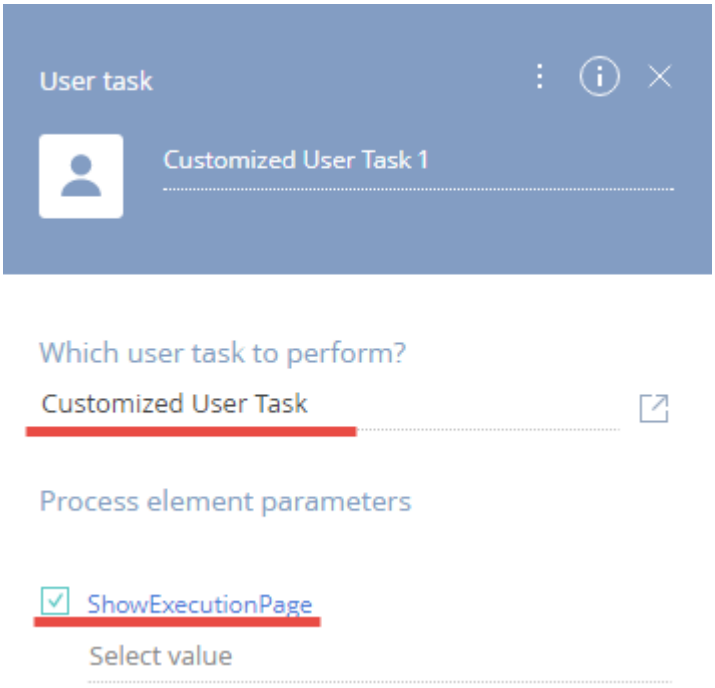


After making the changes, publish the schema.

4. Use the created element in the business process.

After publishing the user task schema, this action becomes available for use in bpm'online business processes. To use this custom action, add the [User Task] element on the process diagram and in the [Which user task to perform?] field, select "Customized User Task". After this, *ShowExecutionPage* Boolean parameter will be added to the user task parameters (Fig. 4). This parameter is optional.

Fig. 4. The *ShowExecutionPage* user task parameter



To correctly use the user task, select the [Serialize in DB] checkbox for it. Enable advanced mode in the process element properties (Fig. 5), and select the needed check box (Fig. 6, 1).

Fig. 5. Opening advanced properties of a process element

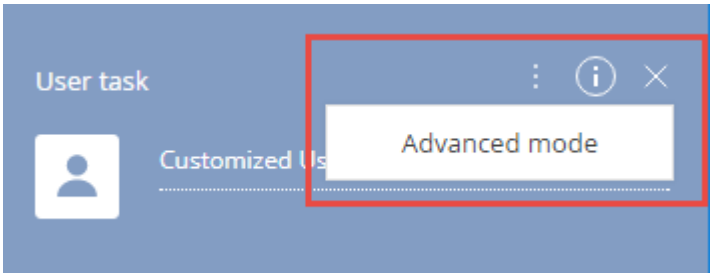
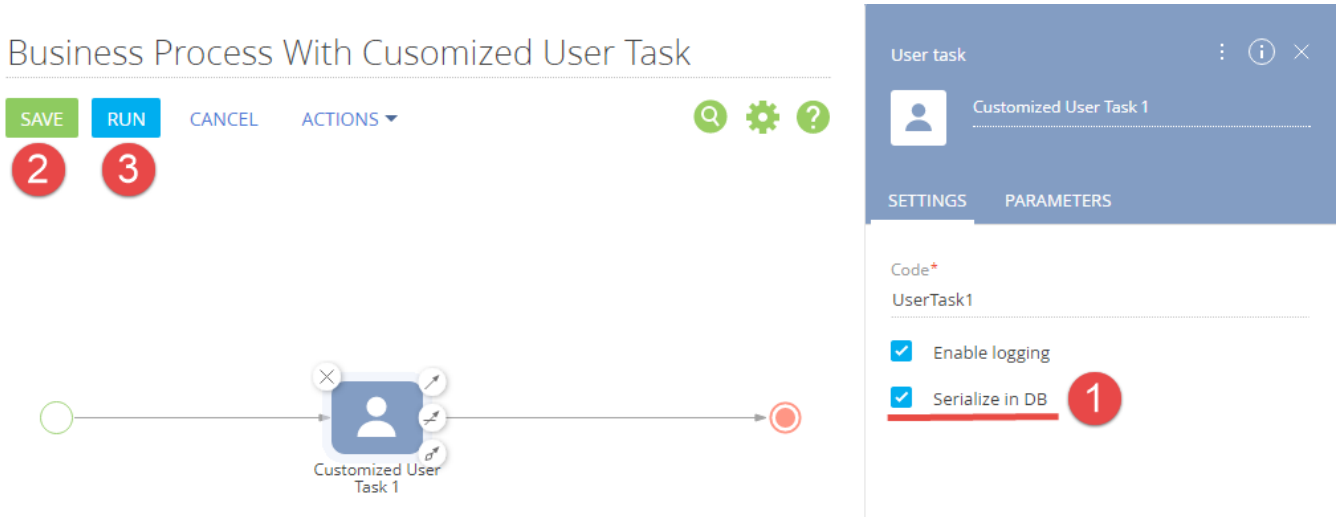


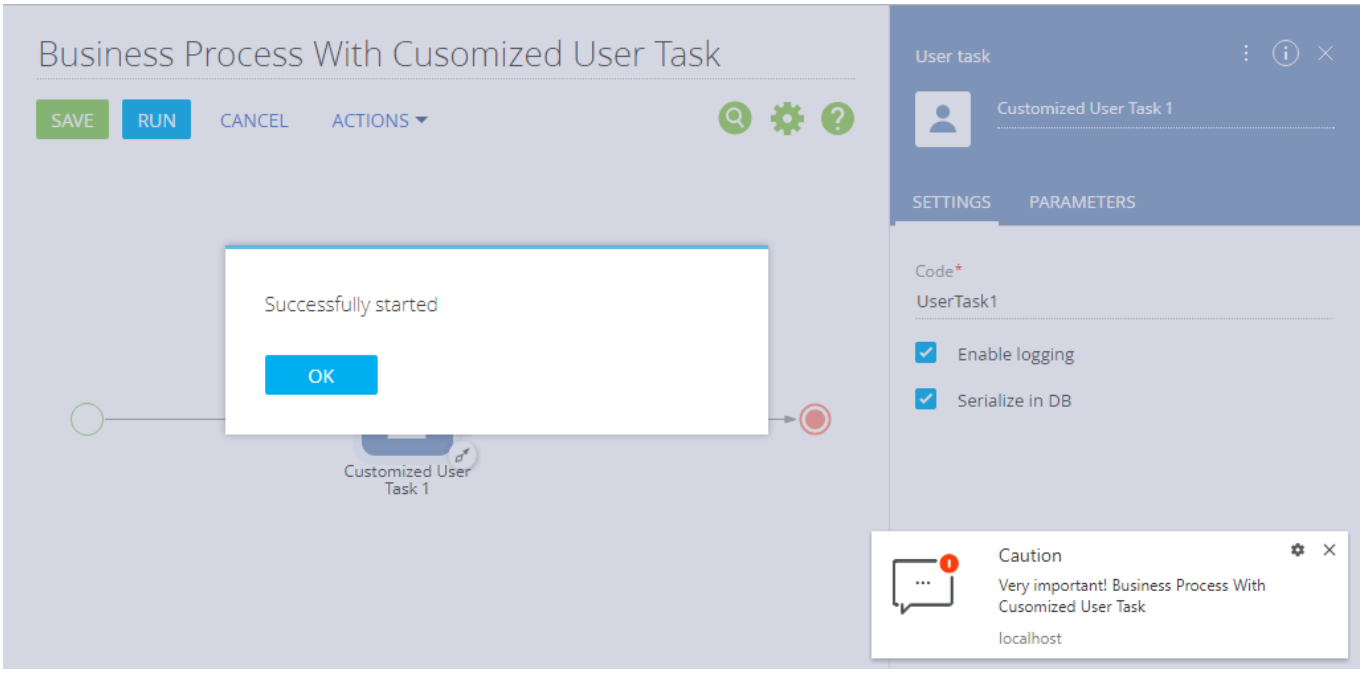
Fig. 6. The [Serialize in DB] check box



Save the process (Fig. 6, 2) and run it (Fig. 6, 1).

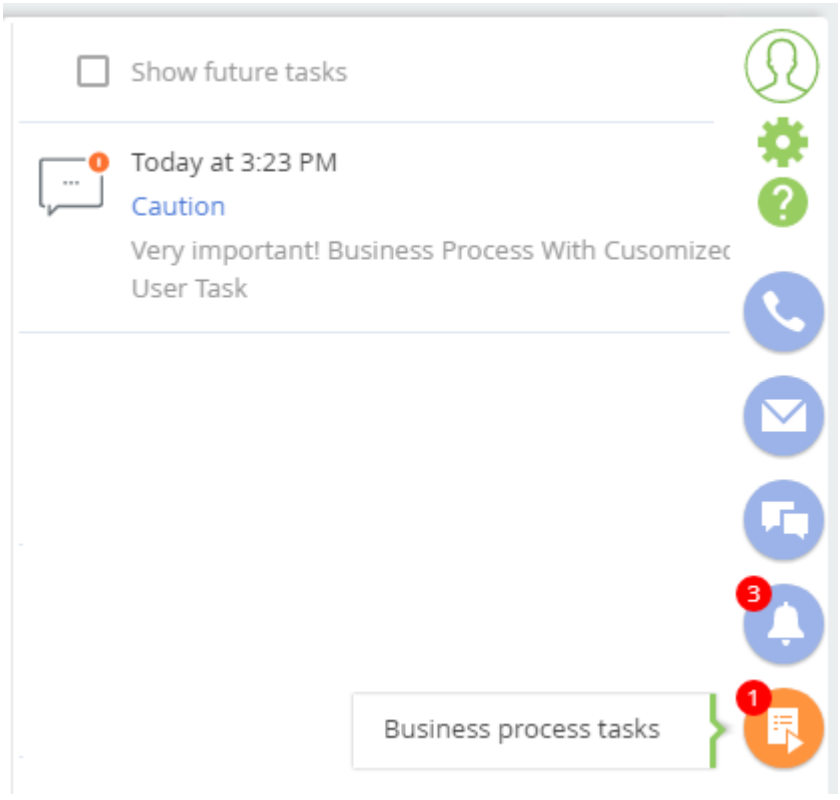
As a result, the corresponding notification (configured in the *GetNotificationData()* method) will be displayed (Fig. 7).

Fig. 7. Case result: Custom notification



The [Process tasks] panel will display a notification with the same properties (Fig. 8).

Fig. 8. Case result: Custom notification



How to run bpm'online processes via web service

Difficulty level

Beginner Easy Medium Advanced

Introduction

The *ProcessEngineService.svc* web service is used to run business processes from the third-party applications. Main features of the *ProcessEngineService.svc* web service are described in “**The ProcessEngineService.svc web service**” article.

Case description

Run a demo business processes of creating and reading bpm’online contacts from the browser address bar or third-party application via the *ProcessEngineService.svc* web service.

Case implementation algorithm

To implement the case:

1. Create the demo processes of adding new contact and reading all contacts.

2. Check the operability of the *ProcessEngineService.svc* web service from the browser address bar.

3. In the third-party application, create a class and implement logic of interaction with the *ProcessEngineServise.svc* web service in this class.

1. Creating the demo business processes

> ⚠ NOTE
>
> Best practices of business process creation in bpm’online are provided in the business process guide.

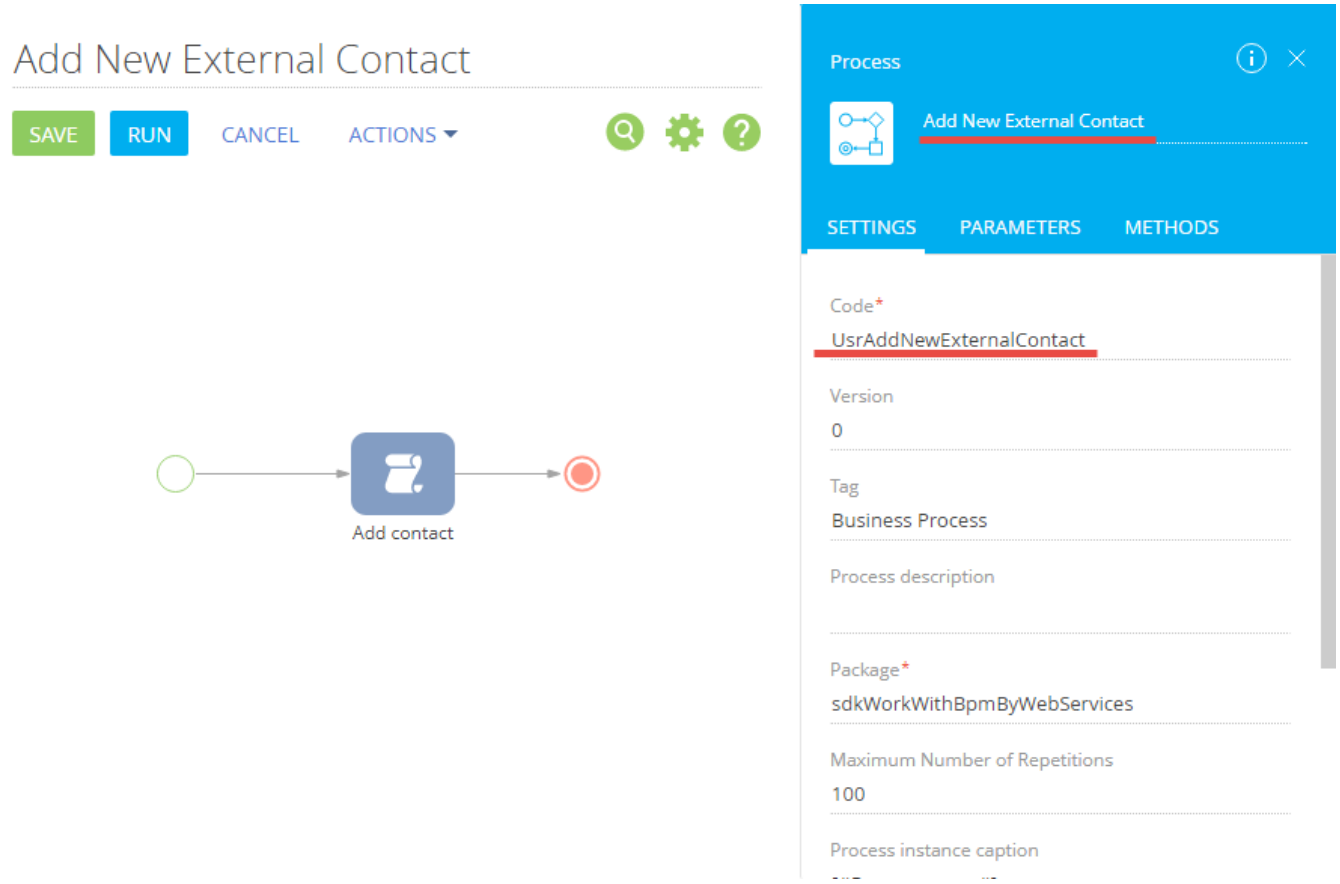
1.1. Creating the process of adding a new contact

The business process of adding a new contact has start event, end event and [ScriptTask] element in which the logic of adding a new contact is implemented. The values of business process properties (Fig. 1):

- [Name] – "Add New External Contact"
- [Code] – "UsrAddNewExternalContact".

Default values may be used for the other properties.

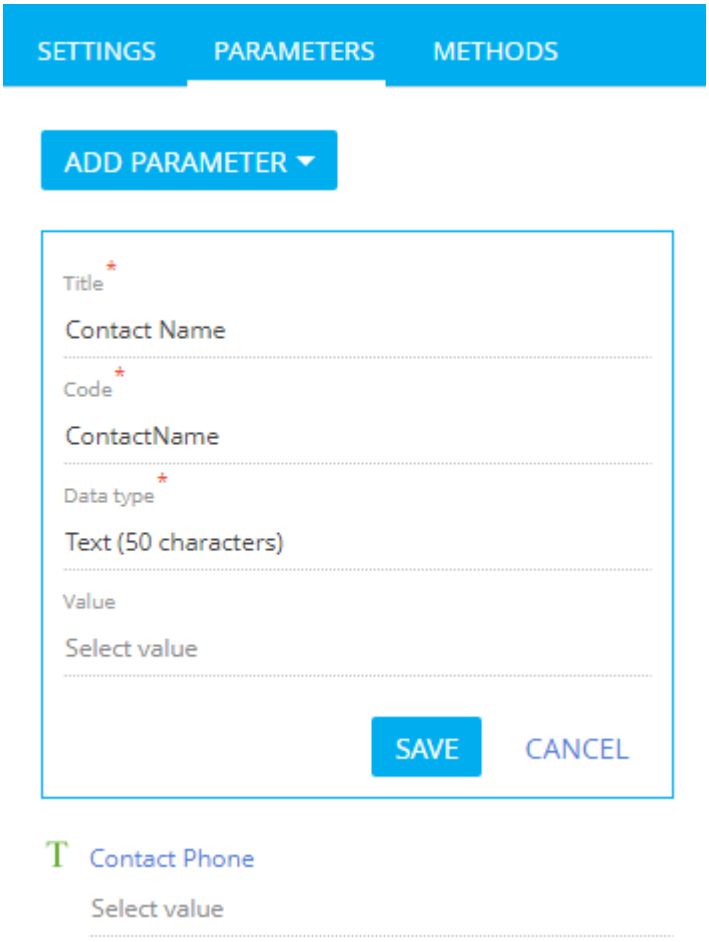
Fig. 1. Properties of the UsrAddNewExternalContact business process



The business process contains two text parameters (Fig. 2). The contact details are sent to the process via these parameters:

- *ContactName* – contains a name of the new contact
- *ContactPhone* – contains a phone number of the new contact.

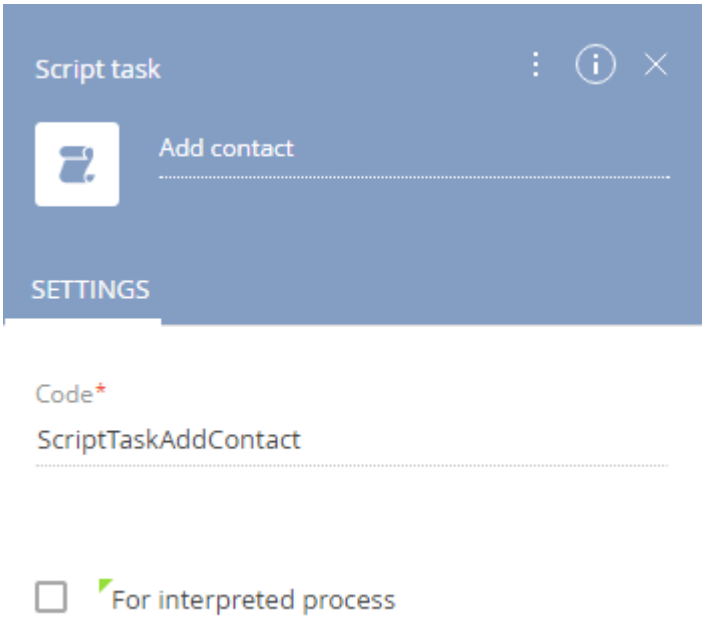
Fig. 2. The parameters of the business process.



Logic of adding a new contact is implemented in the [ScriptTask] element. The values of element properties (Fig. 3):

- [Name] – "Add contact"
- [Code] – "ScriptTaskAddContact"
- [For interpreted process] – checkbox unchecked.

Fig. 3. [ScriptTask] element properties



The source code for the ScriptTaskAddContact element:

```
// Create an instance of the schema of the "Contact" object.
var schema = UserConnection.EntitySchemaManager.GetInstanceByName("Contact");
// Create an instance of a new object.
var entity = schema.CreateEntity(UserConnection);
// Set the default values for the object columns.
entity.SetDefColumnValues();
// Set the value of the "Name" column from the process parameter.
entity.SetColumnValue("Name", ContactName);
// Set the value of the "Phone" column from the process parameter.
entity.SetColumnValue("Phone", ContactPhone);
// Saving a new contact.
entity.Save();
return true;
```

Save the business process to apply changes.

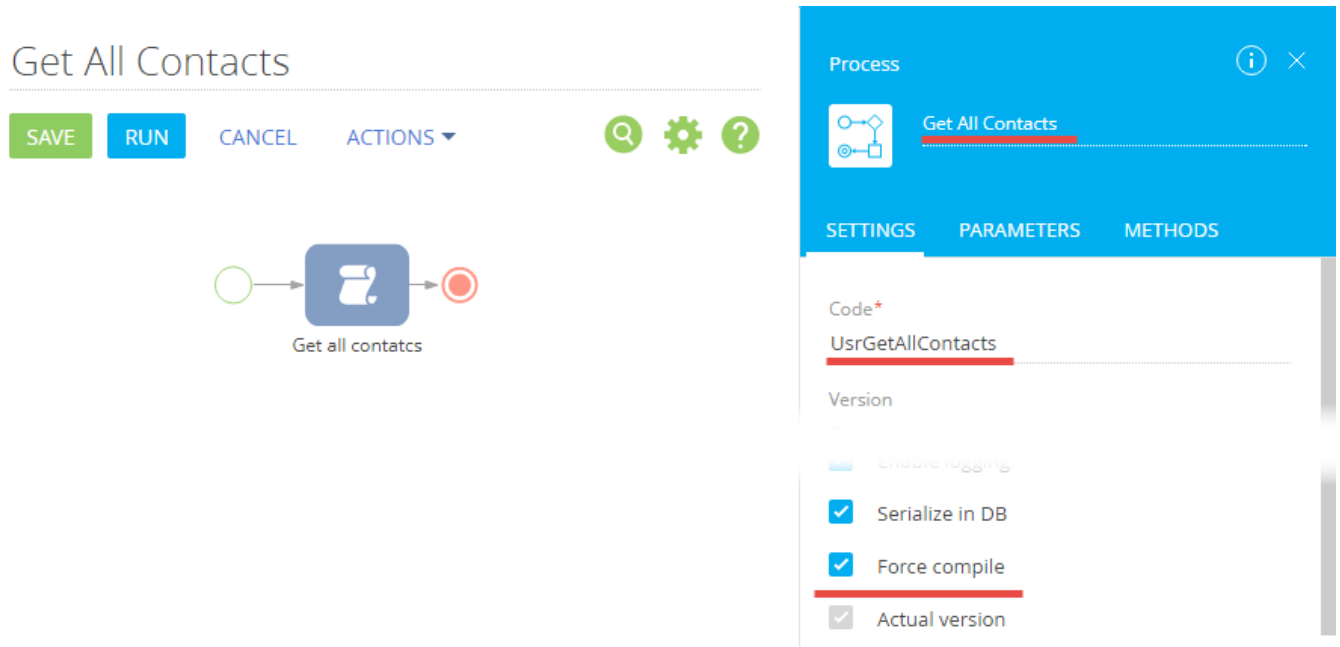
Creating the process of reading contacts

Business process that generates a list of all contacts is also contains one [ScriptTask] element in which the necessary logic is implemented. The values of business process properties (Fig. 4):

- [Name] – "Get All Contacts"
- [Code] – "UsrGetAllContacts"
- [Force compile] – checkbox checked.

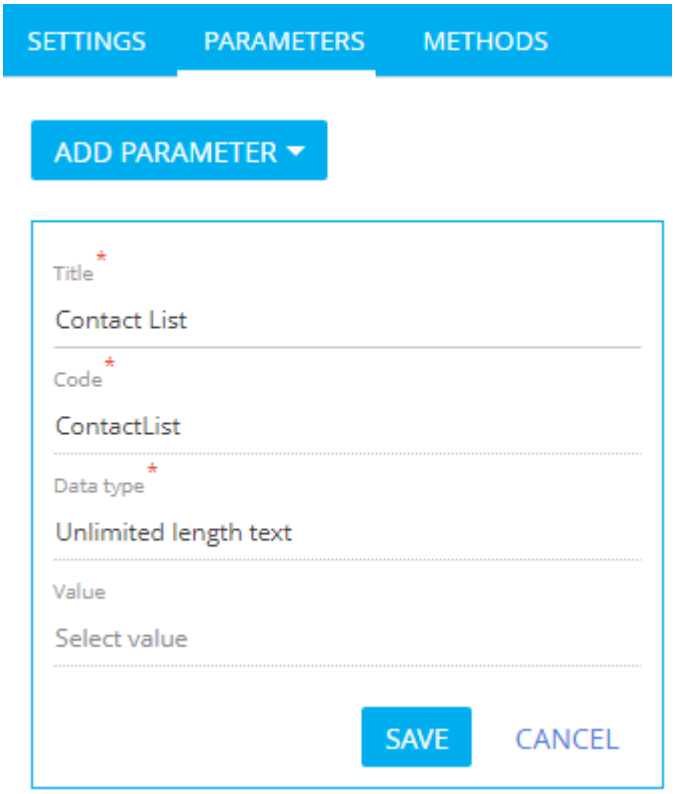
Default values may be used for the other properties.

Fig. 4. Properties of the contacts reading business process



The *UsrGetAllContacts* process contains the *ContactList* parameter. The process will get a list of all contacts as a JSON object through this parameter. Parameter type – unlimited length string. Parameter properties are listed on Fig. 5.

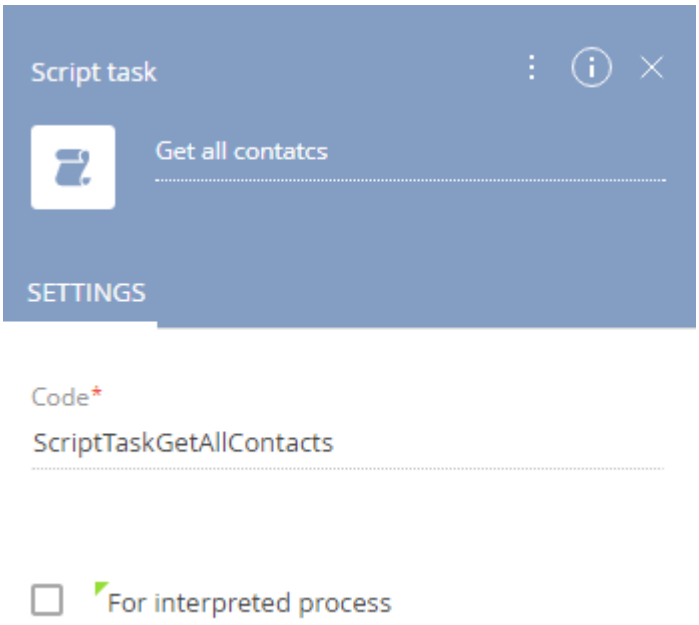
Fig. 5. Parameter properties



Logic of getting contacts is implemented in the [ScriptTask] process element. The values of element properties (Fig. 6):

- [Name] – "Get all contatcs"
- [Code] – "ScriptTaskGetAllContacts"
- [For interpreted process] – checkbox unchecked.

Fig. 6. [ScriptTask] element properties



The source code for the *ScriptTaskGetAllContacts* element:

```
// Create an EntitySchemaQuery instance.
EntitySchemaQuery query = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"Contact");
```

```
// A flag for required selection of the primary column (Id).
query.PrimaryQueryColumn.IsAlwaysSelect = true;
// Adding columns to the request.
query.AddColumn("Name");
query.AddColumn("Phone");
// Getting the result collection.
var list = query.GetEntityCollection(UserConnection);
// Creating a list of contacts for serialization in JSON.
List<object> contacts = new List<object>();
foreach (var item in list)
{
    var contact = new
    {
        Id = item.GetTypedColumnValue<Guid>("Id"),
        Name = item.GetTypedColumnValue<string>("Name"),
        Phone = item.GetTypedColumnValue<string>("Phone")
    };
    contacts.Add(contact);
}
// Save the serialized JSON collection of contacts to the ContactList parameter.
ContactList = JsonConvert.SerializeObject(contacts);
return true;
```

Save the business process to apply changes.

2. Run business processes from the browser address bar

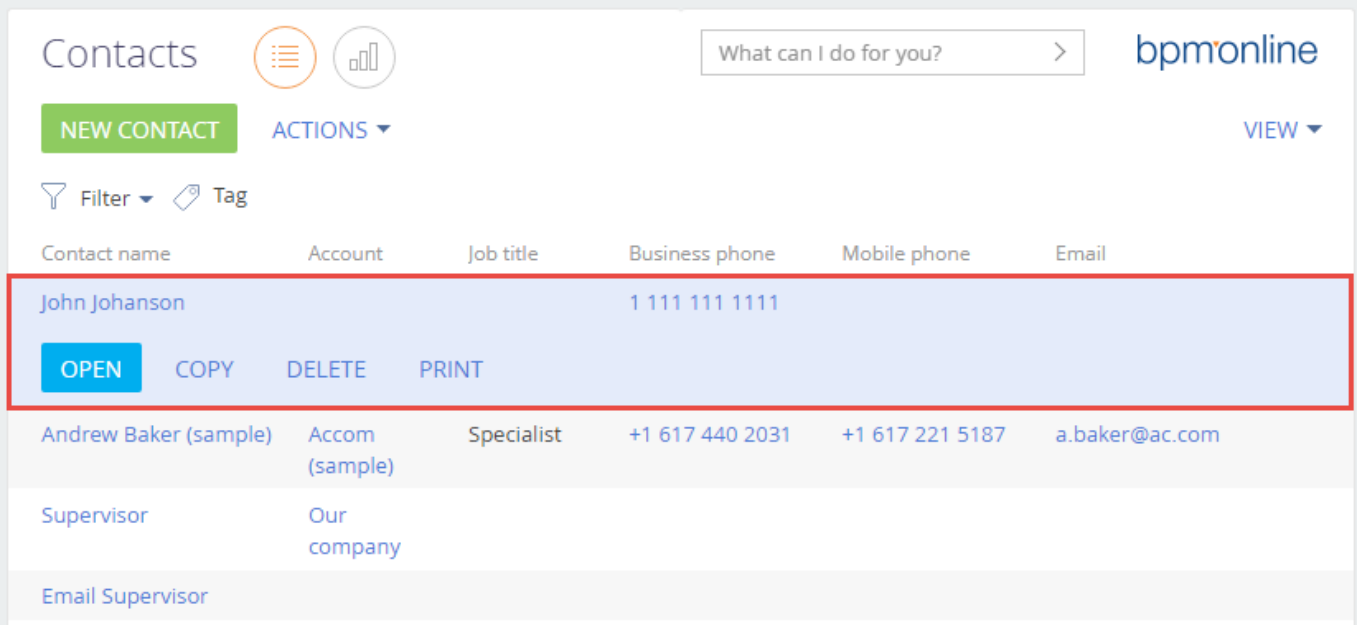
The call to the service method is possible using an HTTP GET request and you can use a standard browser to start a business process. General URL formats of calling a service for business processes with parameters are described in the "**The ProcessEngineService.svc web service**" article.

To launch the process of creation a new contact, enter the following URL in the browser address bar:

```
http[s]://<bpm'online_application_address>/0/ServiceModel/ProcessEngineService.svc/UsrAddNewExternalContact/Execute?ContactName=John Johanson&ContactPhone=+1 111 111 1111
```

After executing the request, a new contact will be added to bpm'online (Fig. 7).

Fig. 7. New contact



ATTENTION

A new contact will be created after each successful request to the service. If you run a number of queries with the same parameters, multiple duplicate contacts will be created.

To launch the process of reading all contacts, enter the following URL in the browser address bar:

```
http[s]://<bpm'online_application_address>/0/ServiceModel/ProcessEngineService.svc/UsrGetAllContacts/Execute?ResultParameterName=ContactList
```

After executing the request, a JSON object with collection of contacts will be displayed in the browser window (Fig. 8).

Fig. 8. Result of the contacts reading process

3. Run business processes from the third-party application

Before making requests to *ProcessEngineService.svc*, a third party application must be authenticated. Use the *AuthService.svc* authentication service for this. Information and examples of authentication of third-party application can be found in the "**The AuthService.svc authentication service**" article. Console application created according the example can be used for the case below.

ATTENTION

Full source code of the console application used for running business processes via the *ProcessEngineService.svc* service is available by a link https://github.com/bpmonline-academy/DevelopmentGuide/tree/master/Examples/WorkWithBpmByWebServices.

To generate requests to the *ProcessEngineService.svc* service, add a string field that contains base service URL to the *Program* class source code:

```
private const string processServiceUri = baseUri +
@"/0/ServiceModel/ProcessEngineService.svc/";
```

To run the business process of adding a new contact, add the following method to the source code of the *Program* class:

```
public static void AddContact(string contactName, string contactPhone)
{
    // Generating the URL request.
    string requestString = string.Format(processServiceUri +
            "UsrAddNewExternalContact/Execute?ContactName={0}&ContactPhone={1}",
                        contactName, contactPhone);
    // Generating Http request.
    HttpWebRequest request = HttpWebRequest.Create(requestString) as HttpWebRequest;
    request.Method = "GET";
    request.CookieContainer = AuthCookie;
    // Execute the request and analyze the Http response.
    using (var response = request.GetResponse())
    {
        // Because the service returns an empty string,
```

```
        // you can display the http response properties.
        Console.WriteLine(response.ContentLength);
        Console.WriteLine(response.Headers.Count);
    }
}
```

Add a method of starting the process of reading contacts:

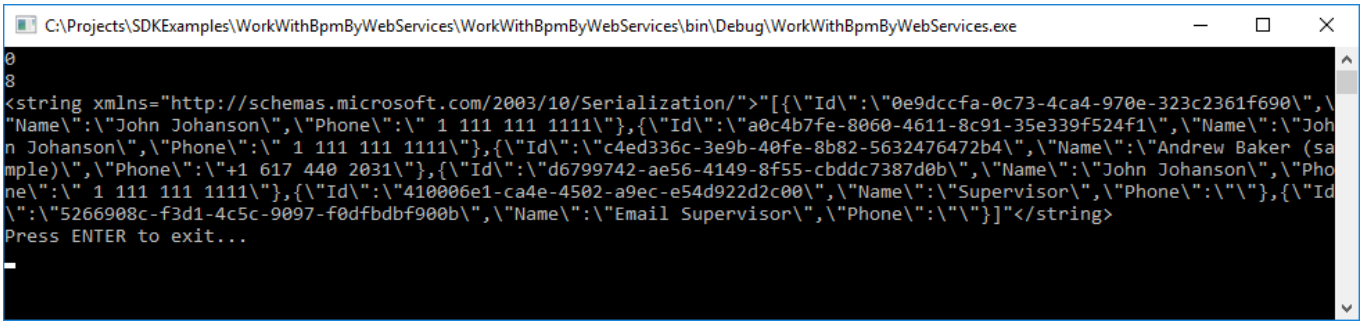
```
 public static void GetAllContacts()
{
    // Generating the URL request.
    string requestString = processServiceUri +
                        "UsrGetAllContacts/Execute?ResultParameterName=ContactList";
    HttpWebRequest request = HttpWebRequest.Create(requestString) as HttpWebRequest;
    request.Method = "GET";
    request.CookieContainer = AuthCookie;
    // Generating Http request.
    using (var response = request.GetResponse())
    {
        // Executing the request and output the result.
        using (var reader = new StreamReader(response.GetResponseStream()))
        {
            string responseText = reader.ReadToEnd();
            Console.WriteLine(responseText);
        }
    }
}
```

The added methods can be called in the main program method after successful authentication:

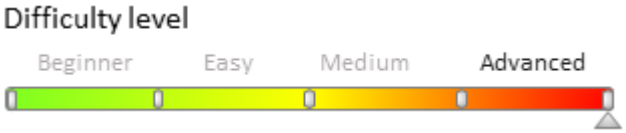
```
static void Main(string[] args)
{
    if (!TryLogin("Supervisor", "Supervisor"))
    {
        Console.WriteLine("Wrong login or password. Application will be
terminated.");
    }
    else
    {
        try
        {
            // Calling methods for starting business processes.
            AddContact("John Johanson", "+1 111 111 1111");
            GetAllContacts();
        }
        catch (Exception)
        {
            // Exception Handling.
            throw;
        }
    };
    Console.WriteLine("Press ENTER to exit...");
    Console.ReadLine();
}
```

The program result is shown in Fig. 9.

Fig. 9. Result of executing the application



How to save the record without closing the edit page which is opened by the business process



Introduction

If the record edit page is opened by the [Open edit page] business process element, the saving of the record (by clicking the [Save] button or by the *this.save()* method in the schema source code) will cause the closing of the page. Edit page is being closed even if the [Open edit page] element is not complete (configured in the [When is the element considered complete?] element property).

If you need to save the record several times without closing the edit page, pass the configuration object with the *isSilent* property set to *true* to the *this.save()* method. Example:

```
this.save({isSilent : true});
```

Case description

Create a business process that will open the invoice edit page. Save the Id of the edited record in the process parameter. In the source code of the edit page schema implement the program logic of saving the record each time the [Product in invoice] detail is being modified. Ensure the ability to edit detail records without closing the invoice edit page.

Case implementation

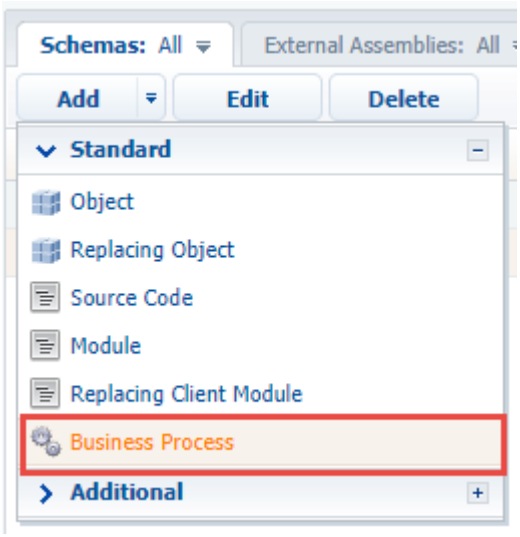
1. Business process creation

To do so, execute the following steps.

1.1 Create business process

In the [Configuration] section execute the [Add] – [Business process] action (Fig. 1).

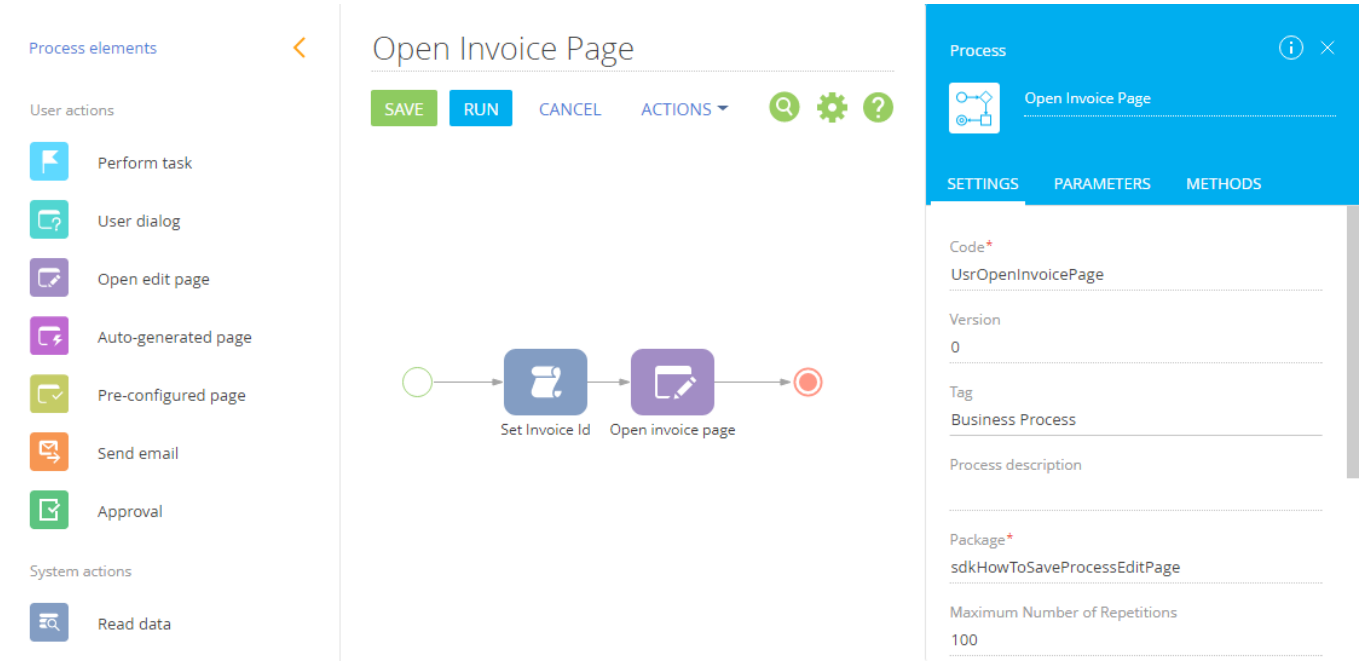
Fig. 1. [Add] – [Business process] action



In the opened process designer set the following values for the properties (Fig. 2):

- [Title] – "Open Invoice Page".
- [Code] – "UsrOpenInvoicePage".

Fig. 2. The properties of the business process

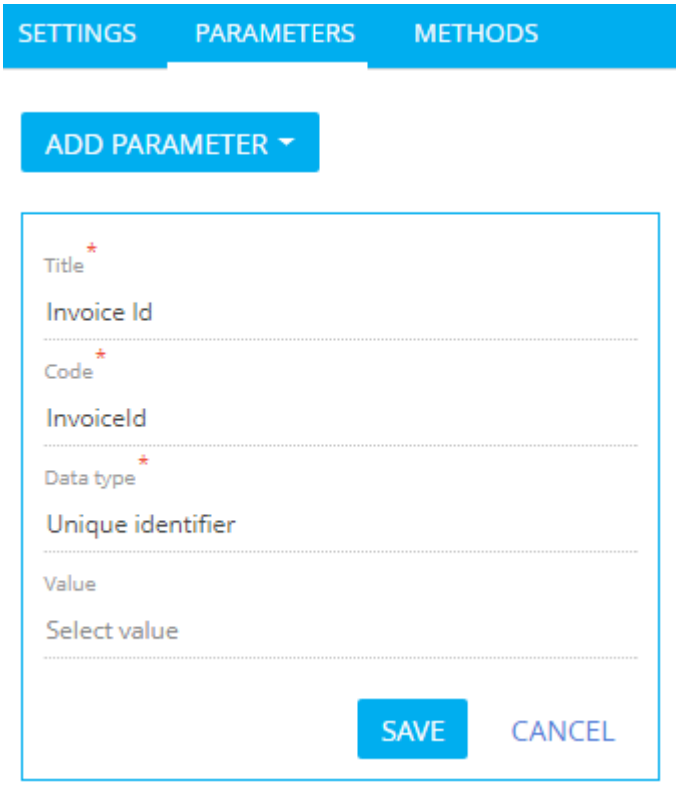


1.2 Add parameter

Add the parameter to the business process created on the previous step. This parameter will store the Id of the opened order record. Set following properties for the parameter (Fig. 3):

- [Title] – "Invoice Id".
- [Code] – "InvoiceId".
- [Data type] – "Unique identifier".

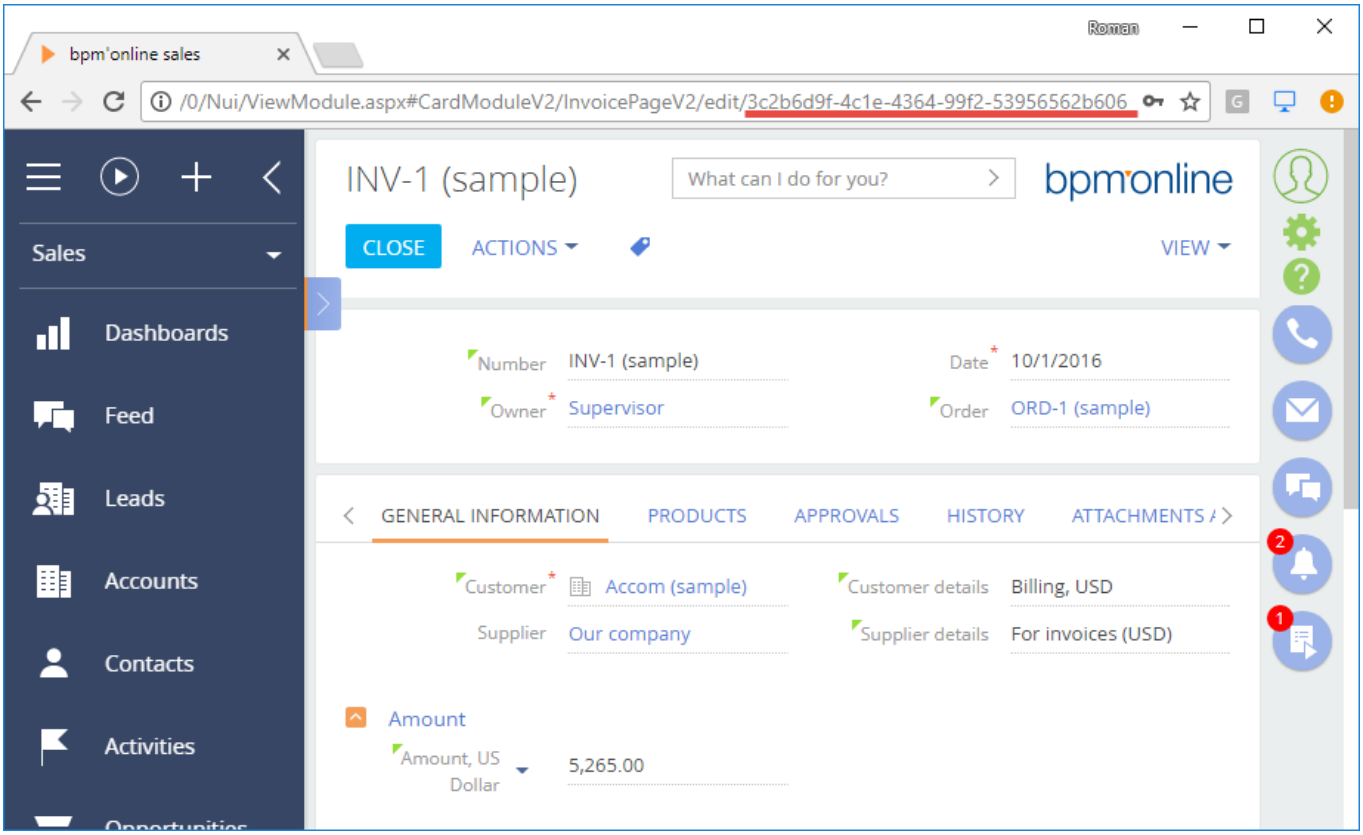
Fig. 3. The properties for the parameter of the business process



1.3 Add the [ScriptTask] element

You can find the value of the invoice record Id from the browser navigation bar by opening a record for editing (Fig. 4).

Fig. 4. Getting the record Id



This value can be saved to the *InvoiceId* parameter by the program code executed by the [ScriptTask] element.

For this add the [ScriptTask] element to the business process. The [Title] property of the element can be set to "Set Invoice Id". The element can execute following program code

```
Set<Guid>("InvoiceId", new Guid("3c2b6d9f-4c1e-4364-99f2-53956562b606"));
return true;
```

The InvoiceId parameter is set here. The instance of the *Guid* class is created on the basis of the string with the invoice record Id obtained from the browser navigation bar (Fig. 5).

Fig. 5. [ScriptTask] element properties

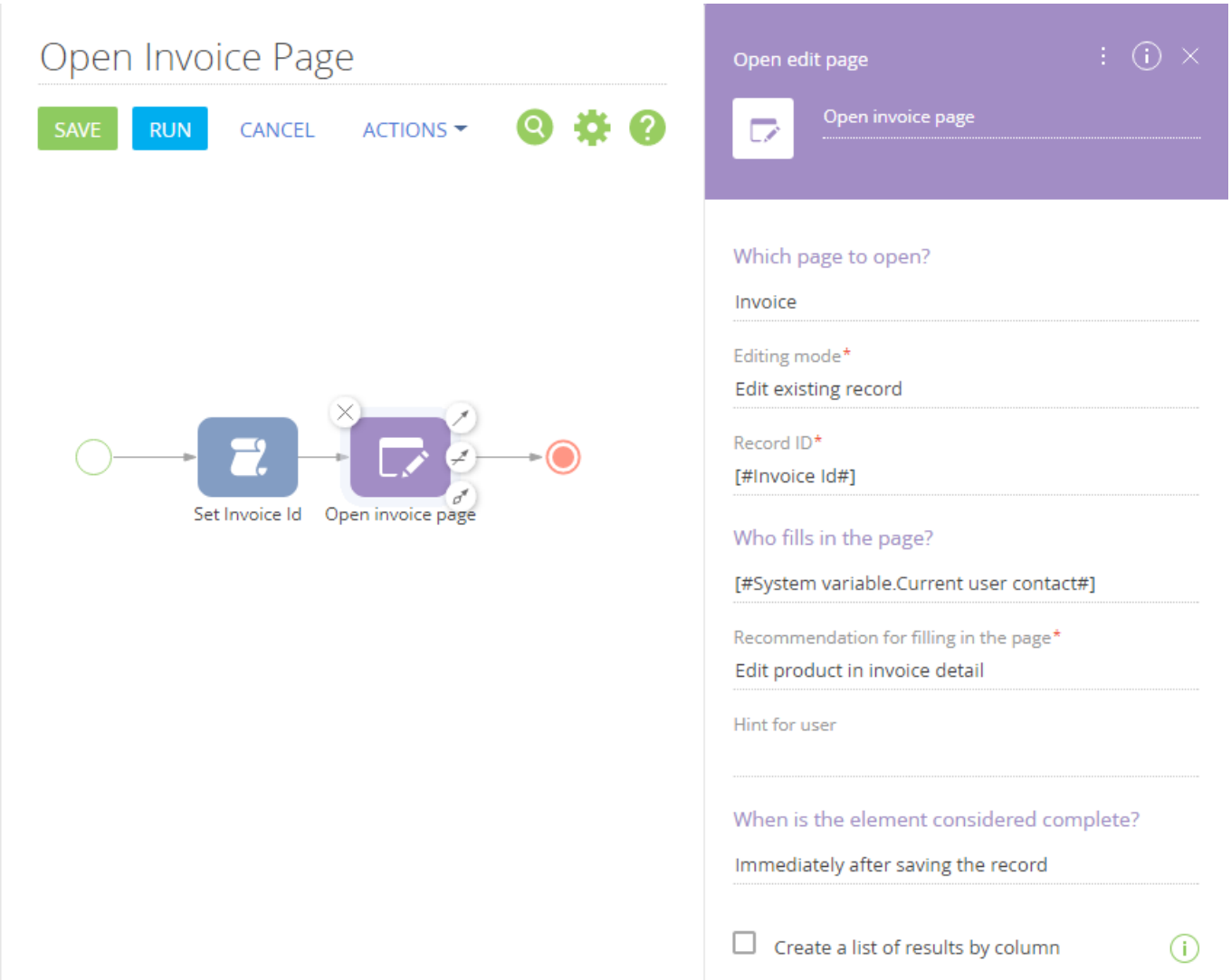
> **NOTE**
>
> The Id can be obtained by the instance of the *EntitySchemaQuery* class (see “**The use of EntitySchemaQuery for creation of queries in database**”).

1.5. Add the [Open edit page] element

Use the [Open edit page] element to open the page for editing during the process execution. Set following properties for this element (Fig. 6):

- [Title] – "Open invoice Page".
- [Which page to open?] [Which page to open?] – "Invoice".
- [Editing mode] – "Edit existing record".
- [Record Id] – select the [Invoice Id] process parameter added on the Step 1.2.
- [Recommendation for filling in the page] – "Edit product in invoice detail".
- [When element is considered complete?] – "Immediately after saving the record".

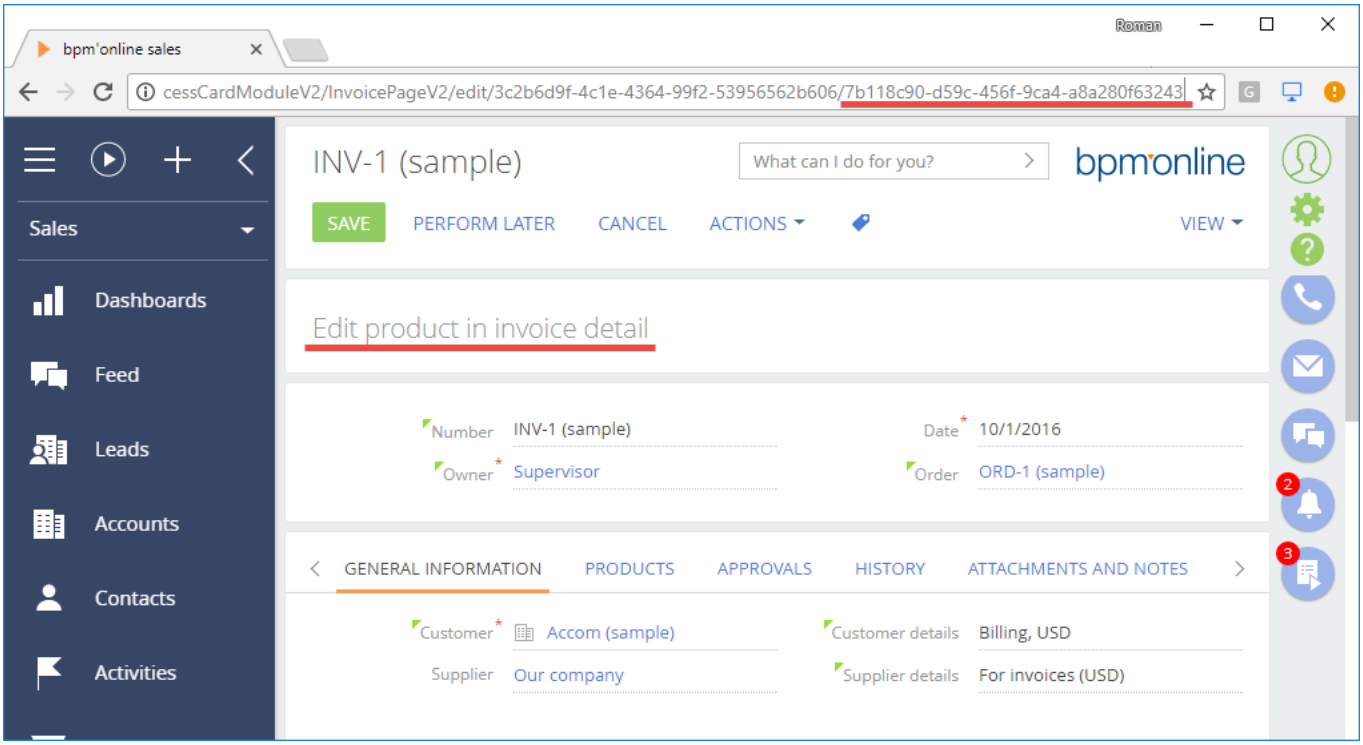
Fig. 6. [Open edit page] element properties



Save the business process to apply changes.

The start of the business process will open the record edit page which will be automatically closed when saving (Fig. 7).

Fig. 7. Edit page opened by the business process



2. Add the program logic to the edit page schema

To save the record when modifying the [Product in invoice] detail without closing the edit page, execute the following steps.

2.1 Add a replacing schema of the invoice edit page

The procedure for creating a replacing schema of the edit page is covered in the "**Creating a custom client module schema**" article. Select the "Invoice edit page" (*InvoicePageV2*) schema as a parent object.

2.2 Override the onDetailChanged() method

In the replacing schema of the invoice edit page override the *onDetailChanged()* method implemented in the *BaseEntityPage* base schema. This method is the handler of the received message about modification of the detail on the edit page.

To ensure editing of the records of the [Product in invoice] detail without closing the invoice edit page, add the following source code to the schema.

```
define("InvoicePageV2", [], function() {
    return {
        entitySchemaName: "Invoice",
        methods: {
            // The handler for the detail change message.
            onDetailChanged: function(detail, args) {
                this.callParent(arguments);
                // Only for the [Products in invoice] detail
                if (detail.schemaName === "InvoiceProductDetailV2") {
                    // Save a record with the automatic closing of the edit page.
                    //this.save();
                    // Save the record without closing the edit page.
                    this.save({isSilent : true});
                }
            }
        },
        diff: /**SCHEMA_DIFF*/[
        ]/**SCHEMA_DIFF*/
```

```
    };
});
```

Save the schema to apply changes.

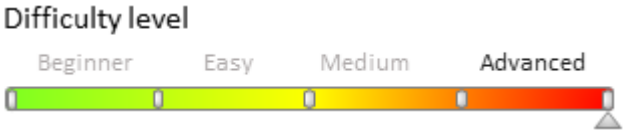
As a result, after start of the business process the record edit page will open (Fig. 7). The page will be closed only after clicking the [Save] button. The record opened for edit will be saved after each modification of the [Product in invoice] detail without closing the edit page.

Typical customizations

Contents

- **Creating pop-up summaries (mini pages)**
- **Adding pop-up summaries (mini pages) to a module**
- **Creating a pop-up summary (mini page) for adding records**
- **Adding pop-up hints**
- **How to modify sales pipeline calculations**
- **How to enable additional filtering in a sales pipeline**
- **Adding a custom dashboard widget**
- **The Terrasoft.AlignableContainer custom element**
- **Adding a duplicate search rule**
- **Junk case custom filtering**
- **How to display custom implementation of approving in the section wizard**
- **How to create custom reminders and notifications**
- **How to create the [Timeline] tab tiles bound to custom section**
- **Adding multi-language email templates to a custom section**

Creating pop-up summaries (mini pages)



Introduction

Starting from version 7.7 we have introduced a new module in bpm’online - a pop-up summary. For regular users, pop-up summaries are improved screen tips containing additional functions based on the current section. Using pop-up summaries enables receiving information about the account address and opening its location on a map, sending emails or making contact calls directly from the section without opening the edit page. You can see examples of pop-up summaries by hovering the cursor over hyperlinks pointing at edit pages in the [Accounts] and [Contact] sections.

Primary purposes of using pop-up summaries:

- Enabling users to get the necessary information by record without opening edit pages.
- Providing possibility to quickly add records to sections with populating only the required fields without opening full record pages.

The structure of a pop-up summary view model schema does not differ from the general structure of bpm’online module schema. Required properties of pop-up summary schema structure include:

- *entitySchemaName* containing the object schema name bound to a pop-up summary
- the *diff* modification array

These parameters enable building a module view in bpm’online custom interface.

You can also use other general schema structure elements to implement the necessary functions, such as attributes,

methods, mixins and messages that can be used to add custom control elements, register messages and form the pop-up summary business logics. The appearance of pop-up summary visual elements can be modified using custom styles.

> ⚠ ATTENTION
>
> Pop-up summaries do not support the mechanism of business logics setup via business rules.

To add a custom pop-up summary to a current bpm’online section:

1. Add a pop-up view model schema to the custom package. Select *BaseMiniPage* schema as a parent object.
2. Modify the *SysModuleEdit* system table in the bpm’online database via a special SQL query.
3. Add the necessary pop-up summary functions to the schema source code. Specify the object schema name in the *entitySchemaName* element bound to the pop-up summary and perform at least one modification in the *diff* array.
4. Apply styling to the pop-up summary.
5. Add the *Has[Section code]MiniPageAddMode* setting.

> ⚠ ATTENTION
>
> To bind a pop-up summary to specific section objects, specify the unique pop-up summary identifier in the *MiniPageSchemaUId* column of these objects. Currently you can only do it by modifying the *SysModuleEdit* system table of bmp’online database via an SQL-query.
>
> Pay high attention to creating and executing the SQL query. Executing an incorrect SQL query can damage the existing data and disrupt the system.

> ⚠ NOTE
>
> For bpm’online sections with default pop-up summaries, there are system settings, whose codes have the following format *Has[Section code]MiniPageAddMode* (for instance, *HasAccountMiniPageAddMode*). These system settings are used to toggle between the two modes: adding new records and editing existing records via pop-up summaries.

You can create a pop-up summary for any bpm’online object.

Case description

Creating a custom pop-up summary for the [Knowledge base] section. The pop-up summary will be used for viewing the basic [Name] and [Tags] fields with a possibility to download the attached files.

Source code

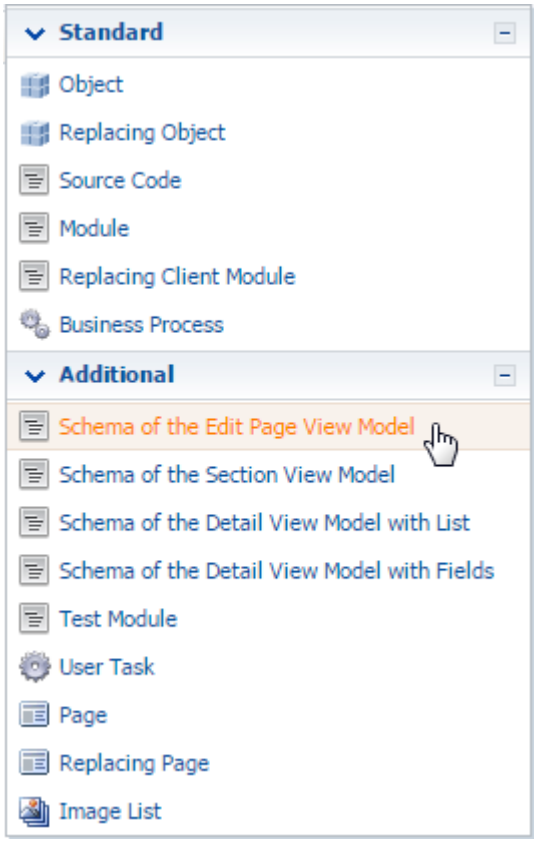
Use the following link to download a package with the [Knowledge base] section pop-up summary schema implemented according to this case.

Case implementation algorithm

1. Creating a pop-up summary view model schema

Open the [Schemas] tab in the [Configuration] section and select the [Add] — [Replacing Client Module] command from the menu (Fig.1).

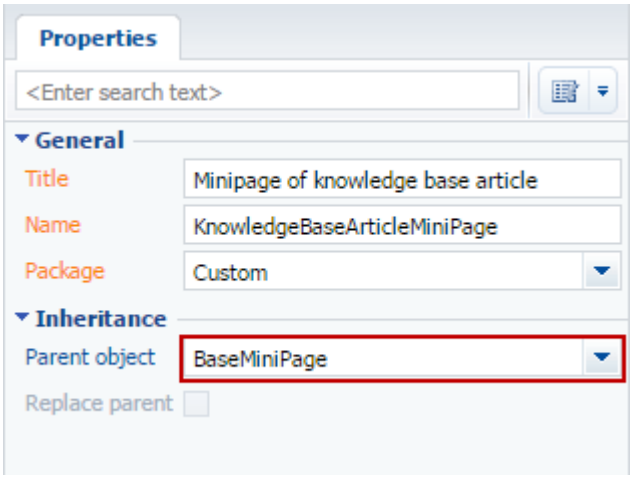
Fig.1 Adding a pop-up summary view schema



Populate the following properties of the pop-up summary view schema (Fig.2):

- [Title] – "UsrKnowledgeBaseArticleMiniPage"
- [Name] – "KnowledgeBase Mini Page"
- [Package] – the custom package, in which the development is performed, for instance, *UsrPackage*
- [Parent object] - the *BaseMiniPage* schema from the *NUI* package

Fig.2 Properties of the pop-up summary view model schema



2. Registering a the pop-up summary in the database

Execute the following SQL query to perfrom modifications in the database:

```
DECLARE
    -- Name of the created pop-up summary view schema.
    @ClientUnitSchemaName NVARCHAR(100) = 'UsrKnowledgeBaseArticleMiniPage',
    -- Name of the object schema bound to the pop-up summary.
    @EntitySchemaName NVARCHAR(100) = 'KnowledgeBase'
```

```
UPDATE SysModuleEdit
SET MiniPageSchemaUId = (
    SELECT TOP 1 UId
    FROM SysSchema
    WHERE Name = @ClientUnitSchemaName
)
WHERE SysModuleEntityId = (
    SELECT TOP 1 Id
    FROM SysModuleEntity
    WHERE SysEntitySchemaUId = (
        SELECT TOP 1 UId
        FROM SysSchema
        WHERE Name = @EntitySchemaName
            AND ExtendParent = 0
    )
);
```

As a result of this query execution you will have a unique pop-up identifier, populated in *SysModuleEdit* table of the record *MiniPageSchemaUId* field that corresponds to the [Knowledge base] section (Fig.3).

Fig.3 Unique pop-up summary identifier value in SysModuleEdit table

CardSchemaUId	ActionKindCaption	ActionKindName	PageCaption	MiniPageSchemaUId
9DBD0611-FA52-4A90-9542-E5FD997B4AFD	New article	KnowledgeBase	Knowledge base article	48A427D4-0BF8-4017-A1CC-4A1F96C39243
NULL				NULL
38B9E577-152C-4EAC-8A68-C296183B690F				NULL
625E1D4C-BC26-4872-B76E-267C473ECD...				NULL

3. Displaying primary object fields

The pop-up summary code structure is identical to the edit page structure. Specify the *KnowledgeBase* schema as the object schema and add the necessary modifications to the *diff* view model modification array.

The base pop-up summary consists of the following elements:

- *MiniPage – Terrasoft.GridLayout* – pop-up summary field
- *HeaderContainer – Terrasoft.Container* – pop-up summary name (initially is placed in the first row of the pop-up summary field

Two objects that configure the [Name] and [Keywords] fields are added to the *diff* modification array.

At this stage the pop-up summary can already be used and the following actions are not required.

Source code of the pop-up summary view model schema:

```
define("UsrKnowledgeBaseArticleMiniPage", [], function() {
    return {
        entitySchemaName: "KnowledgeBase",
        attributes: {
            "MiniPageModes": {
                "value": [this.Terrasoft.ConfigurationEnums.CardOperation.VIEW]
            }
        },
        diff: /**SCHEMA_DIFF*/[
            {
                "operation": "insert",
                "name": "Name",
                "parentName": "HeaderContainer",
                "propertyName": "items",
                "index": 0,
                "values": {
                    "labelConfig": {
                        "visible": false
                    },
```

```
                    "isMiniPageModelItem": true
                }
            },
            {
                "operation": "insert",
                "name": "Keywords",
                "parentName": "MiniPage",
                "propertyName": "items",
                "values": {
                    "labelConfig": {
                        "visible": false
                    },
                    "isMiniPageModelItem": true,
                    "layout": {
                        "column": 0,
                        "row": 1,
                        "colSpan": 24
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

4. Adding a function button to the pop-up summary

As per the example conditions, the pop-up summary must enable downloading files bound to the knowledge base.

You can access additional data via a drop-down list of a pre-configured button. To add a button of selecting files from the knowledge base article:

1. Add the button description to the *diff* array – the *FilesButton* element.
2. Add an attribute binding the primary and additional records – the *Article* virtual column.
3. Add the *MiniPageModes* attribute – the array containing a collection of necessary operations performed by the pop-up summary.
4. Add the button image to bpm'online resources. For example, you can add the following image –  . Adding an image to resources is covered in the "**How to add a field with an image to the edit page**" article.
5. To add methods of working with a drop-down list of a file selection button:
   - override the *init()* method.
   - override the *onEntityInitialized()* method
   - set the *Article* attribute value via the *setArticleInfo()* method
   - get information about the current knowledge base article files via the *initFilesMenu(files)* method
   - populate the drop-down list collection of the file selection button via the *initFilesMenu(files)* method
   - initiate the file upload and adding to the drop-down list of the file selection button via the *fillFilesExtendedMenuData()* method
   - initiate the selected file download via the *downloadFile()* method

5. Applying styling to the pop-up summary.

Create the *UsrKnowledgeBaseArticleMiniPageCss* module and specify the necessary styles on the LESS tab.

```
div[data-item-marker="UsrKnowledgeBaseArticleMiniPageContainer"] > div {
width: 250px;
}
```

Specify the pop-up summary schema dependency on the style module in the page designer and add this module download in the source code.

Below is the full source code of a pop-up summary:

```
define("UsrKnowledgeBaseArticleMiniPage",
```

```
["terrasoft", "KnowledgeBaseFile", "ConfigurationConstants",
"css!UsrKnowledgeBaseArticleMiniPageCss"],
    function(Terrasoft, KnowledgeBaseFile, ConfigurationConstants) {
        return {
            entitySchemaName: "KnowledgeBase",
            attributes: {
                "MiniPageModes": {
                    "value": [this.Terrasoft.ConfigurationEnums.CardOperation.VIEW]
                },
                "Article": {
                    "type": Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                    "referenceSchemaName": "KnowledgeBase"
                }
            },
            methods: {
                // Initiates the drop-down list collection of the file selection
button.
                init: function() {
                    this.callParent(arguments);
                    this.initExtendedMenuButtonCollections("File", ["Article"],
this.close);
                },
                // Initiates the attribute value binding the primary and additional
records.
                // Populates the drop-down list collection of the file selection
button.
                onEntityInitialized: function() {
                    this.callParent(arguments);
                    this.setArticleInfo();
                    this.fillFilesExtendedMenuData();
                },
                // Initiates the file download and adding to the drop-down list of
the file selection button.
                fillFilesExtendedMenuData: function() {
                    this.getFiles(this.initFilesMenu, this);
                },
                // Sets the attribute value binding the primary and additional
records.
                setArticleInfo: function() {
                    this.set("Article", {
                        value: this.get(this.primaryColumnName),
                        displayValue: this.get(this.primaryDisplayColumnName)
                    });
                },
                // Receives information about files of the current knowledge base
article.
                getFiles: function(callback, scope) {
                    var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
                        rootSchema: KnowledgeBaseFile
                    });
                    esq.addColumn("Name");
                    var articleFilter =
this.Terrasoft.createColumnFilterWithParameter(
                        this.Terrasoft.ComparisonType.EQUAL, "KnowledgeBase",
this.get(this.primaryColumnName));
                    var typeFilter = this.Terrasoft.createColumnFilterWithParameter(
                        this.Terrasoft.ComparisonType.EQUAL, "Type",
ConfigurationConstants.FileType.File);
                    esq.filters.addItem(articleFilter);
                    esq.filters.addItem(typeFilter);
                    esq.getEntityCollection(function(response) {
                        if (!response.success) {
```

```
                            return;
                        }
                        callback.call(scope, response.collection);
                    }, this);
                },
                // Populates the drop-down list collection of the file selection
button.
                initFilesMenu: function(files) {
                    if (files.isEmpty()) {
                        return;
                    }
                    var data = [];
                    files.each(function(file) {
                        data.push({
                            caption: file.get("Name"),
                            tag: file.get("Id")
                        });
                    }, this);
                    var recipientInfo = this.fillExtendedMenuItems("File",
["Article"]);
                    this.fillExtendedMenuData(data, recipientInfo,
this.downloadFile);
                },
                // Initiates the selected file download.
                downloadFile: function(id) {
                    var element = document.createElement("a");
                    element.href = "../rest/FileService/GetFile/" +
KnowledgeBaseFile.uId + "/" + id;
                    document.body.appendChild(element);
                    element.click();
                    document.body.removeChild(element);
                }
            },
            diff: /**SCHEMA_DIFF*/[
                {
                    "operation": "insert",
                    "name": "Name",
                    "parentName": "HeaderContainer",
                    "propertyName": "items",
                    "index": 0,
                    "values": {
                        "labelConfig": {
                            "visible": true
                        },
                        "isMiniPageModelItem": true
                    }
                },
                {
                    "operation": "insert",
                    "name": "Keywords",
                    "parentName": "MiniPage",
                    "propertyName": "items",
                    "values": {
                        "labelConfig": {
                            "visible": true
                        },
                        "isMiniPageModelItem": true,
                        "layout": {
                            "column": 0,
                            "row": 1,
                            "colSpan": 24
                        }
```

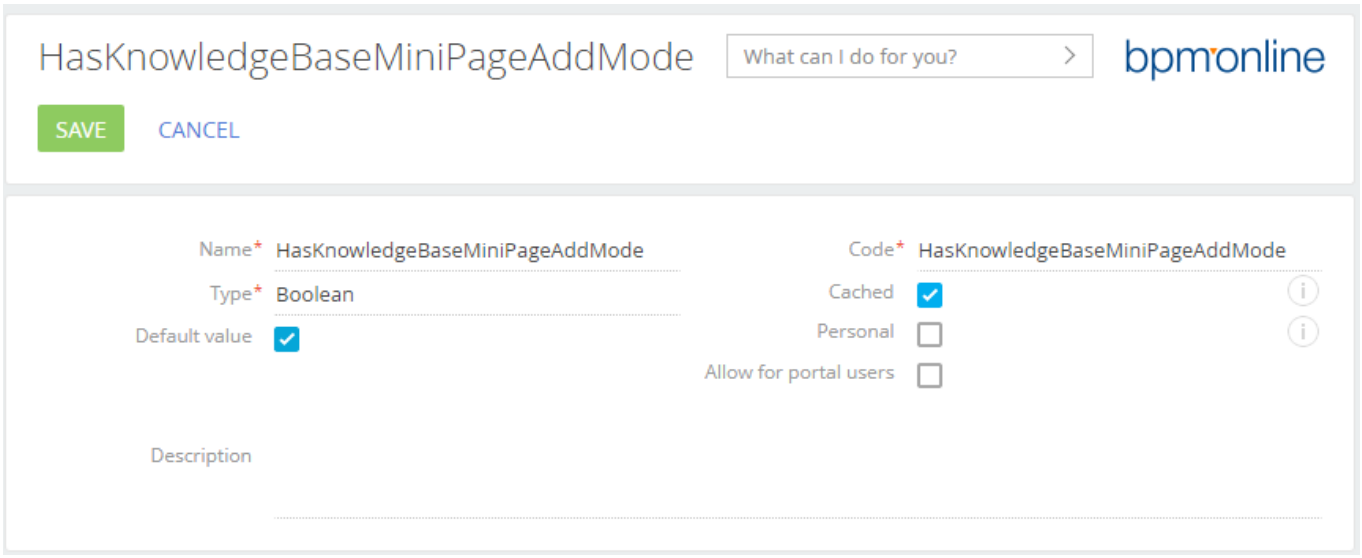
```
                    }
                },
                {
                    "operation": "insert",
                    "parentName": "HeaderContainer",
                    "propertyName": "items",
                    "name": "FilesButton",
                    "values": {
                        "itemType": Terrasoft.ViewItemType.BUTTON,
                        // Button image setup.
                        "imageConfig": {
                            // You need to preliminary add the image to the pop-up
summary resources.
                            "bindTo": "Resources.Images.FilesImage"
                        },
                        // Drop-down list setup.
                        "extendedMenu": {
                            // Drop-down list element name.
                            "Name": "File",
                            // The name of pop-up summary attribute binding the
primary and additional records.
                            "PropertyName": "Article",
                            // Setup of button click handler.
                            "Click": {
                                "bindTo": "fillFilesExtendedMenuData"
                            }
                        }
                    },
                    "index": 1
                }
            ]/**SCHEMA_DIFF*/
        };
    });
```

6. Adding the HasProductMiniPageAddMode system setting

Add a system setting with the following properties to the [System settings] section of the system designer (Fig.4):

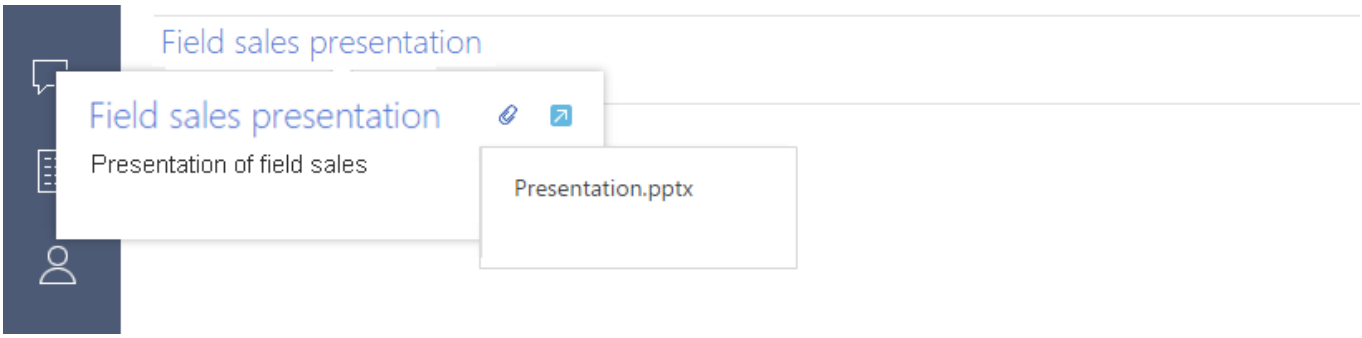
- [Name] – “HasKnowledgeBaseMiniPageAddMode”
- [Code] – “HasKnowledgeBaseMiniPageAddMode”
- [Type] – “Boolean”
- [Default value] – checkbox selected

Fig. 4. System setting

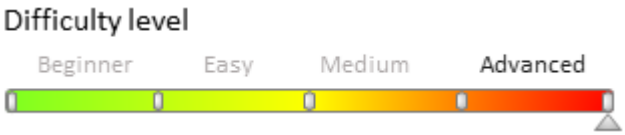


After you save the schema and update the application web-page, a custom pop-up summary containing the record bound files will be displayed when you hover over a name in the [Knowledge base] section. You will be able to download the displayed files (Fig.5).

Fig. 5. Case result



Adding pop-up summaries (mini pages) to a module



Introduction

When adding object pop-up summaries, it sometimes becomes required to connect them to bpm'online modules. **Modules** enable creating links to specific objects in bpm'online. A pop-up summary displayed upon hovering over such a link provides additional information about the object without opening the object section.

In bpm'online base version, an object pop-up summary is connected to the following modules:

- telephony in the communication panel
- email in the communication panel
- notification center in the communication panel
- the [Feed] section in the communication panel
- chart-list in the dashboards section

Case description

Display the current bpm'online user in the application top right corner next to the user profile. Open a pop-up

summary upon hovering over the current bpm'online user link.

Source code

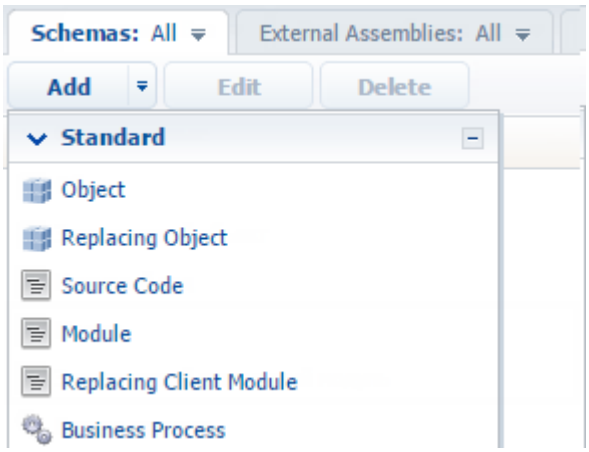
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Creating a module

Perform the [Add] – [Standard] – [Module] menu command on the [Schemas] tab in the [Configuration] section (Fig. 1).

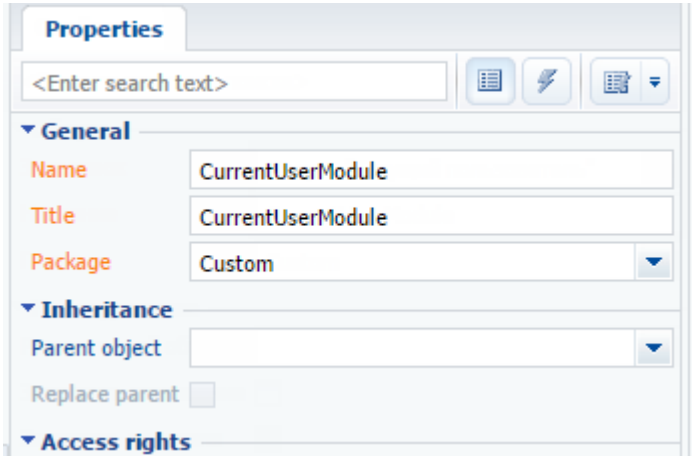
Fig. 1. Adding a module



Specify properties for the created module (Fig. 2):

- [Name] – "UsrCurrentUserModule"
- [Title] – "Current user module"

Fig. 2. Module properties



2. Creating a view and a module view model

To create a view model in the *UsrCurrentUserModule* module, implement the class inherited from *Terrasoft.BaseViewModel*. Connect the *Terrasoft.MiniPageUtilities* utility class to the *mixins* property of the module view model, which enables using the pop-up summary call methods.

To create the view, implement the class inherited from *Terrasoft.BaseModule*.

Override the *init()* and *render()* methods of the *Terrasoft.BaseModule* base class in the created class. The *init()* method initializes the module view model and the *render()* method connects the view model with the view display in container rendered in the *renderTo* parameter. To create the view model, use the *getViewModel()* method. The link

to the received view model is stored in the *viewModel* property.

Define the *getView()* method for receiving the view for its further display. The view must display the full name of the current user and a hyperlink to the contact edit page. When creating a hyperlink, define the event handler of hovering over the mouse cursor.

Below you can find the complete source code:

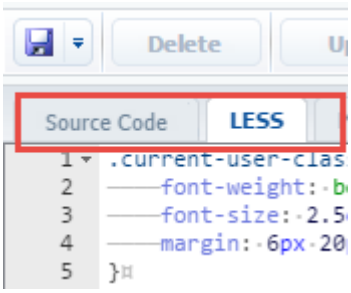
```
// Defining the module.
define("UsrCurrentUserModule", ["MiniPageUtilities"], function() {
    // Defining the CurrentUserViewModel class.
    Ext.define("Terrasoft.configuration.CurrentUserViewModel", {
        // Parent class name.
        extend: "Terrasoft.BaseViewModel",
        // Shortened class name.
        alternateClassName: "Terrasoft.CurrentUserViewModel",
        // Used mixins.
        mixins: {
            MiniPageUtilitiesMixin: "Terrasoft.MiniPageUtilities"
        }
    });
    // Defining the UsrCurrentUserModule class.
    Ext.define("Terrasoft.configuration.UsrCurrentUserModule", {
        // Shortened class name.
        alternateClassName: "Terrasoft.UsrCurrentUserModule",
        // Parent class name.
        extend: "Terrasoft.BaseModule",
        // The Ext object.
        Ext: null,
        // The sandbox object.
        sandbox: null,
        // The Terrasoft object.
        Terrasoft: null,
        // View model.
        viewModel: null,
        // Creates module views.
        getView: function() {
            // Receiving the contact of the current user.
            var currentUser = Terrasoft.SysValue.CURRENT_USER_CONTACT;
            // View — the Terrasoft.Hyperlink class instance.
            return Ext.create("Terrasoft.Hyperlink", {
                // Populating the link caption with the contact name.
                "caption": currentUser.displayValue,
                // Event handler of hovering over the link.
                "linkMouseOver": {"bindTo": "linkMouseOver"},
                // The property containing additional object parameters.
                "tag": {
                    // Current user identifier.
                    "recordId": currentUser.value,
                    // Object schema name.
                    "referenceSchemaName": "Contact"
                }
            });
        },
        // Creates module view model.
        getViewModel: function() {
            return Ext.create("Terrasoft.CurrentUserViewModel");
        },
        // Module initialization.
        init: function() {
            this.viewModel = this.getViewModel();
        },
        // Displays the module view.
```

```
        render: function(renderTo) {
            // Receiving the view object.
            var view = this.getView();
            // Connecting the view with the view model.
            view.bind(this.viewModel);
            // Displaying the view in the renderTo element.
            view.render(renderTo);
        }
    });
    return Terrasoft.UsrCurrentUserModule;
});
```

Add styles to the created module for a better display of the hyperlink. To do this, add the following code to the LESS tab of the module designer:

```
.current-user-class a {
    font-weight: bold;
    font-size: 2.0em;
    margin: 6px 20px;
}

.current-user-class a:hover {
    text-decoration: none;
}
```

Fig. 3. The LESS tab of the module designer



Save the created module.

3. Creating the view display container

To display a link in the user profile in the top right corner of the application, locate the container and download the view of the created module into it. Create a replacing client module that would extend the *MainHeaderSchema* schema functionality implemented in the *NUI* package. The procedure for creating a replacing client module is covered in the "**Creating a custom client module schema**" article.

To display the view, use the *diff* property in the replacing schema source code. To display the container in the top right corner of the page, set the *RightHeaderContainer* element as a parent element of the created container. Override the *onRender()* method and download the created module.

Below you can find the complete source code.

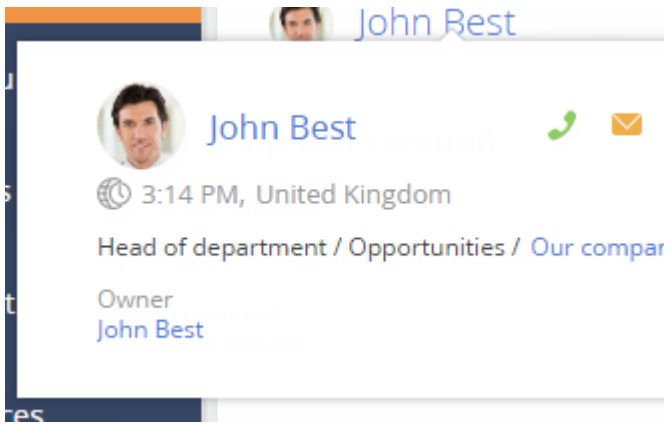
```
// Defining a module.
define("MainHeaderSchema", [], function() {
    return {
        methods: {
            // Performs the action after the view display.
            onRender: function() {
                // Calling the parent method.
                this.callParent(arguments);
                // Downloading the module of the current user.
                this.loadCurrentUserModule();
            },
            // Downloads the module of the current user.
```

```
            loadCurrentUserModule: function() {
                // Receiving the container for downloading the module.
                var currentUserContainer = this.Ext.getCmp("current-user-container");
                // Verifying if a container is available.
                if (currentUserContainer && currentUserContainer.rendered) {
                    // Downloading the module into a container.
                    this.sandbox.loadModule("UsrCurrentUserModule", {
                        // Container name.
                        renderTo: "current-user-container"
                    });
                }
            }
        },
        diff: [
            {
                // Element insert operation.
                "operation": "insert",
                // Element name.
                "name": "CurrentUserContainer",
                // Parent container name.
                "parentName": "RightHeaderContainer",
                // Property name.
                "propertyName": "items",
                // Element values.
                "values": {
                    // Container identifier.
                    "id": "current-user-container",
                    // Element type.
                    "itemType": Terrasoft.ViewItemType.CONTAINER,
                    // Conatiner classes.
                    "wrapClass": ["current-user-class"],
                    // Container elements.
                    "items": []
                }
            }
        ]
    };
});
```

Save the created module.

After you update the application page, the full name with a link to the contact edit page will be displayed in the top right corner. When you hover the cursor over the link, a pop-up summary with the current user details will appear (Fig. 4).

Fig. 4. The contact pop-up summary



Creating a pop-up summary (mini page) for adding records

Difficulty level

Introduction

You can quickly add and view records using bpm’online pop-up summaries. Information about adding pop-up summaries that display record details is available in the “**Creating pop-up summaries (mini pages)**” and “**Adding pop-up summaries (mini pages) to a module**” articles.

To implement a custom pop-up summary page for adding new records in an existing section:

1. Add a pop-up view model schema to the custom package. Select *BaseMiniPage* schema as a parent object.
2. Modify the *SysModuleEdit* system table in the bpm’online database via a special SQL query.
3. Add the necessary pop-up summary functionality to the schema source code.
4. Add the *HasProductMiniPageAddMode* system setting.

> ⚠ NOTE
>
> For bpm’online sections with default pop-up summaries, there are system settings, whose codes have the following format *Has[Section code]MiniPageAddMode* (for instance, *HasAccountMiniPageAddMode*). These system settings are used to toggle between the two modes: adding new records and editing existing records.

Case description

Create a custom pop-up summary page for adding new records in the [Products] section. The pop-up summary must contain a base set of fields: [Name] and [Code].

Source code

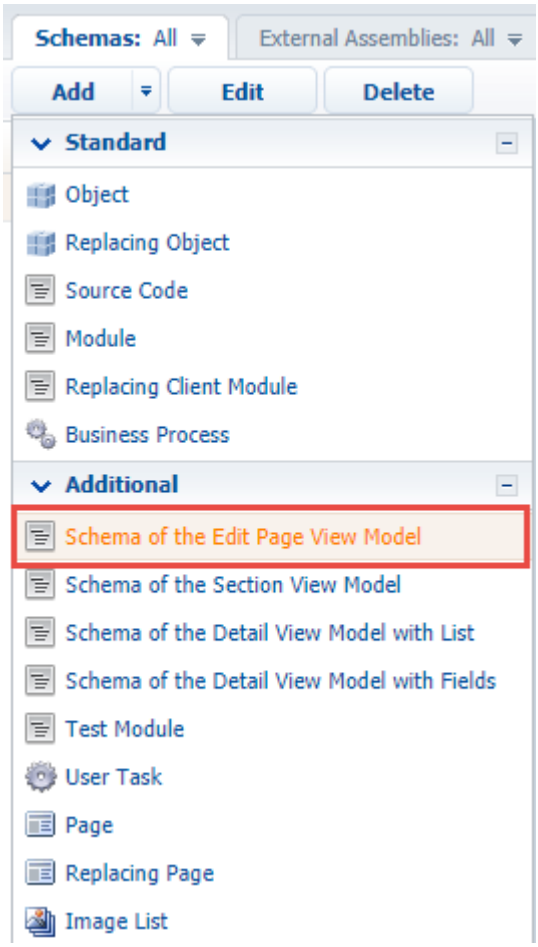
Use the following link to download a package with the [Products] section pop-up summary schema, implemented according to this case.

Case implementation algorithm

1. Create a pop-up summary view model schema

Execute the [Add] — [Additional] — [Schema of the Edit Page View Model] menu command on the [Schemas] tab in the [Configuration] section (Fig.1).

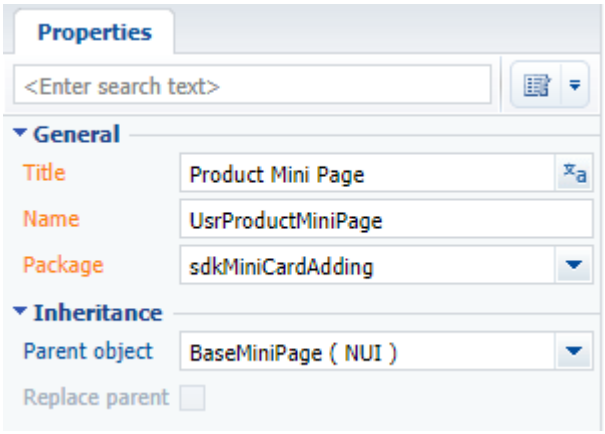
Fig. 1. Adding a pop-up summary view schema



Populate the following properties of the pop-up summary view schema (Fig.2):

- [Name] – "UsrProductMiniPage".
- [Subject] – "Product Mini Page".
- [Package] – the custom package, in which the development is performed, for instance, *Custom*.
- [Parent object] — the *BaseMiniPage* schema from the *NUI* package.

Fig. 2. Properties of the pop-up summary view model schema



2. Register the pop-up summary in the database

Execute the following SQL query to make changes in the database:

```
DECLARE
    -- The name of the created pop-up summary view schema.
    @ClientUnitSchemaName NVARCHAR(100) = 'UsrProductMiniPage',
```

```
    -- The name of the pop-up summary object schema.
    @EntitySchemaName NVARCHAR(100) = 'Product'

UPDATE SysModuleEdit
SET MiniPageSchemaUId = (
    SELECT TOP 1 UId
    FROM SysSchema
    WHERE Name = @ClientUnitSchemaName
)
WHERE SysModuleEntityId = (
    SELECT TOP 1 Id
    FROM SysModuleEntity
    WHERE SysEntitySchemaUId = (
        SELECT TOP 1 UId
        FROM SysSchema
        WHERE Name = @EntitySchemaName
            AND ExtendParent = 0
    )
);
```

As a result of this query execution you will have a unique pop-up identifier, populated in *SysModuleEdit* table of the record *MiniPageSchemaUId* field that corresponds to the [Products] section (Fig.3).

Fig. 3. Unique pop-up summary identifier value in SysModuleEdit table

CardSchemaUId	ActionKindCaption	ActionKindName	PageCaption	MiniPageSchemaUId	SearchRowSchemaUId
0DAEC87E-A84D-44BC-9DD0-C90B8D1BAA33	New product	Product	Product	15D85C82-D78F-400F-B10C-44BB13C72282	NULL

> **ATTENTION**
>
> Since the changes were made directly in the database, log in to your bpm'online again to see them. You may need to compile the application using the corresponding action in the [Configuration] section.

3. Add fields from the primary object to the pop-up summary

Add the source code below to the created pop-up summary view model schema.

```
define("UsrProductMiniPage", ["UsrProductMiniPageResources"],
    function(resources) {
        return {
            entitySchemaName: "Product",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            attributes: {
                "MiniPageModes": {
                    "value": [this.Terrasoft.ConfigurationEnums.CardOperation.ADD]
                }
            },
            diff: /**SCHEMA_DIFF*/[
                {
                    "operation": "insert",
                    "parentName": "MiniPage",
                    "propertyName": "items",
                    "name": "Name",
                    "values": {
                        "isMiniPageModelItem": true,
                        "layout": {
                            "column": 0,
                            "row": 1,
                            "colSpan": 24
                        },
                        "controlConfig": {
                            "focused": true
```

```
                    }
                }
            },
            {
                "operation": "insert",
                "parentName": "MiniPage",
                "propertyName": "items",
                "name": "Code",
                "values": {
                    "isMiniPageModelItem": true,
                    "layout": {
                        "column": 0,
                        "row": 2,
                        "colSpan": 24
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

The array containing the collection of the necessary pop-up summary operations is assigned to *MiniPageModes* attribute, which was declared in the base schema. Two objects that configure the [Name] and [Code] fields are added to the *diff* array.

> NOTE
>
> Add *this.Terrasoft.ConfigurationEnums.CardOperation.VIEW* value to the array assigned to *MiniPageModes* attribute if you also need to display the pop-up summary on the section page (see "**Creating pop-up summaries (mini pages)**").

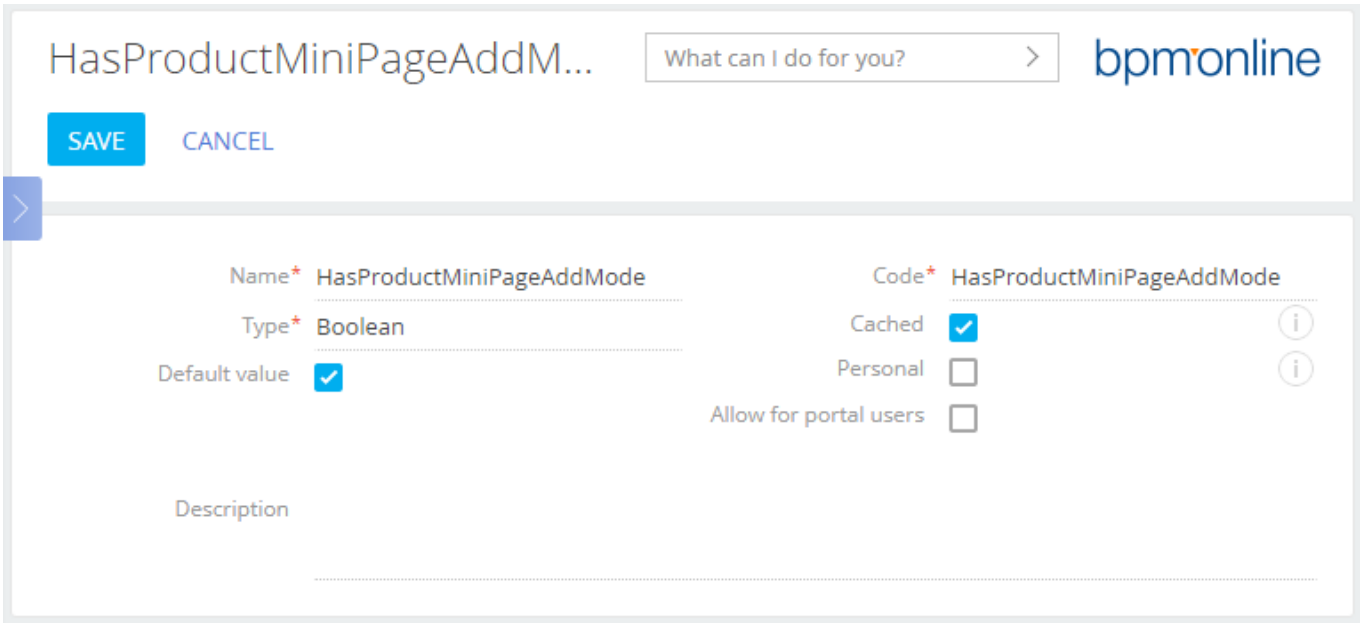
> ATTENTION
>
> If the required columns are not indicated in the *diff* array, they will be displayed at the bottom of the pop-up summary.

4. Add the HasProductMiniPageAddMode system setting

In the [System settings] section of the system designer, add a new system setting with the following properties (fig. 4)

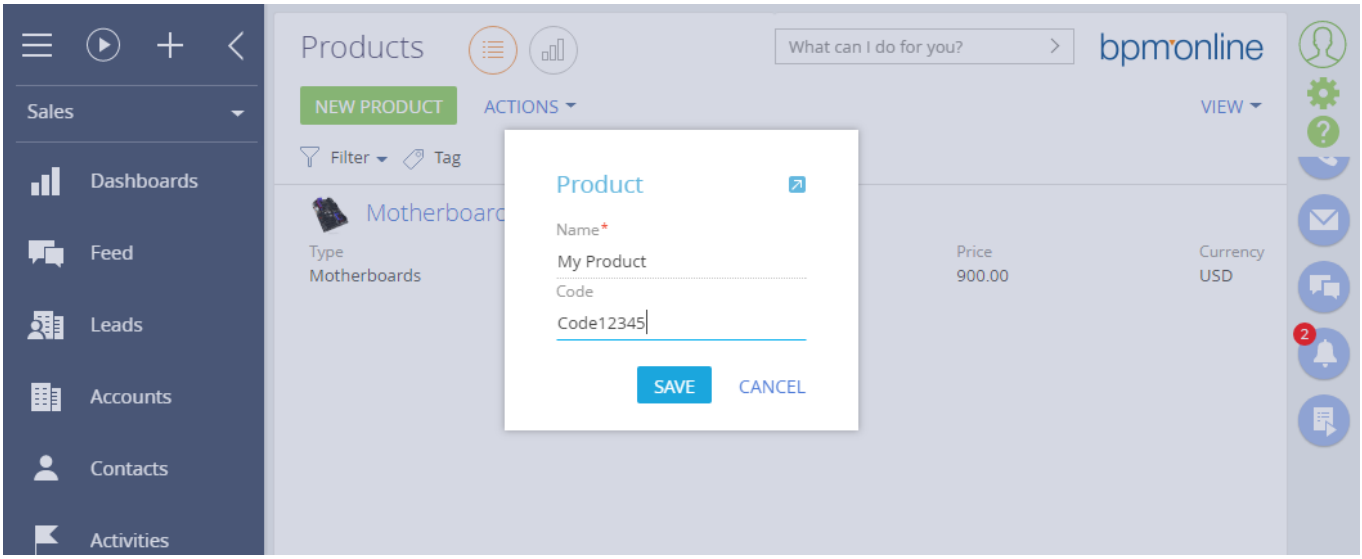
- [Name] — "HasProductMiniPageAddMode".
- [Code] — "HasProductMiniPageAddMode".
- [Type] — "Boolean".
- [Default value] — true.

Fig. 4. — System setting



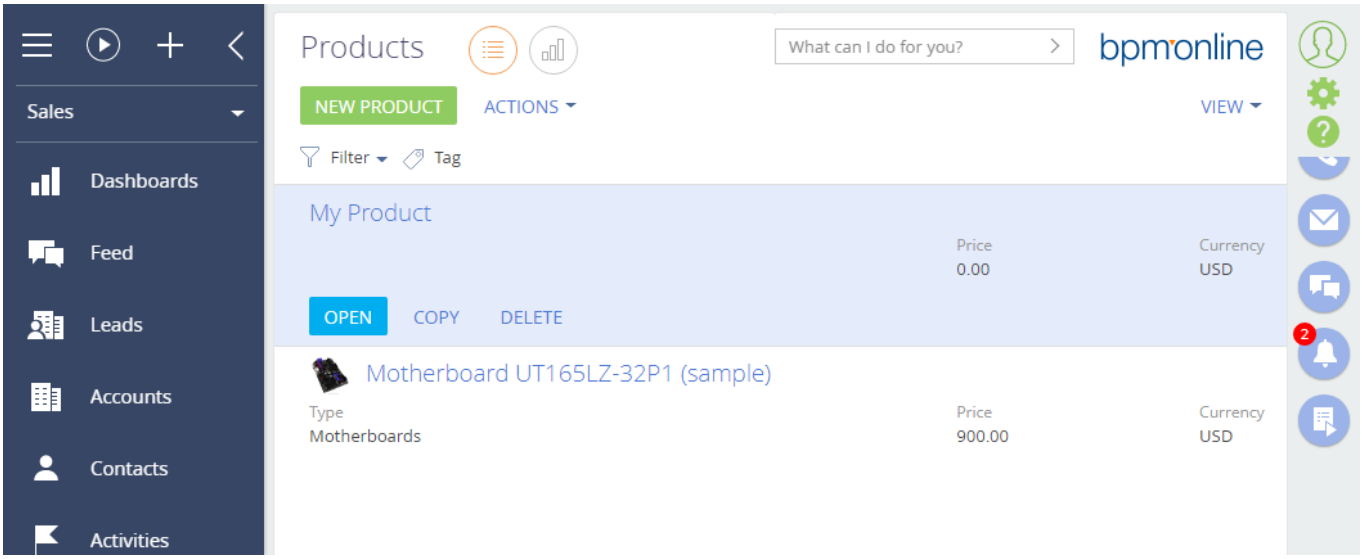
As a result, a pop-up summary with two fields will be displayed when you add a new product (Fig.4).

Fig. 4. Implemented pop-up summary



After saving the pop-up summary, a corresponding record will appear in the section list (Fig.5).

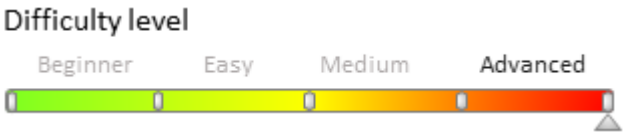
Fig. 5. Records in the [Products] section



ATTENTION

The record will only display in the section list after you update the browser page. To display the record immediately after saving the pop-up summary, add the corresponding functions to pop-up summary and the section page schema via the message mechanism (for more information, see “**Sandbox. Module message exchange**”).

Adding pop-up hints



Introduction

You can add pop-up hints to bpm’online elements - text messages providing additional information about the element functionality and rules of its population.

Pop-up hints can be divided into 3 main groups:

1. A pop-up hint to a field (if such hint is available, the field caption is marked by a small green triangle symbol. The hint appears when a cursor is hovered over the triangle or upon clicking the field caption.

2. A pop-up hint to other control elements (buttons, completion indicators, images). The hint appears when a cursor is hovered over the control element.

3. Information button . The hint appears when a cursor is hovered over the information button.

General algorithm of adding pop-up hints to standard control elements:

1. Create a replacing schema of the page or section.
2. Add the pop-up hint text to the schema localizable string collection.
3. Describe the necessary schema element modifications in the diff array.

Source code

You can download the package with case implementation using the following link.

Example 1

Case description

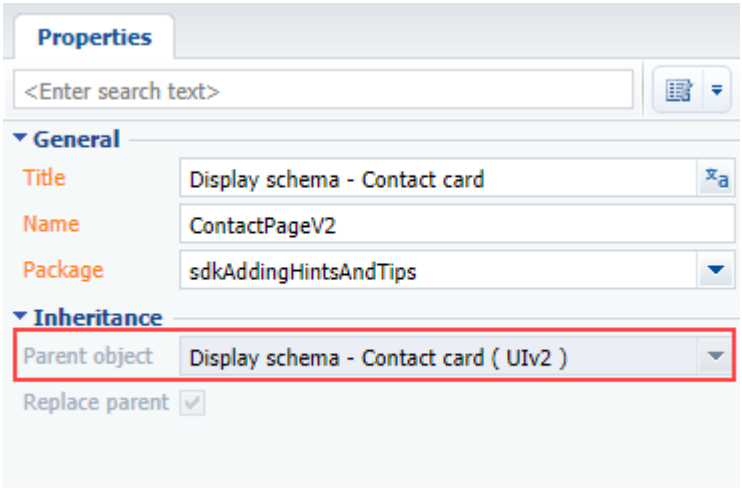
Add a pop-up hint to the [Save] button of contact edit page.

Case implementation algorithm

1. Creating a replacing contact page

Create a replacing client module and specify [Display schema – Contact card] (*ContactPageV2*) as parent object (Fig. 1). Creating a replacing page is covered in the "**Creating a custom client module schema**" article.

Fig. 1. Properties of the replacing contact edit page

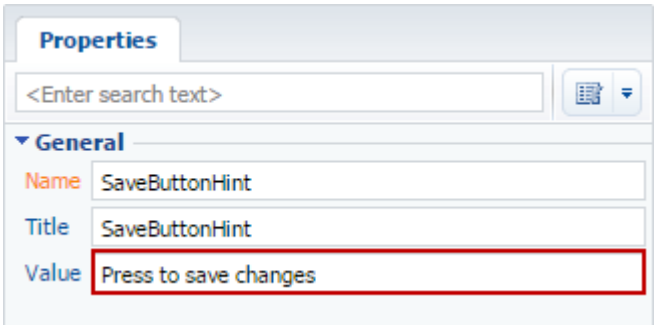


2. Adding a localized string with the pop-up hint text

Add a string with the pop-up hint text to the collection of localizable strings of the edit page replacing schema. Properties of the created string (Fig. 2):

- [Name] – "SaveButtonHint"
- [Value] – "Press to save the changes"

Fig. 2. Properties of the custom localizable string



3. Adding a button configuration object to the *diff* array

There are several methods to add a pop-up hint to a control element.

Method 1

Add the *hint* property containing the pop-up hint text to the control element *values* property.

The source code of the contact edit page replacing schema when adding the pop-up hint using method 1:

```
define("ContactPageV2", [],
function () {
    return {
        // Name of the edit page object schema.
```

```
        entitySchemaName: "Contact",
        //Pop-up hint visualization setup.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding pop-up hint to the button.
            {
                // Modification of the existing element.
                "operation": "merge",
                "parentName": "LeftContainer",
                "propertyName": "items",
                "name": "SaveButton",
                "values": {
                    // Pop-up hint for a button.
                    "hint": { "bindTo": "Resources.Strings.SaveButtonHint" }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

Method 2

Add the *tips* array to control element *values* property. Add the pop-up hint configuration object to the *tips* array using the *insert* operation. Specify the *content* property, which is the pop-up hint text in the *values* property of this object. You can have a more individual approach to pop-up hint setup using this method. You can change the image style, connect the pop-up hint visibility to a view model event, add control elements, etc.

> ⚠ ATTENTION
>
> The specified method works for *itemType*:
>
> - *Terrasoft.ViewItemType.BUTTON*
> - *Terrasoft.ViewItemType.LABEL*
> - *Terrasoft.ViewItemType.COLOR_BUTTON*
> - *Terrasoft.ViewItemType.HYPERLINK*
> - *Terrasoft.ViewItemType.INFORMATION_BUTTON*
> - for elements with the specified *generator* property

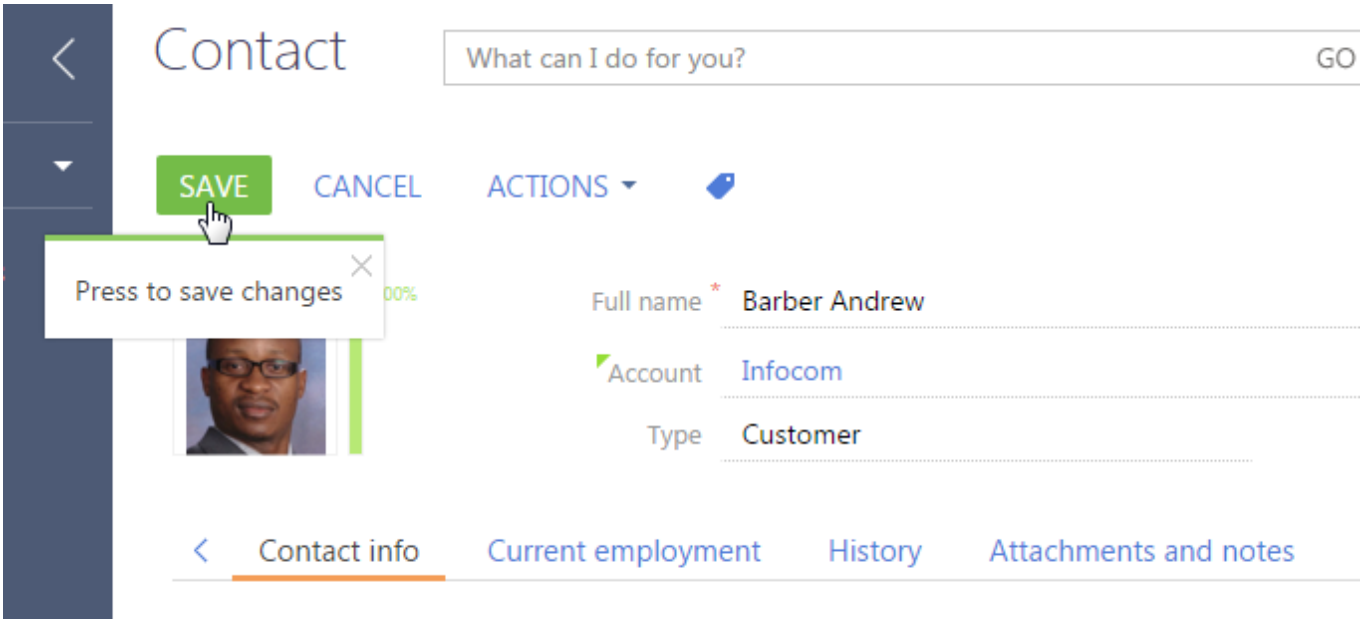
The source code of the contact edit page replacing schema when adding the pop-up hint using method 2:

```
define("ContactPageV2", [],
function () {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Contact",
        //Pop-up hint visualization setup.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding pop-up hint to the button.
            {
                // Modification of the existing element.
                "operation": "merge",
                "parentName": "LeftContainer",
                "propertyName": "items",
                "name": "SaveButton",
                "values": {
                    // Pop-up hint array for a button.
                    "tips": []
                }
            },
            // Simple hint configuration object.
            {
                // Adding a new element.
                "operation": "insert",
```

```
                "parentName": "SaveButton",
                "propertyName": "tips",
                "name": "CustomShowedTip",
                "values": {
                    // Pop-up hint text.
                    "content": {"bindTo": "Resources.Strings.SaveButtonHint"}
                    // You can setup additional parameters of pop-up hint
                    // display and operation.
                }
            },
        ]/**SCHEMA_DIFF*/
    };
});
```

After you save the schema, a pop-up hint will appear next to the [Save] button on the contact edit page (Fig. 3).

Fig. 3. Case result demonstration



Example 2

Case description

Add a pop-up hint to the [Type] field of contact edit page.

Case implementation algorithm

1. Create a replacing contact page

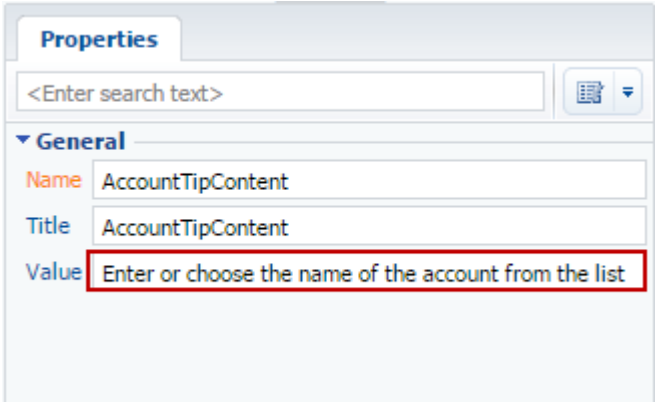
Create a replacing client module and specify [Display schema – Contact card] (*ContactPageV2*) as parent object (Fig. 1).

2. Adding a localized string with the pop-up hint text

Add a string with the pop-up hint text to the collection of localizable strings of the edit page replacing schema. Properties of the created string (Fig. 4):

- [Name] – "TypeTipContent"
- [Value] – 'Choose the type of contact from the list"

Fig. 4. Properties of the custom localizable string



3. Adding a field configuration object to the *diff* array

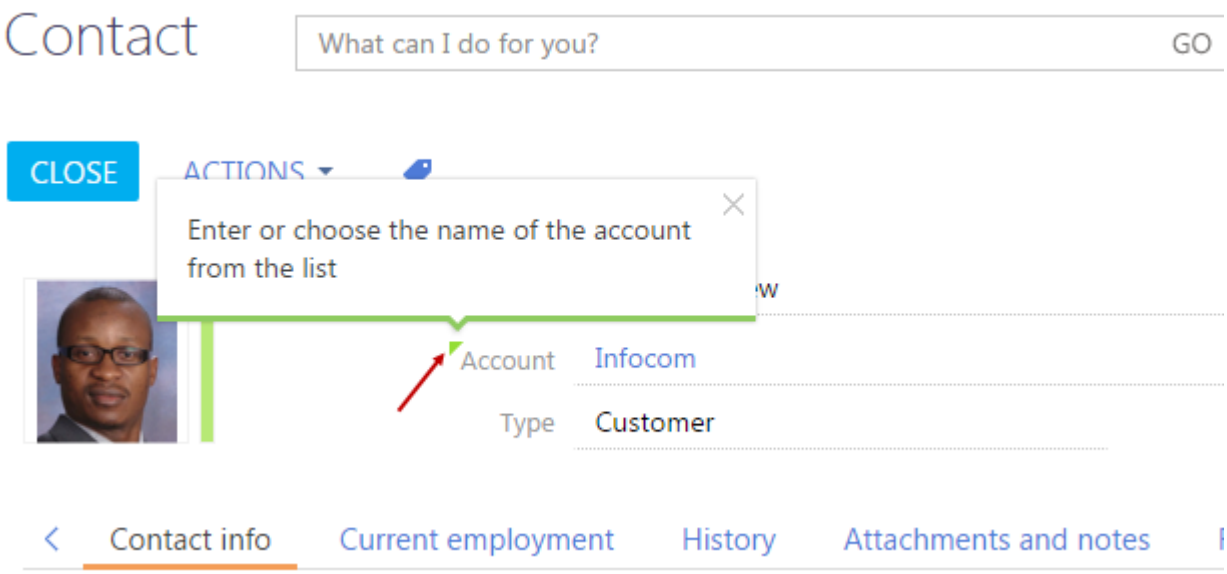
Add the *tip* property containing the *content* property to the *values* field property. The *content* property value will be the pop-up hint text.

Below is the source code of the page replacing schema.

```
define("ContactPageV2", [],
function () {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Contact",
        //Pop-up hint visualization setup.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding pop-up hint to the field.
            {
                // Modification of the existing element.
                "operation": "merge",
                "name": "Type",
                "parentName": "ContactGeneralInfoBlock",
                "propertyName": "items",
                "values": {
                    // Field propery responsible for displaying the pop-up hint.
                    "tip": {
                        // Pop-up hint text.
                        "content": { "bindTo": "Resources.Strings.TypeTipContent" },
                        // Pop-up hint display mode.
                        // WIDE is the default mode - thickness of a green band,
                        // displayed in the pop-up hint.
                        "displayMode": Terrasoft.controls.TipEnums.displayMode.WIDE
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

After you save the schema, a pop-up hint will appear in the [Type] field on the contact edit page (Fig. 5).

Fig. 5. Case result demonstration



Example 3

Case description

Add an information button to the [Full name] contact edit page.

Case implementation algorithm

1. Create a replacing contact page

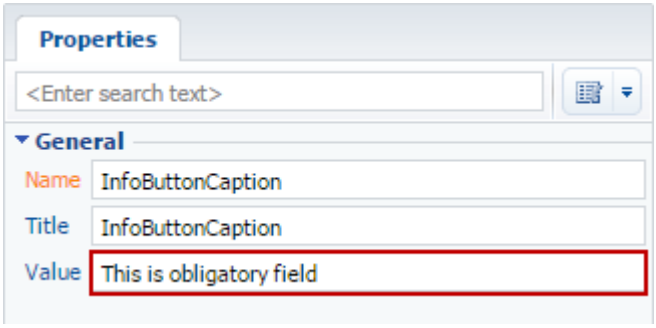
Create a replacing client module and specify [Display schema – Contact card] (*ContactPageV2*) as parent object (Fig. 1).

2. Adding a localized string with the pop-up hint text

Add a string with the pop-up hint text to the collection of localizable strings of the edit page replacing schema. Properties of the created string (Fig. 6):

- [Name] – "InfoButtonCaption"
- [Value] – "This is obligatory field"

Fig. 6. Properties of the custom localizable string



3. Adding a button configuration object to the diff array

Add a new element with the *Terrasoft.ViewItemType.INFORMATION_BUTTON* type and the *content* property to the *diff* array. The *content* property value will be the pop-up hint text.

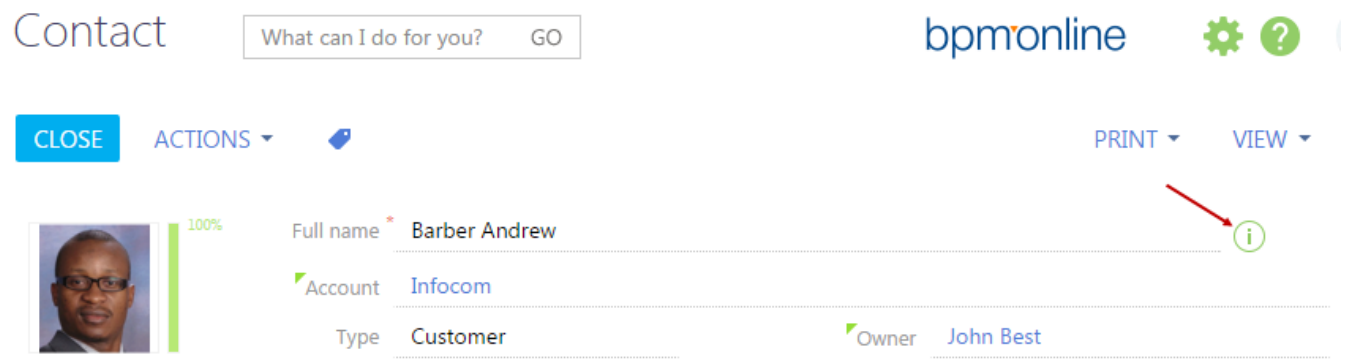
The source code of the edit page replacing schema:

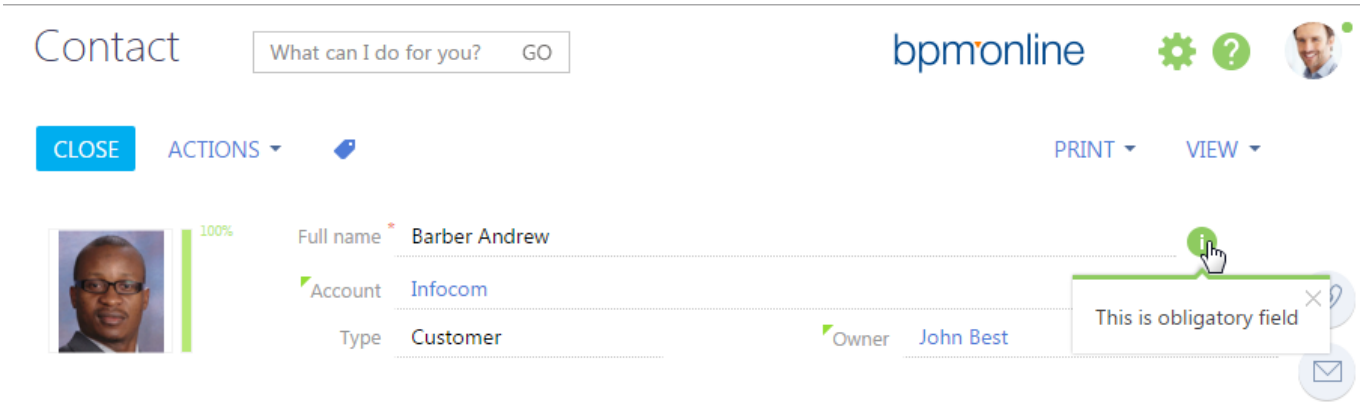
```
define("ContactPageV2", [],
```

```
function () {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Contact",
        //Pop-up hint visualization setup.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding pop-up hint to the button.
            {
                // Modification of the existing element.
                "operation": "merge",
                "parentName": "ProfileContainer",
                "propertyName": "items",
                "name": "AccountName",
                "values": {
                    "layout": { "column": 0, "row": 1, "colSpan": 22, "rowSpan": 1 }
                }
            },
            {
                // Adding a new element.
                "operation": "insert",
                "parentName": "ProfileContainer",
                "propertyName": "items",
                "name": "SimpleInfoButton",
                "values": {
                    "layout": { "column": 22, "row": 1, "colSpan": 1, "rowSpan": 1 },
                    "itemType": Terrasoft.ViewItemType.INFORMATION_BUTTON,
                    "content": { "bindTo": "Resources.Strings.InfoButtonCaption" }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

After you save the schema, the pop-up hint will appear in the [Account] field on the contact edit page (Fig. 7).

Fig. 7. Case result demonstration

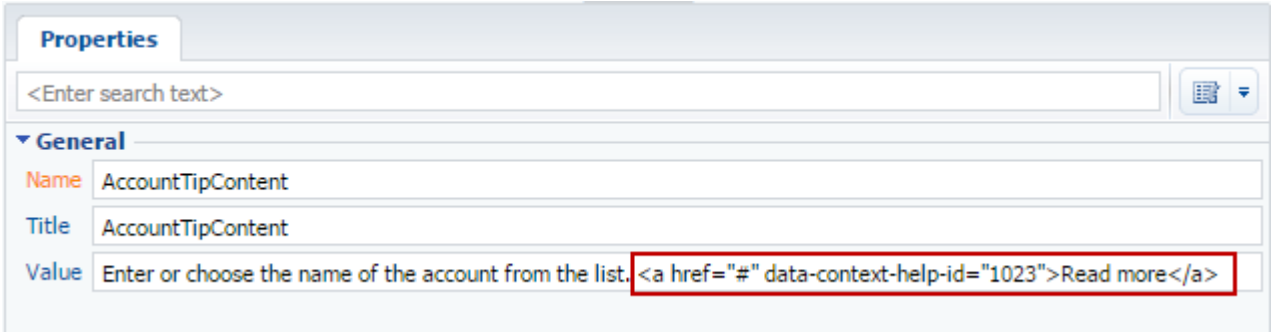




Example 4. Adding a link to web resource to the pop-up hint

You can add links to web resources or context help to pop-up hints. Add an html code of the link directly to the localizable string of the pop-up hint text (Fig. 8).

Fig. 8. Example of defining the pop-up hint with a link

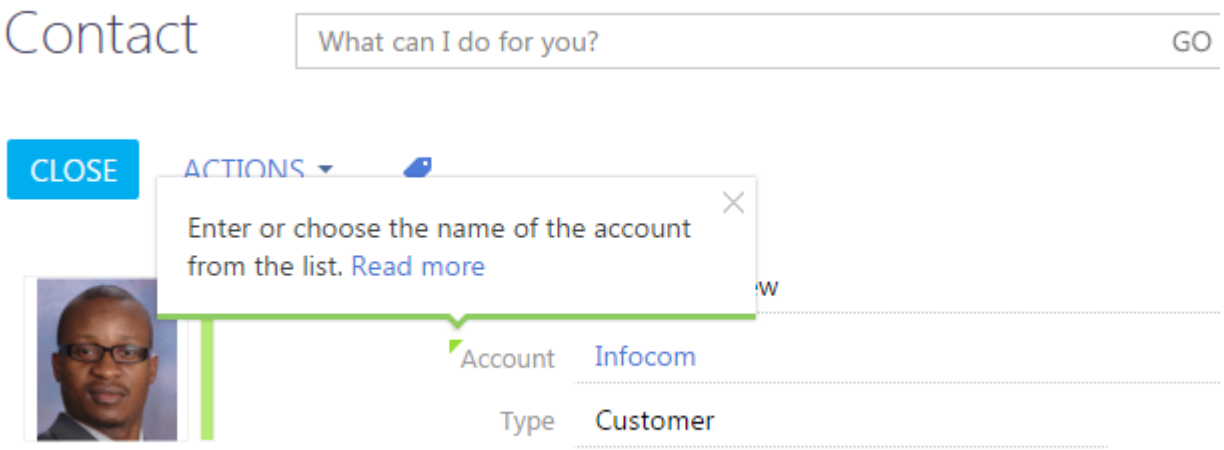


Example of adding a direct link to web resource:

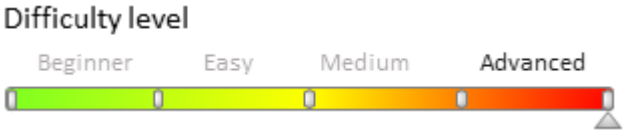
Learn more

As a result, the pop-up hint will look as shown in figure 9.

Fig. 9. Example of displaying the pop-up hint with a link



How to modify sales pipeline calculations



You can modify the way values are calculated for the sales pipeline dashboard element in the [Opportunities section]. To do this, you need to create a new module for calculations and replace the sales pipeline display client schema.

To modify the sales pipeline calculations:

1. Create a new class inherited from *FunnelBaseDataProvider* and specify the calculation logic.
2. Create a replacing *FunnelChartSchema* client schema and use the new calculation class in it.

Example of modifying the calculations displayed in the "Number of opportunities" view of the sales pipeline

Case description

Modify the sales pipeline calculation algorithms by replacing the number of opportunities with the number of products added to opportunities.

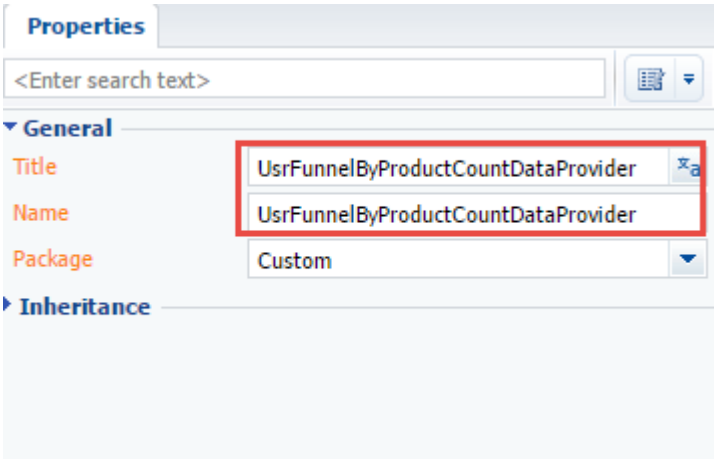
Case implementation algorithm

1. Create a new module in the custom package

Create a new calculation provider client module in the custom package. *Calculation provider* is a class responsible for selecting, filtering and processing data for sales pipeline chart.

Specify a name and caption for the new module, for example, *UsrFunnelByProductCountDataProvider* (Fig. 1).

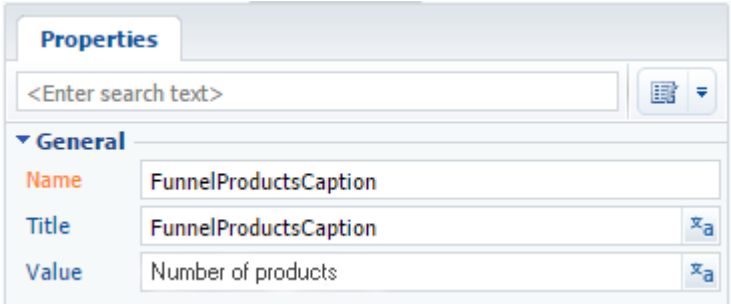
Fig. 1. Calculation provider module properties



2. Add localizable strings

Add a string with the *Number of products* value to the collection of localizable strings of the created module. To do this, right-click the [LocalizableStrings] structure node and select [Add] from the context menu. Set the properties for the new string as shown on Fig. 2.

Fig. 2. Localizable string properties



Add *CntOpportunity* localizable string with the *Number of opportunities* value in the similar way.

3. Add implementation to the provider module

To modify sales pipeline calculations, override the following methods:

- *addQueryColumns* column generation method for data selection
- methods for selection data processing.

To process one record from the selection, define the *getSeriesDataConfigByItem* method. To process the whole collection, define the *prepareFunnelResponseCollection* method. To filter the records, define the *applyFunnelPeriodFilters* method.

Below is the source code of the new calculation provider module for the sales pipeline.

```
define("UsrFunnelByProductCountDataProvider", ["ext-base", "terrasoft",
"UsrFunnelByProductCountDataProviderResources",
        "FunnelBaseDataProvider"],
    function(Ext, Terrasoft, resources) {
        // Defining a new calculation provider.
        Ext.define("Terrasoft.configuration.UsrFunnelByProductCountDataProvider", {
            // Inheriting from the basic provider.
            extend: "Terrasoft.FunnelBaseDataProvider",
            // New provider short name
            alternateClassName: "Terrasoft.UsrFunnelByProductCountDataProvider",
            // Collection processing method
            prepareFunnelResponseCollection: function(collection) {
                this.callParent(arguments);
            },
            // Extending the FunnelBaseDataProvider base model method.
            // Sets the column number of products for data sampling
            addQueryColumns: function(entitySchemaQuery) {
                // Parent method calling
                this.callParent(arguments);
                // Adds the number of products column to the sample
                entitySchemaQuery.addAggregationSchemaColumn("
[OpportunityProductInterest:Opportunity].Quantity",
                        Terrasoft.AggregationType.SUM, "ProductsAmount");
            },
            // Extending the FunnelBaseDataProvider base class method.
            // Sets sample filtration
            applyFunnelPeriodFilters: function(filterGroup) {
                // Parent method calling
                this.callParent(arguments);
                // Creates a filter group.
                var endStageFilterGroup = Terrasoft.createFilterGroup();
                // Sets the group operator type.
                endStageFilterGroup.logicalOperation =
Terrasoft.LogicalOperatorType.OR;
                // Sets the filter that shows whether the sale stage is over yet.
                endStageFilterGroup.addItem(

Terrasoft.createColumnIsNullFilter(this.getDetailColumnPath("DueDate")));
```

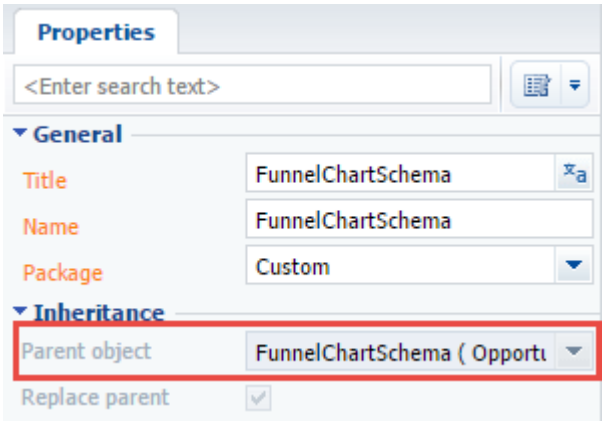
```
                // Sets the filter that shows whether the sale stage is final.
                endStageFilterGroup.addItem(

Terrasoft.createColumnFilterWithParameter(Terrasoft.ComparisonType.EQUAL,
                        this.getDetailColumnPath("Stage.End"), true,
Terrasoft.DataValueType.BOOLEAN));
                filterGroup.addItem(endStageFilterGroup);
            },
            // Extending the FunnelBaseDataProvider base model method.
            // Processes data for the stages in the pipeline.
            getSeriesDataConfigByItem: function(responseItem) {
                // Object that stores localizable strings.
                var lcz = resources.localizableStrings;
                // Receives a stage data object from the parent method.
                var config = this.callParent(arguments);
                // Receives data about the number of products in an opportunity from
the sample result.
                var products = responseItem.get("ProductsAmount");
                products = Ext.isNumber(products) ? products : 0;
                // Formats the strings.
                var name = Ext.String.format("{0}<br/>{1}: {2}<br/>{3}: {4}",
                    config.menuHeaderValue, lcz.CntOpportunity, config.y,
lcz.FunnelProductsCaption, products);
                var displayValue = Ext.String.format("<br/>{0}: {1}",
lcz.FunnelProductsCaption, products);
                // Installs new data in the data object and returns it.
                return Ext.apply(config, {
                    name: name,
                    displayValue: displayValue
                });
            }
        });
    });
```

4. Create a sales pipeline replacing schema

To use the new provider module in the calculations, override the sales pipeline calculation provider generator method.

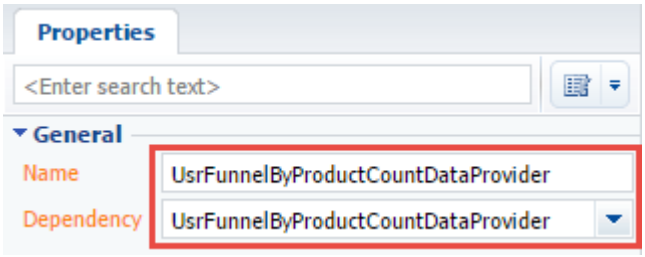
To do this, create a replacing client module and specify *FunnelChartSchema* as a parent (Fig. 3).

Fig. 3. Properties of the replacing module



Add the new calculation module to dependencies (the *Dependencies* section), by specifying its name in the [Dependency] field and the *UsrFunnelByProductCountDataProvider* value in the [Name] field (Fig. 4).

Fig. 4. Sales pipeline schema dependency properties



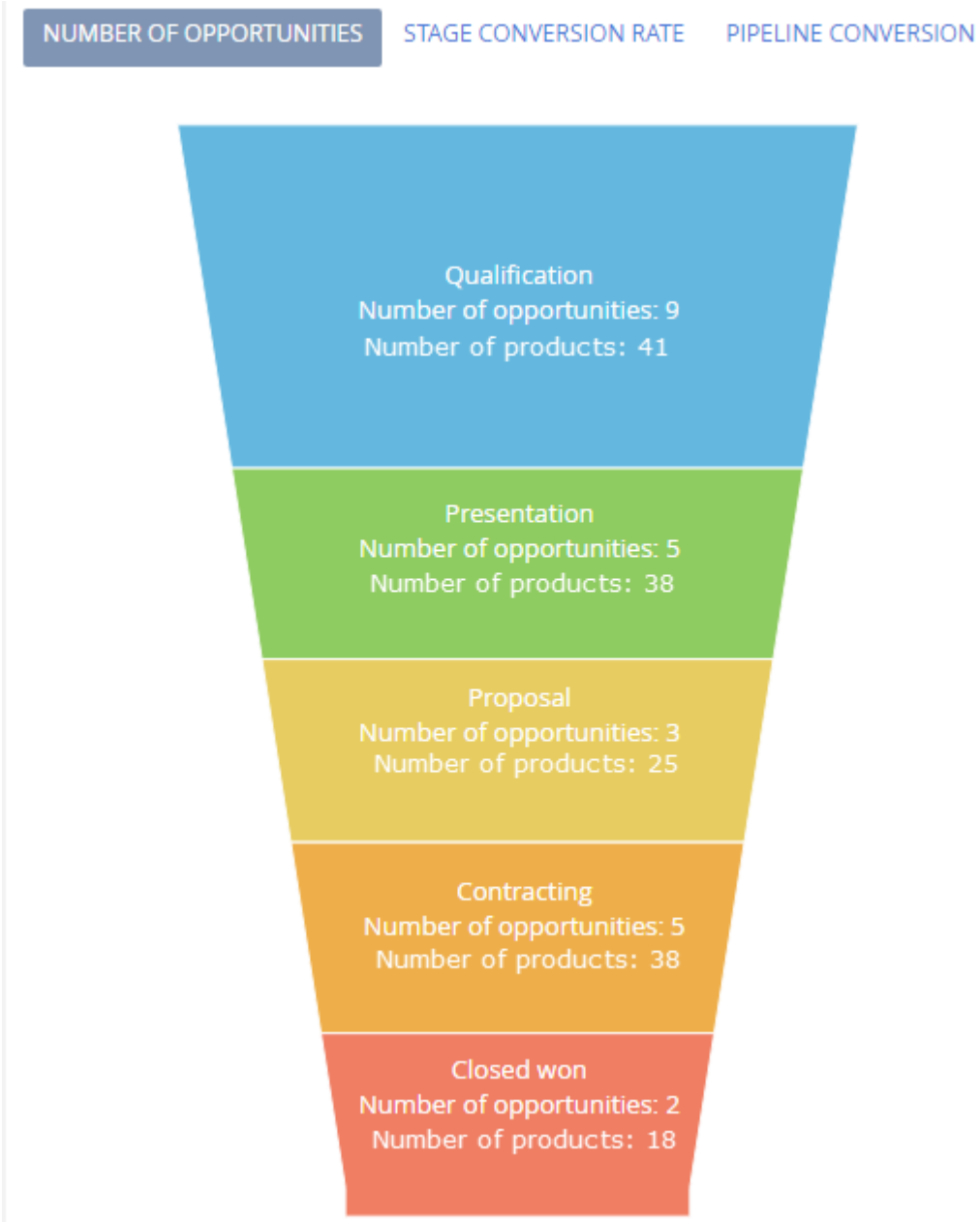
5. Specify the new calculation provider in the sales pipeline replacing schema

To do this, override the getProvidersCollectionConfig method in the replacing schema that gets the configuration object with the collection of providers.

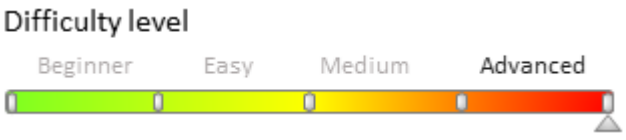
```
define("FunnelChartSchema", ["UsrFunnelByProductCountDataProvider"],
    function() {
        return {
            entitySchemaName: "Opportunity",
            methods: {
                getProvidersCollectionConfig: function() {
                    // Calls parent method.
                    // Gets array of providers.
                    var config = this.callParent();
                    // Searches data provider in the measurement by the number of
opportunities.
                    var byCount = Terrasoft.findItem(config, {tag:
"byNumberConversion"});
                    // Replaces with new class.
                    byCount.item.className =
"Terrasoft.UsrFunnelByProductCountDataProvider";
                    return config;
                }
            }
        };
    });
```

After saving the schema, the new calculation module will be used in the sales pipeline and the sales pipeline itself will display the total number of products by stages (Fig. 5).

Fig. 5. Sales pipeline displaying the number of products added to opportunities



How to enable additional filtering in a sales pipeline



Introduction

In bmp’online, you can enable additional filtering for calculations in sales pipeline charts.

To do it this:

1. Create a new class inherited from the calculation provider and implement the necessary filtering logic.
2. Create a replacing FunnelChartSchema client schema and use the new calculation class in it.

Case description

Add filtering to sales pipeline calculations displayed in the "Number of opportunities" view for selecting the opportunities whose [Customer] field is populated with an account.

Source code

You can download the package with case implementation using the following link.

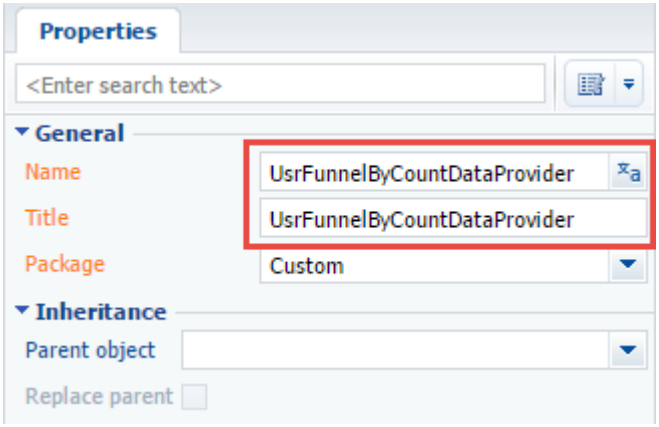
Case implementation algorithm

1. Creating a new module in the custom package

Create a new calculation provider client module in the custom package. *Calculation provider* is a class responsible for selecting, filtering and processing data for sales pipeline chart.

Specify a name and caption for the new module, for example, *UsrFunnelByCountDataProvider* (fig. 1).

Fig. 1. Calculation provider module properties



2. Defining the new provider class and specifying the filtering logic

Inherit the created class from the *FunnelByCountDataProvider* class and override the *getFunnelFixedFilters* method.

The module source code:

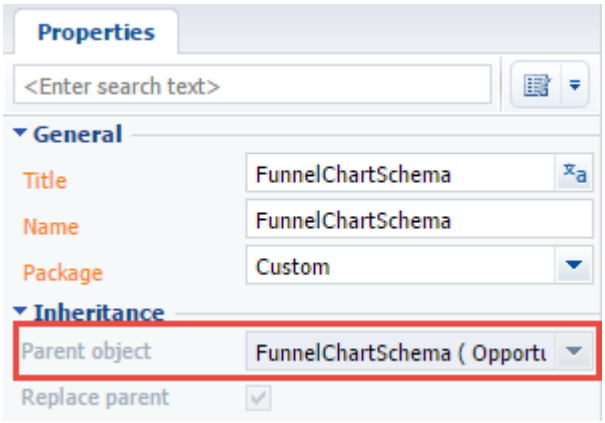
```
define("UsrFunnelByCountDataProvider", ["ext-base",
    "terrasoft", "UsrFunnelByCountDataProviderResources",
    "FunnelByCountDataProvider"],
    function(Ext, Terrasoft, resources) {
        // Defining the new calculation provider.
        Ext.define("Terrasoft.configuration.UsrFunnelByCountDataProvider", {
            // Inheritance from the provider "by number".
            extend: "Terrasoft.FunnelByCountDataProvider",
            // Contracted name of the new provider.
            alternateClassName: "Terrasoft.UsrFunnelByCountDataProvider",
            // Extending the FunnelByCountDataProvider base module method.
            // Returns filter for selection.
            getFunnelFixedFilters: function() {
                // Calling the parent method.
                var esqFiltersGroup = this.callParent(arguments);
                // Adds filter specifying that the customer of an opportunity is an
account.
                esqFiltersGroup.addItem(
                    Terrasoft.createColumnIsNotNullFilter("Account"));
                return esqFiltersGroup;
            }
        });
    });
```

Save the module.

3. Implementing the pipeline chart module in custom package

To use the new provider module in calculations, cerate a replacing client module and specify *FunnelChartSchema* from the *Opportunity* package as a parent schema (fig. 2).

Fig. 2. Properties of the replacing module



4. Specify the new calculation provider in the sales pipeline replacing schema

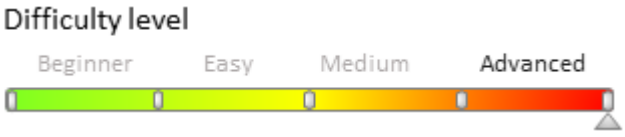
Override the provider generator method of sales pipeline calculation in the replacing schema and specify the new provider class for calculations.

The replacing schema source code is as follows:

```
define("FunnelChartSchema", ["UsrFunnelByCountDataProvider"], function() {
    return {
        entitySchemaName: "Opportunity",
        methods: {
            getProvidersCollectionConfig: function() {
                // Calls parent method returning the provider array.
                var config = this.callParent();
                // Searches for data provider for displaying in the "Number of
opportunities" view.
                var byCount = Terrasoft.findItem(config, {tag:
"byNumberConversion"});
                // Changes for a new class.
                byCount.item.className = "Terrasoft.UsrFunnelByCountDataProvider";
                return config;
            }
        }
    };
});
```

After you save the schema, the new calculation module will be used in the sales pipeline. It will display the opportunities whose [Customer] field is populated with an account.

Adding a custom dashboard widget



Introduction

Dashboard widgets (analytic elements) are used for data analysis of sections. Go to the “Dashboards” view of the required section to work with its analytics. Use the [Dashboards] section to work with the entirety of bpm’online section data analytics.

To learn more about bpm’online dashboard widgets, please refer to the “**Section analytics**” article.

You can create custom dashboard widgets in bpm'online.

Creating a custom widget

To create a custom widget you need to:

1. Create new or select the existing module. Custom module must be an inheritor of the *BaseNestedModule* module or one of its inheritors: *ChartModule*, *IndicatorModule*, *GaugeModule*, etc. More information about dashboard widget modules can be found in the “**Dashboard widgets**” article.
2. Add the source code that implements the necessary functionality to the created module.
3. Specify the module dependency in the [Dependencies] block of the module properties. Add messages that are used.
4. Set the widget parameters in the [Module parameters] field when adding widgets on the dashboards panel. More information about parameters can be found in the “**Dashboard widgets**” article.

Case description

Create custom widget that shows currency exchange rate.

Source code

Use this link to download the case implementation package.

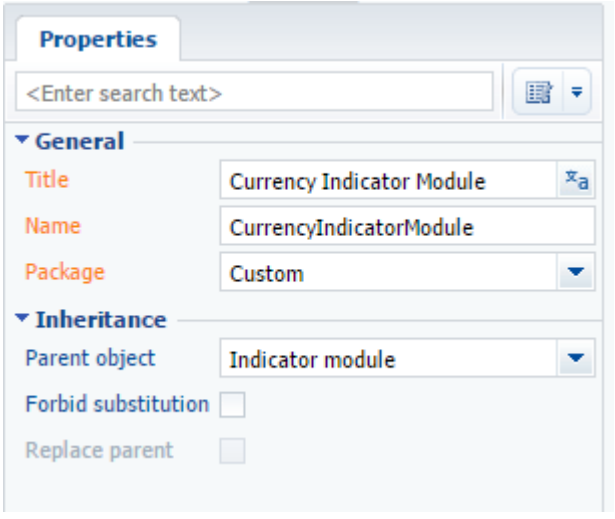
Case implementation algorithm

1. Create a currency indicator module.

Go to the [Configuration] section in the system designer and on the [Schemas] tab, select [Add] -> [Standard] -> [Module] command. For the created module specify (Fig. 1):

- [Parent object] – [Indicator module]
- [Name] – “UsrCurrencyIndicatorModule”.
- [Title] – “Currency Indicator Module”.

Fig. 1. Currency indicator module properties



2. Add the source code

The module source code:

```
define("UsrCurrencyIndicatorModule", ["UsrCurrencyIndicatorModuleResources",
"IndicatorModule"], function() {

    // Class that generates the configuration of the currency indicator module view..
    Ext.define("Terrasoft.configuration.CurrencyIndicatorViewConfig", {
        extend: "Terrasoft.BaseModel",
        alternateClassName: "Terrasoft.CurrencyIndicatorViewConfig",
        // Generates the configuration of the currency indicator module view.
        generate: function(config) {
            var style = config.style || "";
            var fontStyle = config.fontStyle || "";
            var wrapClassName = Ext.String.format("{0}", style);
            var id = Terrasoft.Component.generateId();
            // The returned configuration view object.
            var result = {
                "name": id,
                "itemType": Terrasoft.ViewItemType.CONTAINER,
                "classes": {wrapClassName: [wrapClassName, "indicator-module-
wrapper"]},
                "styles": {
                    "display": "table",
                    "width": "100%",
                    "height": "100%"
                },
                "items": [
                    {
                        "name": id + "-wrap",
                        "itemType": Terrasoft.ViewItemType.CONTAINER,
                        "styles": {
                            "display": "table-cell",
                            "vertical-align": "middle"
                        },
                        "classes": {wrapClassName: ["indicator-wrap"]},
                        "items": [
                            // Display the name of the currency.
                            {
                                "name": "indicator-caption" + id,
                                "itemType": Terrasoft.ViewItemType.LABEL,
                                "caption": {"bindTo": "CurrencyName"},
                                "classes": {"labelClass": ["indicator-caption"]}
                            },
                            // Display the currency exchange rate.
                            {
                                "name": "indicator-value" + id,
                                "itemType": Terrasoft.ViewItemType.LABEL,
                                "caption": {
                                    "bindTo": "CurrencyValue"
                                },
                                "classes": {"labelClass": ["indicator-value " +
fontStyle]}
                            }
                        ]
                    }
                ]
            };
            return result;
        }
    });

    // Class of the view model of the currency indicator module.
```

```
    Ext.define("Terrasoft.configuration.CurrencyIndicatorViewModel", {
        extend: "Terrasoft.BaseModel",
        alternateClassName: "Terrasoft.CurrencyIndicatorViewModel",
        Ext: null,
        Terrasoft: null,
        sandbox: null,
        columns: {
            // Currency name.
            CurrencyName: {
                type: Terrasoft.core.enums.ViewModelSchemaItem.ATTRIBUTE,
                dataValueType: Terrasoft.DataValueType.TEXT,
                value: null
            },
            // Currency value.
            CurrencyValue: {
                type: Terrasoft.core.enums.ViewModelSchemaItem.ATTRIBUTE,
                dataValueType: Terrasoft.DataValueType.FLOAT,
                value: null
            }
        },
        onRender: Ext.emptyFn,
        // Returns the currency value, depending on the name. This method is given as
an example.
        // For each specific task, you should select an individual method to obtain
data,
        // for example REST API, database query, etc.
        getCurrencyValue: function(currencyName, callback, scope) {
            var result = 0;
            if (currencyName === "USD") {
                result = 26;
            }
            if (currencyName === "EUR") {
                result = 32.3;
            }
            if (currencyName === "RUB") {
                result = 0.45;
            }
            callback.call(scope || this, result);
        },
        // Gets the data and displays them on the widget.
        prepareIndicator: function(callback, scope) {
            this.getCurrencyValue(this.get("CurrencyName"), function(currencyValue) {
                this.set("CurrencyValue", currencyValue);
                callback.call(scope);
            }, this);
        },
        // Initializes the widget.
        init: function(callback, scope) {
            this.prepareIndicator(callback, scope);
        }
    });

    // Widget module class.
    Ext.define("Terrasoft.configuration.CurrencyIndicatorModule", {
        extend: "Terrasoft.IndicatorModule",
        alternateClassName: "Terrasoft.CurrencyIndicatorModule",
        // The name of the wdget view model class.
        viewModelClassName: "Terrasoft.CurrencyIndicatorViewModel",
        // The name of the view configuration generating class.
        viewConfigClassName: "Terrasoft.CurrencyIndicatorViewConfig",
        // Subscribing to messages from third-party modules.
        subscribeMessages: function() {
```

```
            this.sandbox.subscribe("GenerateIndicator", this.onGenerateIndicator,
this, [this.sandbox.id]);
        }
    });

    return Terrasoft.CurrencyIndicatorModule;
});
```

3. Add a style to the LESS tab

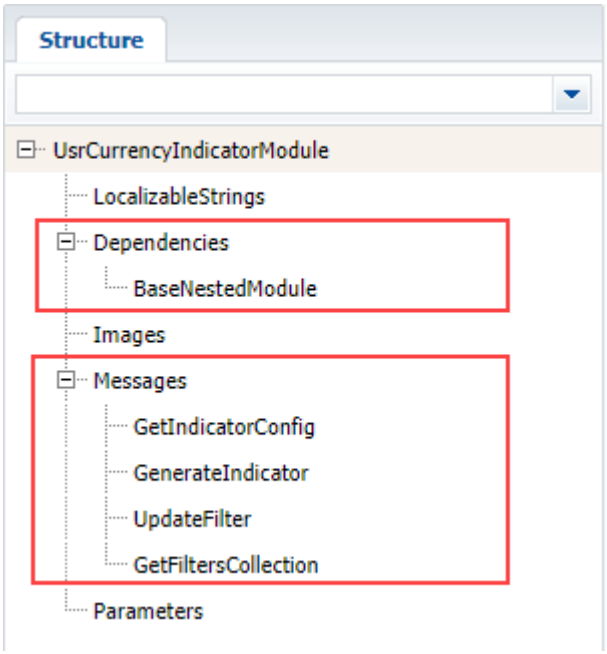
To display the widget text at the center, add the following style to the LESS tab of the module:

```
.indicator-module-wrapper {
    text-align: center;
}
```

3. Check the dependencies and messages

The dependencies and messages of parent module should automatically display in the created module (Fig. 2).

Fig. 2. Dependencies and messages of created module



If it doesn't happen, add them manually:

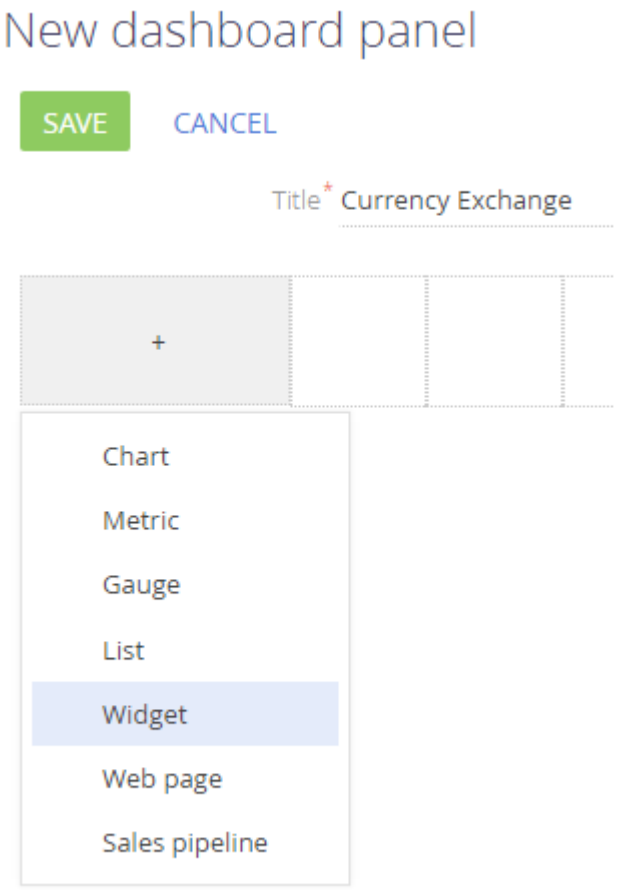
- Add a parent module to the [Dependencies] block
- Add the *GetIndicatorConfig* message to the [Messages] block. Sett the "Publish" direction for the message and the *GenerateIndicator* as address message with the "Follow" direction.

Save the new module.

5. Add the widget to the dashboard panel and set its parameters

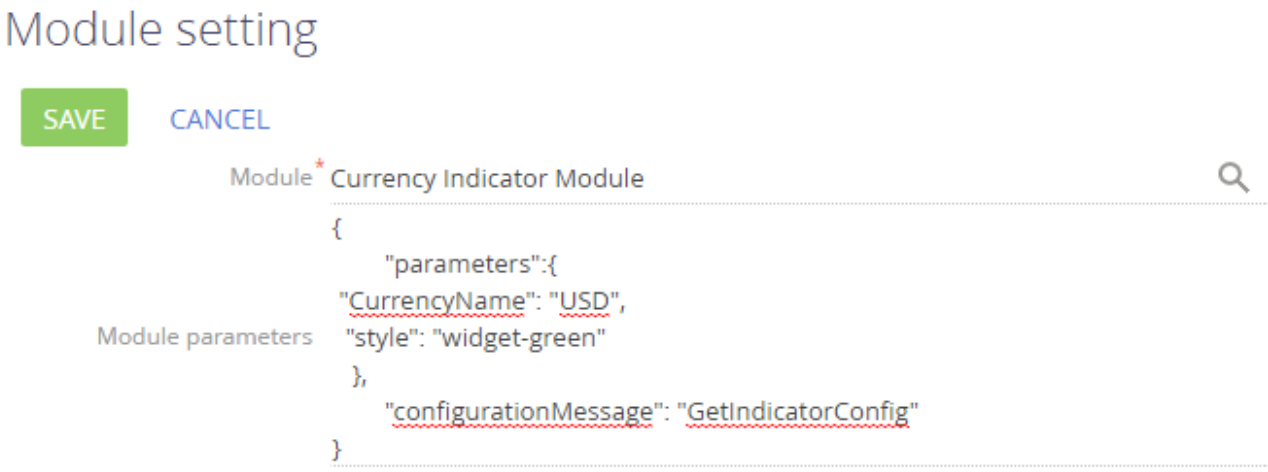
To display the widget, add it to the dashboard panel (Fig. 3).

Fig. 3. Adding the widget to the dashboard panel



In addition, you need to set the parameters of the module bound to the widget (Fig. 4).

Fig. 4. Configuration of the added widget module



To bind the module to the added widget, add the “Currency Indicator Module” value in the [Module] field and add the configuration JSON object with the required parameters to the [Module parameters] field.

```
{
    "parameters": {
        "CurrencyName": "USD",
        "style": "widget-blue"
    },
```

```
    "configurationMessage": "GetIndicatorConfig"
}
```

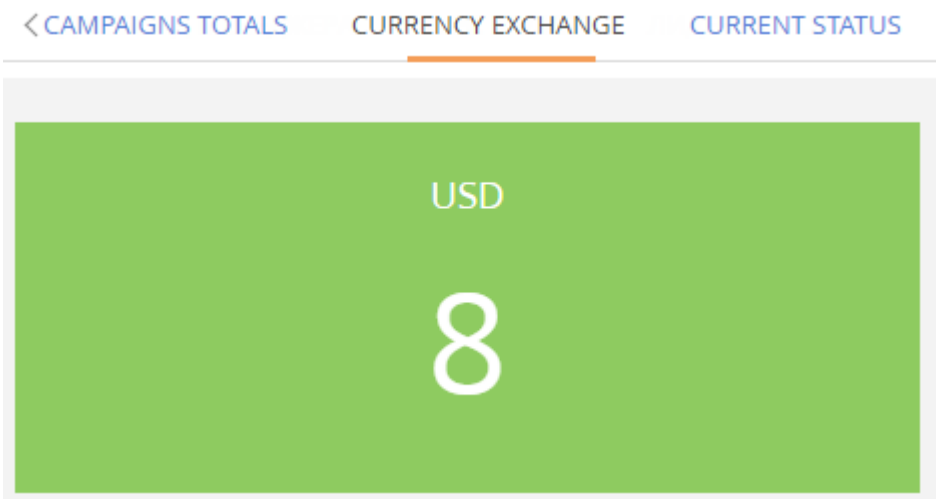
A “*CurrencyName*” parameter sets the currency for which the exchange rate is displayed. A “*style*” parameter sets the widget style and “*configurationMessage*” parameter sets the message name that will be used to transfer the configuration object.

You can set up any of bpm’online system colors in the *style* parameter as widget color (Fig. 5).

Fig. 5. Style types of the widget

After saving the created widget and refreshing the page, the custom widget will be displayed on the dashboards panel (Fig. 6).

Fig. 6. Currency exchange rate widget



The Terrasoft.AlignableContainer custom element

Difficulty level

Introduction

The *Terrasoft.AlignableContainer* custom element has been introduced in bpm'online version 7.8. This element is inherited from the *Terrasoft.Container* element, and contains the properties associated with the fixed container positioning, which depends on another element.

The *Terrasoft.AlignableContainer* view depends on the element used to position the container (the container is positioned at the center of the screen in case of element absence). Default container positioning order is defined by the following sequence:

1. The container displays under the element at first.
2. If there is no space under the element, the container displays above it.
3. If placing either below or above is impossible, the container displays on the right.
4. If placing on the right is impossible, the container displays to the left of the element.

You can also specify the container background for the *Terrasoft.AlignableContainer* custom element. The mentioned *Terrasoft.AlignableContainer* found their

Case description

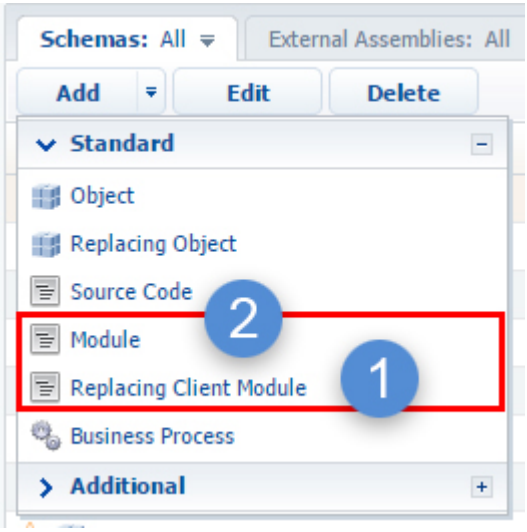
When you click on a photo in the [Contacts] section, display an enlarged image in the center of the screen with a background. When hovering over a photo, a larger version of the contact image relative to the photo container should be displayed.

Case implementation algorithm

1. Create a replacing client module

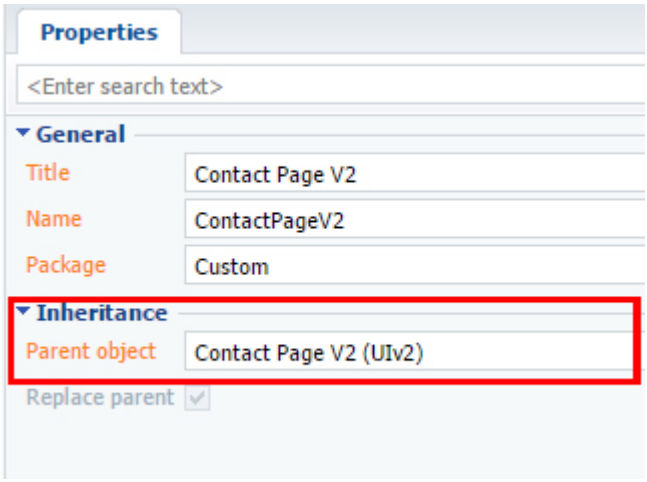
Go to the [Configuration] section, click [Add] and select [Replacing client module] (Fig. 1, 1) to create a replacing client module.

Fig. 1. Creating a replacing module



Select the *ContactPageV2* schema as the parent object of the *UIv2* package (Fig. 2).

Fig. 2. The *ContactPageV2* replacing schema properties



2. Display a larger version of a photo in the screen center.

To display images in the screen center, you must use the Terrasoft.AlignableContainer custom element. The element link (near which you want to display the container) is not specified as a parameter. Set the *true* value to the checkbox which is responsible for displaying the background for it to show up.

To display a photo in the center of the screen, you must create a handling method for the clicking on the contact's photo event. Add a closing button to hide the image.

To do this, add the following source code to the created edit page schema of a contact:

```
// Defining a module and it's dependencies.
define("ContactPageV2", ["css!UsrContactPhotoContainerCSS"], function() {
    return {
        // Object schema name.
        entitySchemaName: "Contact",
        attributes: {
            // The attribute responsible for displaying the container when clicking
on a photo.
            "LargeSizeContainerVisible": {
                // Element type.
                dataValueType: this.Terrasoft.DataValueType.BOOLEAN,
                // Column type.
                type: this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                // Element value.
                value: false
            }
        },
        methods: {
            // A method that displays a container with a larger photo.
            openLargeSizeImage: function() {
                // Sets the container to visible with a larger photo.
                this.set("LargeSizeContainerVisible", true);
                // Hides the container with an average-sized image.
                this.set("MiddleSizeContainerVisible", false);
            },
            // A method that hides containers with images.
            close: function() {
                // Hides a container that displays a large photo.
                this.set("LargeSizeContainerVisible", false);
                // ides a container that displays an average-sized image.
                this.set("MiddleSizeContainerVisible", false);
            }
        },
        diff: /**SCHEMA_DIFF*/[
            {
```

```
                // Connecting element properties.
                "operation": "merge",
                // Element name.
                "name": "Photo",
                // Parent element name.
                "parentName": "AccountPhotoContainer",
                // Property name.
                "propertyName": "items",
                // Element value.
                "values": {
                    // Adding a method handler for a container click event.
                    "onImageClick": {
                        // Binding to the method handler of a container click event.
                        "bindTo": "openLargeSizeImage"
                    }
                }
            },
            {
                // Element inserting.
                "operation": "insert",
                // Element name.
                "name": "AlignableLargePhotoContainer",
                // Element value.
                "values": {
                    // Container id.
                    "id": "AlignableLargePhotoContainer",
                    // Element type.
                    "itemType": Terrasoft.ViewItemType.CONTAINER,
                    // Element object classes.
                    "className": "Terrasoft.AlignableContainer",
                    // Element classes.
                    "wrapClass": ["photo-alignable-container", "large-size-image-
container"],
                    // Container visibility method handler.
                    "visible": {"bindTo": "LargeSizeContainerVisible"},
                    // The element near which you want to display the container.
                    "alignToEl": null,
                    // Background display chackbox.
                    "showOverlay": {"bindTo": "LargeSizeContainerVisible"},
                    // Container elements.
                    "items": []
                }
            },
            {
                // Inserting an element.
                "operation": "insert",
                // Element name.
                "name": "CloseLargePhotoButton",
                // Parent element name.
                "parentName": "AlignableLargePhotoContainer",
                // Property name.
                "propertyName": "items",
                // Element values.
                "values": {
                    // Element type.
                    "itemType": Terrasoft.ViewItemType.BUTTON,
                    // Element classes.
                    "classes": {
                        "imageClass": ["close-no-repeat-button"],
                        "wrapperClass": ["close-button-wrapper"]
                    },
                    // Forming an button with an image property.
```

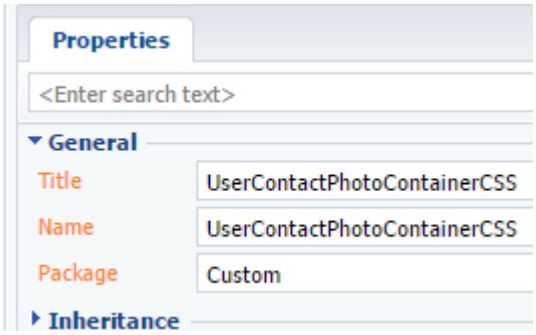
```
                    "imageConfig": {
                        "bindTo": "Resources.Images.CloseButtonImage"
                    },
                    // Method-handler for pressing the button to close the element.
                    "click": {"bindTo": "close"}
                }
            },
            {
                // Inserting an element.
                "operation": "insert",
                // Element name.
                "name": "AccountLargeResizedPhotoContainer",
                // Parent element name.
                "parentName": "AlignableLargePhotoContainer",
                // Property name.
                "propertyName": "items",
                // Element values.
                "values": {
                    // Element type.
                    "itemType": Terrasoft.ViewItemType.BUTTON,
                    // Element object class.
                    "className": "Terrasoft.ImageView",
                    // Method of obtaining a link to an image.
                    "imageSrc": {"bindTo": "getContactImage"}
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

In order to set the required dimensions of the displayed photo container, you need to define its CSS-style.

To do this, go to the [Configuration] section, and select [Add] > [Standard] > [Module] in the [Schemas] tab (Fig. 1, 2).

In module properties, set the title and header to "UsrContactPhotoContainerCSS" (Fig. 3).

Fig. 3. Module properties



Module styles are defined on the LESS tab (Fig. 4).

Fig. 4. LESS module tab

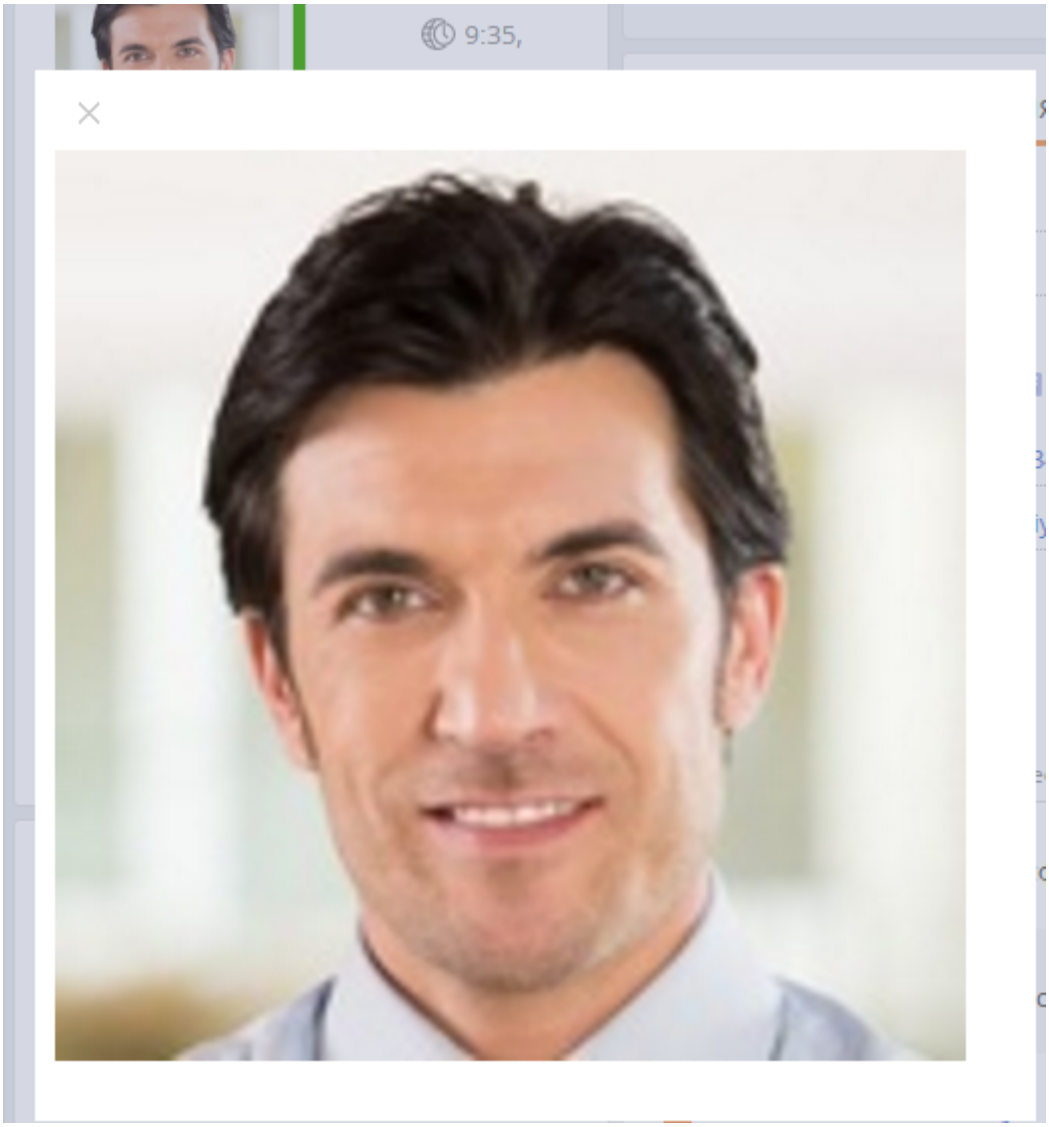
Add the following CSS selectors for the contact photo to display correctly in the screen center:

```
.schema-wrap .photo-alignable-container::before,
.schema-wrap .alignable-container-overlay::before {
    background: transparent;
```

```
}

.schema-wrap .photo-alignable-container.alignable-container {
    background: white;
}


.photo-alignable-container.large-size-image-container {
    width: 500px;
    height: 525px;
}

.photo-alignable-container .close-no-repeat-button {
    background-repeat: no-repeat;
}

.photo-alignable-container .close-button-wrapper:hover {
    background: transparent;
}

#ContactPageV2AccountLargeResizedPhotoContainerButton-image-view,
#ContactPageV2AccountResizedPhotoContainerButton-image-view
{
    height: 90%;
}
```

To check the generated code functionality, save the created modules, restart the application, and go to the contact page and click on the photo. A larger contact photo in the page center on a semi-transparent gray background will show up as the result (Fig. 5).

Fig. 5. A larger contact phone in the screen center



3. Display a larger photo after hovering the mouse cursor over the contact photo container.

To display a larger image after hovering over the user photo container, you must add an event handler for the cursor hovering event. Also, the Terrasoft.AlignableContainer container needs to transfer the name of the element near which the container should be displayed. Finally, add the logic of hiding the Terrasoft.AlignableContainer container. To do this, make the following changes to the source code of the replaced contact page schema.

Add the followinf attributes to the *attributes* section.

```
attributes: {
    // The id of the element near which you want to display the container.
    "AlignToElementId": {
        // Element type.
        dataValueType: this.Terrasoft.DataValueType.TEXT,
        // Column type.
        type: this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
        // Element value.
        value: "ContactPageV2AccountPhotoContainerContainer"
    },
    // The attribute responsible for displaying the container when hovering over a
photo.
    "MiddleSizeContainerVisible": {
        // Element type.
```

```
        dataValueType: this.Terrasoft.DataValueType.BOOLEAN,
        // Column type.
        type: this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
        // Element value.
        value: false
    }...
},
```

Add the following methods to the *methods* section.

```
methods: {
    // A method that performs actions after loading an object entity.
    onEntityInitialized: function() {
        // Calls the parent element.
        this.callParent(arguments);
        // Sets the element near which you want to display the container.
        this.setAlignToEl();
        // Creates a subscription to the mouse cursor hover event.
        this.subscribePhotoContainerEvents();
    },
    // The method that sets the element near which you want to display the container.
    setAlignToEl: function() {
        // Gets the item ID.
        var alignToElementId = this.get("AlignToElementId");
        // Gets the DOM element.
        var alignToEl = this.Ext.get(alignToElementId);
        // Writes the value of the DOM element to the AlignToEl parameter.
        this.set("AlignToEl", alignToEl);
    },
    // A method that creates a subscription to the mouse hover event.
    subscribePhotoContainerEvents: function() {
        // Gets the DOM element of the photo container.
        var container = this.get("AlignToEl");
        // Creates a subscription to the mouse cursor hover event.
        container.on("mouseover", this.openMiddleSizeImage, this);
    },
    // The method that displays the image after hovering over the container.
    openMiddleSizeImage: function() {
        // Makes the average-sized photo container visible.
        this.set("MiddleSizeContainerVisible", true);
        // Hides the larger photo container.
        this.set("LargeSizeContainerVisible", false);
    },
    ...
    // A method that hides containers with images.
    close: function() {
        // Hides the larger photo container.
        this.set("LargeSizeContainerVisible", false);
        // Hides the average-sized photo container.
        this.set("MiddleSizeContainerVisible", false);
    }
},
```

Add the following configuration objects to the *diff* array:

```
diff: /**SCHEMA_DIFF*/[
    {
        // Inserting an element.
        "operation": "insert",
        // Element name.
        "name": "AlignablePhotoContainer",
        // Element values.
```

```
        "values": {
            // Container id.
            "id": "AlignablePhotoContainer",
            // Element type.
            "itemType": Terrasoft.ViewItemType.CONTAINER,
            // Element object class.
            "className": "Terrasoft.AlignableContainer",
            // Element classes.
            "wrapClass": ["photo-alignable-container", "middle-size-image-
container"],
            // Method handler of container visibility.
            "visible": {"bindTo": "MiddleSizeContainerVisible"},
            // The element near which you want to display the container.
            "alignToEl": {"bindTo": "AlignToEl"},
            // A checkbox for displaying the background.
            "showOverlay": false,
            // Container elements.
            "items": []
        }
    },
    {
        // Inserting an element.
        "operation": "insert",
        // Element name.
        "name": "ClosePhotoButton",
        // Parent element name.
        "parentName": "AlignablePhotoContainer",
        // Property name.
        "propertyName": "items",
        // Element values.
        "values": {
            // Element type.
            "itemType": Terrasoft.ViewItemType.BUTTON,
            // Element classes.
            "classes": {
                "imageClass": ["close-no-repeat-button"],
                "wrapperClass": ["close-button-wrapper"]
            },
            // Forming an button with an image property.
            "imageConfig": {
                "bindTo": "Resources.Images.CloseButtonImage"
            },
            // Method-handler for pressing the button to close the element.
            "click": {"bindTo": "close"}
        }
    },
    {
        // Inserting an element.
        "operation": "insert",
        // Element name.
        "name": "AccountResizedPhotoContainer",
        // Parent element name.
        "parentName": "AlignablePhotoContainer",
        // property name.
        "propertyName": "items",
        // Element values.
        "values": {
            // Element type.
            "itemType": Terrasoft.ViewItemType.BUTTON,
            // Element object class.
            "className": "Terrasoft.ImageView",
            // Method of getting an image url.
```

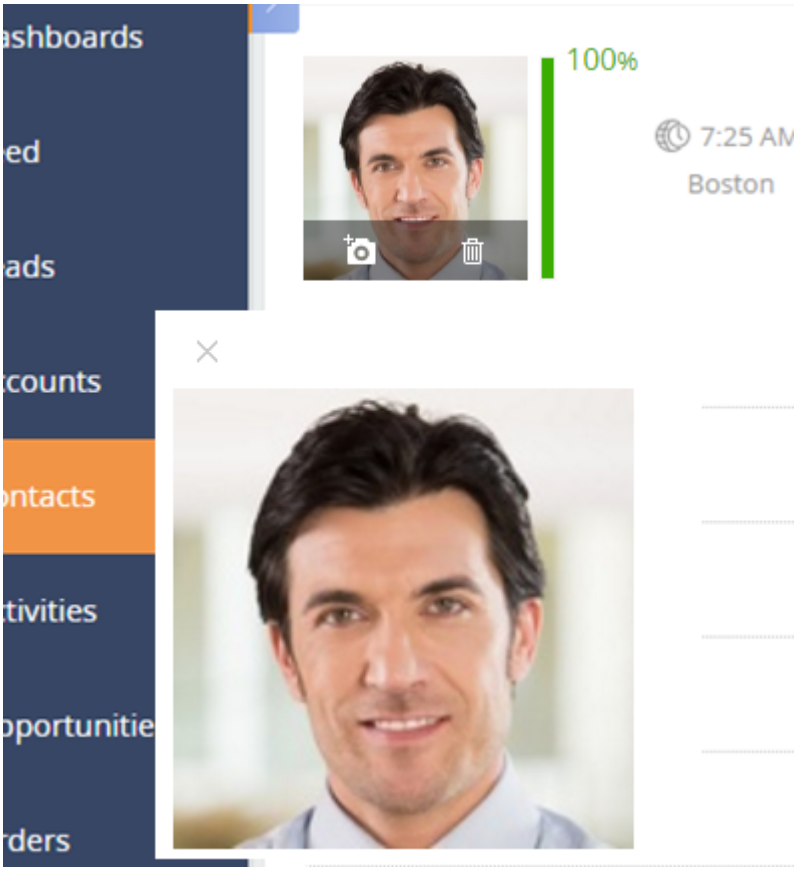
```
                "imageSrc": {"bindTo": "getContactImage"}
            }
        },...
]/**SCHEMA_DIFF*/
```

Also, to set the required size of the "pop-up" photo, add the following CSS selector on the LESS tab of the *UsrContactPhotoContainerCSS* module:

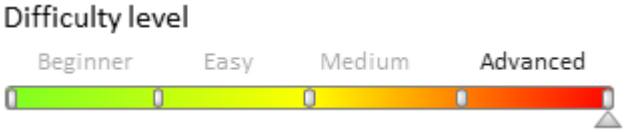
```
.photo-alignable-container.middle-size-image-container {
    width: 250px;
    height: 275px;
}
```

To check the generated code functionality, save the created modules, restart the application, and go to the contact page and hover the cursor over the photo. A larger contact photo will show up next to the main one as the result (Fig. 6).

Fig. 6. A larger contact photo next to the main one



Adding a duplicate search rule



Introduction

Deduplication process uses the rules created as procedures and stored in bpm’online. When a rule is being executed, it populates the *ContactDuplicateSearchResult* table with a list of duplicates. Search results are grouped by rules and displayed on the duplicate page. Learn more about this functionality in the “Finding and merging duplicates” documentation.

To add a duplicate rule:

1. Add a column to the object schema (if needed). The column value will be used for the search of duplicates.

2. Add the stored search procedure to the application database.

3. Register the stored procedure as a new rule.

Case description

When launching the duplicate search process, the contacts with the same [Taxpayer ID] column values should be considered duplicates and should be displayed as search result.

Case implementation algorithm

1. Adding a field whose value will be used for the duplicate search

Since the [Taxpayer ID] field is not available on a standard contact edit page, add the field to the page (e.g., via a section wizard). For more information about adding fields to edit pages see the "Adding a new field to the edit page" article.

The new field properties:

- [Title] – "Taxpayer ID"
- [Name in DB] – "UsrInn"

2. Adding the stored search procedure to the application database

Add the stored duplicate search procedure by the [Taxpayer ID] field to the database. To do so, execute the following SQL script:

```
-- Verifying if the stored tsp_FindContactDuplicateByInn procedure is available.
IF NOT OBJECT_ID('[dbo].[tsp_FindContactDuplicateByInn]') IS NULL
BEGIN
    -- Deleting the stored procedure.
    DROP PROCEDURE [dbo].[tsp_FindContactDuplicateByInn];
END;
GO
-- Creating the stored procedure.
CREATE PROCEDURE [dbo].[tsp_FindContactDuplicateByInn] (
    -- This table parameter is only rendered if a new contact is stored.
    -- Contains the new contact data.
    -- If duplicate global search process is launched, the rendered parameter
contains no data.
    @parsedConfig CreatingObjectInfo READONLY,
    -- Unique identifier of user who launched the duplicate search.
    @sysAdminUnit UNIQUEIDENTIFIER,
    -- Identifier of the current rule from the [ContactDuplicateSearchResult] table.
    -- This identifier is created after a rule is registered in the system.
    @ruleId UNIQUEIDENTIFIER
)
AS
BEGIN
    -- Receiving the quantity of records from the accepted table for defining the
duplicate global search launch.
    DECLARE @parsedConfigRowsCount INT = (SELECT COUNT(*) FROM @parsedConfig);
    -- Creating temporary table with contact data for the search.
    CREATE TABLE #searchContact (
        [UsrInn] INT,
        [SortDate] DATETIME
    );
    -- In case of global search, the temporary table is populated with data.
    IF @parsedConfigRowsCount = 0
```

```
    BEGIN
        -- Adding data for duplicate search to the temporary table.
        INSERT INTO #searchContact ([UsrInn], [SortDate])
        -- Query for contact data selection.
        SELECT
            -- The Taxpayer ID columns of contact modification date are selected.
            [UsrInn],
            MAX([ModifiedOn])
        FROM [Contact]
        -- Grouping by fields is added to enable using qauntity verification.
        GROUP BY [UsrInn]
        -- The table is populated only if more than one contact is available.
        HAVING COUNT(*) > 1;
    END;

    -- Populating the table of results.
    INSERT INTO [ContactDuplicateSearchResult] ([ContactId], [GroupId], [RuleId],
[SysAdminUnitId])
    SELECT
        -- Contact duplicate identifier.
        [vr].[Id],
        -- Group numbering.
        DENSE_RANK() OVER (ORDER BY [vr].[SortDate] DESC, [vr].[UsrInn]),
        -- Rule identifier.
        @ruleId RuleId,
        -- Identifier of user who launched the duplicate search process.
        @sysAdminUnit
    FROM (
        -- Subquery populating the duplicatae table.
        SELECT
            -- Contact identifier.
            [v].[Id],
            --Contact Taxpayer ID.
            [v].[UsrInn],
            -- Date of sorting.
            [r].[SortDate]
        -- Tables providing data.
        FROM [Contact] [v], #searchContact r
        -- The rule defining that contacts are duplicates.
        WHERE [v].[UsrInn] = [r].[UsrInn]
        -- Groupoing of search results.
        GROUP BY [v].[UsrInn], [r].[SortDate], [v].[Id]
    ) [vr];
END;
GO
```

> **NOTE**
>
> Sometimes you can come across the "Cannot resolve the collation conflict between "Cyrillic_General_CI_AS" and "Cyrillic_General_CI_AI" in the equal to operation" error. To fix it, specify the necessary COLLATE when creating the table column.
>
> ```
> CREATE TABLE #searchContact ([Name] NVARCHAR(128) COLLATE Cyrillic_General_CI_AI,
> [BirthDate] DATETIME, [SortDate] DATETIME );
> ```

3. Registering the stored procedure as a new rule

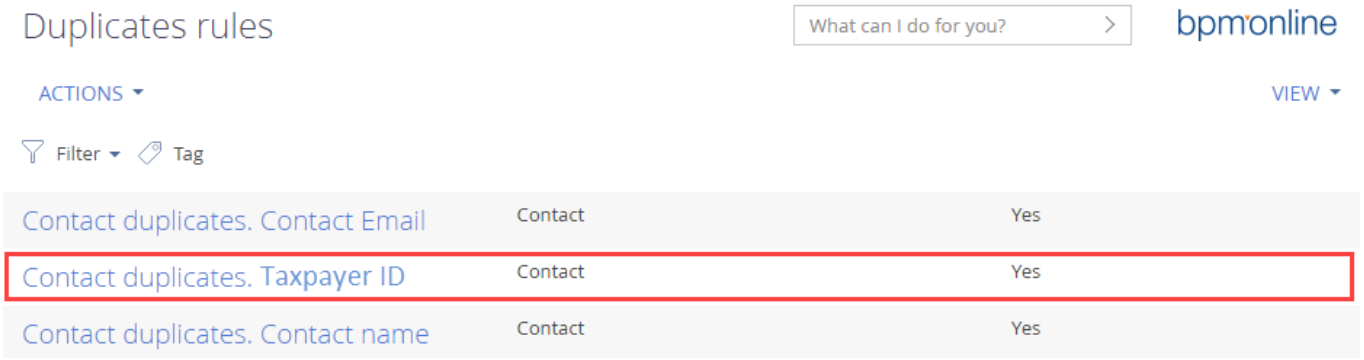
To register the stored procedure as a new duplicate search rule, add the corresponding record to the *DuplicatesRule* table. To do so, execute the following SQL script:

```
-- Veriable that stores the UId column value of the Contact schema.
DECLARE @ContactUId UNIQUEIDENTIFIER;
```

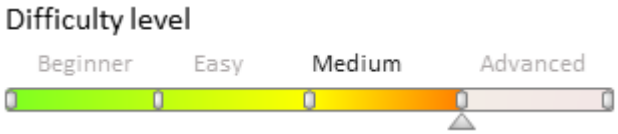
```
-- Receives the UId ccolumn value of the Contact schema.
Set @ContactUId = (SELECT TOP 1 SysSchema.UId FROM SysSchema
WHERE SysSchema.Name = 'Contact' AND SysSchema.ExtendParent = 0);

-- Adds a new rule to the system.
INSERT INTO DuplicatesRule ([IsActive], [ObjectId], [ProcedureName], [Name]) VALUES
  (1, @ContactUId, 'tsp_FindContactDuplicateByInn', 'Дубли контактов. ИНН');
```

After you update the application page and clear the cache, you will have a new rule in the list of duplicate search rules (Fig. 1).

Fig. 1. Duplicate search rule by the [Taxpayer ID] field



Junk case custom filtering



Introduction

Bpm'online users can filter unwanted cases by creating a list of email addresses and domains for automatic spam detection. Incoming cases from these domains or addresses are either not registered at all or have the "Canceled" status (the default status value is set in the [Junk Case Default Status] system setting).

Junk filter enables email header analysis based on certain flags, e.g. *Auto-Submitted*. Emails with those flags are either not registered or register with the pre-defined initial status, set in system settings.

To analyze email addresses and domains, simply add the required value to the [Blacklist of email addresses and domains for case registration] lookup. The emails will be marked as blacklisted during the analysis when you populate the lookup with values (their type is set automatically).

The algorithm of adding a new email filtering property

Email analysis is done through the [Email header properties management] lookup. Follow these steps to add a new analysis property:

1. Add a new class that implements the *IHeaderPropertyHandler* interface. This interface contains a single *Check()* method that returns a value of *bool* type. Check the property value and return the result during the *Check()* method implementation. If the method returns *true*, the system creates the case using the standard mechanism, if *false*, the system treats the email as blacklisted.
2. Add an [Email header properties management] lookup value. Specify the property name used for analysis in the *Name* column. Specify the class name (added in the previous paragraph) in the *handler* column.

Adding a new email filtering property

Case description

Add a new *No-reply* property for case analysis. If the property is found in the email header and its value is anything other than "No", treat the case as blacklisted.

Case implementation algorithm

1. Add a new class that implements the IHeaderPropertyHandler interface.

Add a "Source code" schema in a custom package (e.g. *Custom*). In this schema, define a class that implements the *IHeaderPropertyHandler* interface, e.g.:

```
namespace Terrasoft.Configuration
{
    using System;
    public class NoreplyHandler: IHeaderPropertyHandler
    {
        public bool Check(object value) {
            return string.Equals(value.ToString(), "No",
StringComparison.OrdinalIgnoreCase);
        }
    }
}
```

Save and publish the new schema.

The *IHeaderPropertyHandler* interface is defined in the *JunkFilter* package, so it must be added to the custom package dependency.

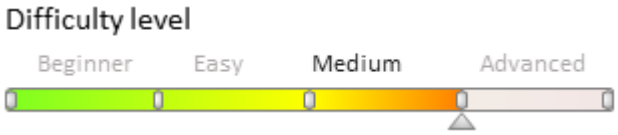
2. Adding a lookup value

Add the *No-reply* property to the [Email header properties management] lookup. Select the *NoreplyHandler* class (created in the previous paragraph) as the handler class (*handler* property).

After receiving an email with a *No-reply* header flag and a value that is anything other than "No", the following options are possible:

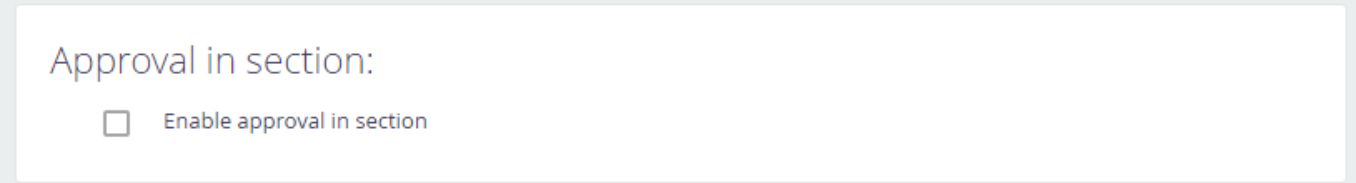
- the case is not created if the [Create Cases From Junk Emails] system setting value is *false*;
- the case is created with the [Junk Case Default Status] system setting status if the [Create Cases From Junk Emails] system setting value is *true*.

How to display custom implementation of approving in the section wizard



Starting with version 7.11 bpm'online can now implement approving functions and informing about approvals in any section. Approvals are enabled in the section wizard. The section wizard has the [Enable approval in section] checkbox to enable approvals (Fig. 1). Previous versions of bpm'online needed a project complex solutions to enable approving functions.

Fig. 1. [Enable approval in section] checkbox in the section wizard



If the custom approving had been already implemented in bpm'online, follow instructions below to enable the [Enable approval in section] checkbox in the section wizard.

1. Add a record in the SysModuleVisa table

To do so, execute the following SQL script:

```
insert SysModuleVisa(
    VisaSchemaUId,
    MasterColumnUId,
    UseCustomNotificationProvider)
select
    'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX',
    'YYYYYYYY-YYYY-YYYY-YYYY-YYYYYYYYYYYY',
    0
```

where

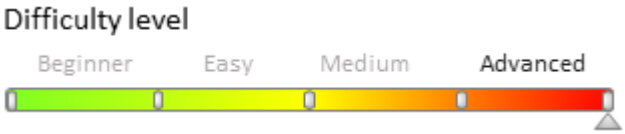
- *VisaSchemaUId* – *UId* of user object inherited from the [Base approval] object.
- *MasterColumnUId* – *UId* of the field of interaction with the section.
- *UseCustomNotificationProvider* – a flag of using custom provider. "0" if you need to use custom provider by default, "1" if user has created own message provider.

2. Update a record for the section in the SysModule table

In the *SysModule* table for corresponding section sill the *SysModuleVisaId* field with value of the *Id* of added record in the *SysModuleVisa* table. To do so, execute the following SQL script:

```
update SysModule
set SysModuleVisaId ='ZZZZZZZZ-ZZZZ-ZZZZ-ZZZZ-ZZZZZZZZZZZZ'
where Code='KnowledgeBase'
```

How to create custom reminders and notifications



Introduction

Starting with version 7.12.0, reminder and notification sending mechanics has been reworked in bpm'online.

Previously, to send a custom notification, you would have to:

- Creates a class that implements *INotificationProvider* interface or an inherited abstract *BaseNotificationProvider* class.
- Add logic for selecting custom notifications by bpm'online.
- Register a class in the *NotificationProvider* table.

The notifications were sent once a minute, calling all classes from the *NotificationProvider* table.

Starting with version 7.12.0, it is sufficient to create a notification or reminder with the needed parameters. After this, the application will either send the notification immediately, or display a reminder at the specified time.

To set up custom notifications:

1. **Create a [Source code] schema** in the custom package and define a class for generating the notification text and pop-up window. The class must implement the *IRemindingTextFormer* interface (declared in the *IRemindingTextFormer* schema of the *Base* package).

2. Replace the needed object (such as [Lead]) or specify notification sending logic in it.

3. Replace the reminder tab schema *ReminderNotificationsSchema* for displaying notifications for the needed object.

Case description

Create a custom reminder about opportunity actualization date in leads. The date must be specified in the [Next actualization date] field on the [Opportunity info] tab.

Source code

A package with implemented example is available for download via the following link.

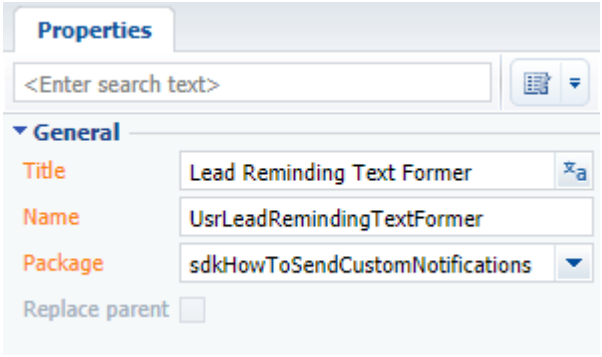
Case implementation algorithm

1. Create a class for generating the reminder text and the pop-up window.

1. Add a [Source code] schema in the custom package (see "**Creating the [Source code] schema**"). Set the following properties (Fig. 1):

- [Title] – "Lead Reminding Text Former".
- [Name] – "UsrLeadRemindingTextFormer"

Fig. 1. The [Source code] schema properties



2. Use the context menu of the [Structure] tab to add two localized strings (Fig. 2). Properties of the localized strings are described in table 1.

Fig. 2. Adding localized strings to schema

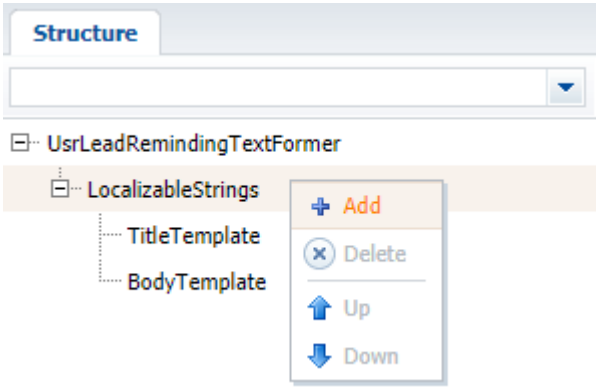


Table 1. Localized string properties

Name	Value
TitleTemplate	You need to update the sale
BodyTemplate	Lead {0} requires update of sales information

3. Add class implementation for forming reminder text and pop-up window:

```
namespace Terrasoft.Configuration
{
    using System.Collections.Generic;
    using Terrasoft.Common;
    using Terrasoft.Core;

    public class UsrLeadRemindingTextFormer : IRemindingTextFormer
    {
        private const string ClassName = nameof(UsrLeadRemindingTextFormer);
        protected readonly UserConnection UserConnection;

        public UsrLeadRemindingTextFormer(UserConnection userConnection) {
            UserConnection = userConnection;
        }
        // Generates reminder text from a collection of inbound parameters and
BodyTemplate localized string.
        public string GetBody(IDictionary<string, object> formParameters) {
            formParameters.CheckArgumentNull("formParameters");
            var bodyTemplate = UserConnection.GetLocalizableString(ClassName,
"BodyTemplate");
            var leadName = (string)formParameters["LeadName"];
            var body = string.Format(bodyTemplate, leadName);
            return body;
        }
        // Generates reminder title from the class name and TitleTemplate localize
string.
        public string GetTitle(IDictionary<string, object> formParameters) {
            return UserConnection.GetLocalizableString(ClassName, "TitleTemplate");
        }
    }
}
```

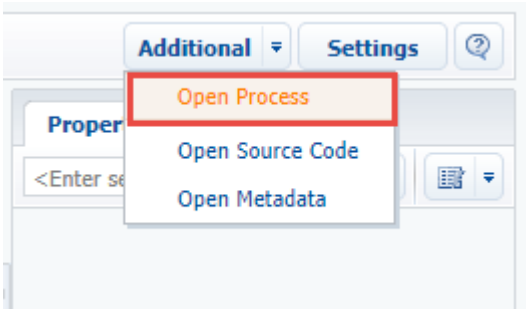
4. Save and publish the schema.

2. Replace the [Lead] object and set the reminder logic in it.

1. Create a replacing schema of the [Lead] object (see "Crating a replacing object schema" section of the "**Creating the entity schema**" article).

2. Click [Additional] and select [Open process]. Built-in process of the replacing [Lead] object will open (Fig. 3).

Fig. 3. Opening built-in process



3. Using the context menu in the [Structure] tab, add process parameter *GenerateReminding* (Fig. 4) with the following properties (Fig. 5):

- [Title] – "Generate Reminding".
- [Name] – "GenerateReminding".
- [Data type] – "Boolean".

Fig. 4. Adding a process parameter

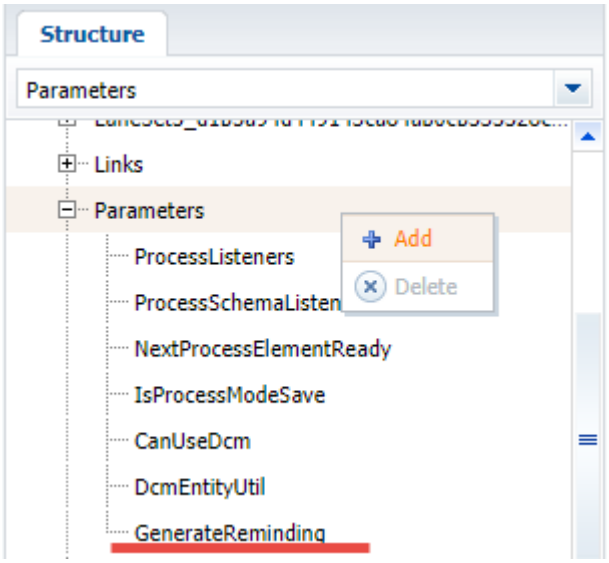
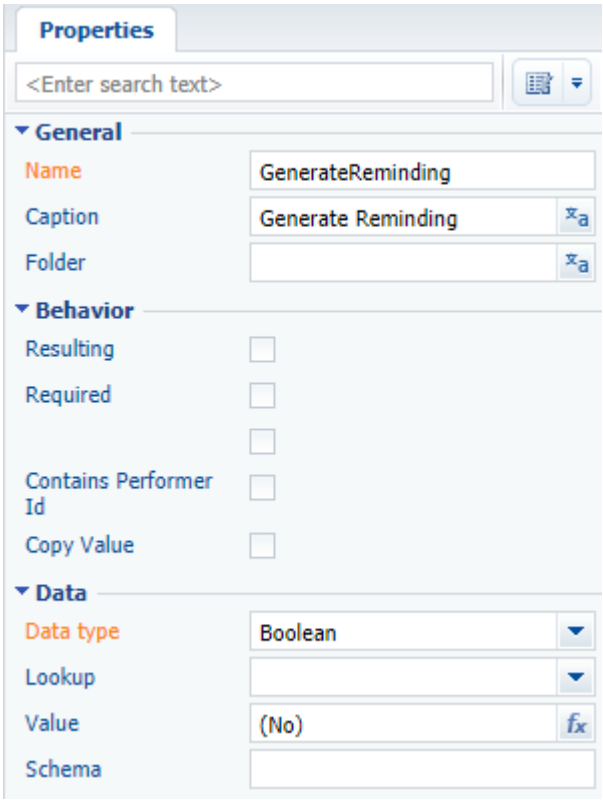


Fig. 5. The properties for the process parameter



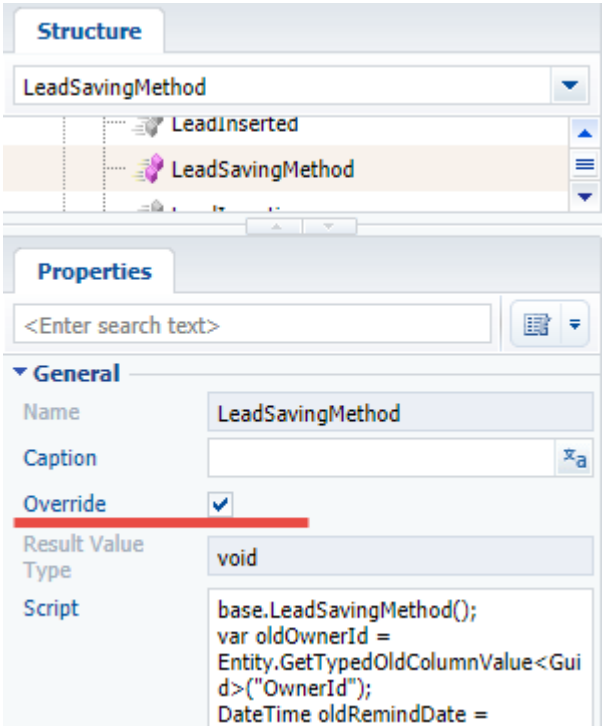
4. Select the *LeadSavingMethod()* (called before saving the object) in the [Methods] node on the [Structure] tab (Fig. 6). Select the [Override] checkbox and add the following code to the method body:

```
// Calling base implementation of the method.
base.LeadSavingMethod();
// Getting owner Id.
var oldOwnerId = Entity.GetTypedOldColumnValue<Guid>("OwnerId");
// Getting next actualization date.
DateTime oldRemindDate = Entity.GetTypedOldColumnValue<DateTime>
("NextActualizationDate");
// Comparing Id of original and current owner.
bool ownerChanged = !Entity.OwnerId.Equals(oldOwnerId);
// Comparing original and current actualization dates.
bool remindDateChanged = !Entity.NextActualizationDate.Equals(oldRemindDate);
```

```
// Set the GenerateReminding process parameter value to "true" if the owner
// or actualization date changed.
GenerateReminding = ownerChanged || remindDateChanged;
```

> **NOTE**
>
> To open source code editor of a method, double-click the method name on the [Structure] tab.

Fig. 6. Properties of the LeadSavingMethod

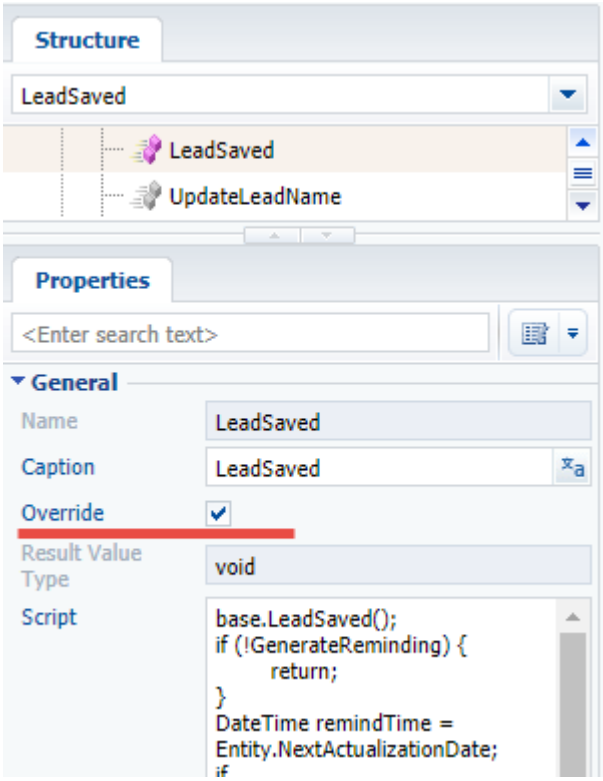


5. Select the *LeadSavedMethod()* (called after saving the object) in the [Methods] node on the [Structure] tab (Fig. 7). Select the [Override] checkbox and add the following code to the method body:

```
base.LeadSaved();
// Checks if the reminder must be generated.
if (!GenerateReminding) {
    return;
}
DateTime remindTime = Entity.NextActualizationDate;
if (Entity.OwnerId.Equals(Guid.Empty) || remindTime.Equals(default(DateTime))) {
    return;
}
// Instantiates the UsrLeadRemindingTextFormer class.
IRemindingTextFormer textFormer =
    ClassFactory.Get<UsrLeadRemindingTextFormer>(new
ConstructorArgument("userConnection", UserConnection));
// Gets lead name.
string leadName = Entity.LeadName;
// Gets the reminder text.
string subjectCaption = textFormer.GetBody(new Dictionary<string, object> {
            {"LeadName", leadName}
        });
// Gets reminder title.
string popupTitle = textFormer.GetTitle(null);
// Configures reminder.
var remindingConfig = new RemindingConfig(Entity);
// Message author — current contact.
```

```
remindingConfig.AuthorId = UserConnection.CurrentUser.ContactId;
// Target recipient – lead owner.
remindingConfig.ContactId = Entity.OwnerId;
// Type – reminder.
remindingConfig.NotificationTypeId = RemindingConsts.NotificationTypeRemindingId;
// Reminder date – next actualization date of an opportunity in lead.
remindingConfig.RemindTime = remindTime;
// Reminder text.
remindingConfig.Description = subjectCaption;
// Reminder title.
remindingConfig.PopupTitle = popupTitle;
// Creating reminder utility class.
var remindingUtilities = ClassFactory.Get<RemindingUtilities>();
// Creating reminder.
remindingUtilities.CreateReminding(UserConnection, remindingConfig);
```

Fig. 7. Properties of the LeadSavedMethod



> **NOTE**
>
> To display reminders on the system notification tab , replace *remindingConfig.NotificationTypeId = RemindingConsts.NotificationTypeRemindingId;* with *remindingConfig.NotificationTypeId = RemindingConsts.NotificationTypeNotificationId;* in the code of the LeadSavedMethod.

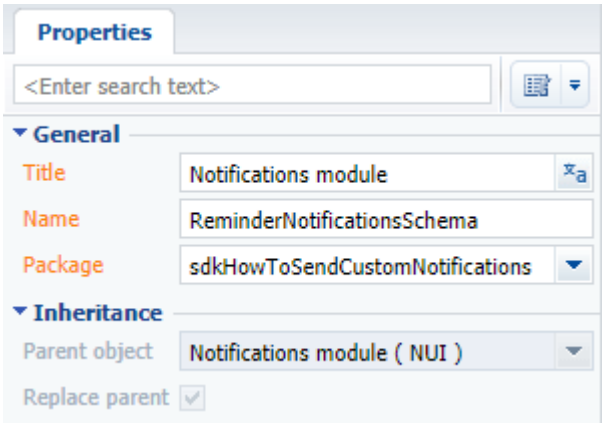
6. Save and publish built-in process schema of the [Lead] object. Publish the [Lead] object schema.

3. Replace the reminder tab schema ReminderNotificationsSchema.

1. To display reminders for a specific object, create a replacing *ReminderNotificationsSchema* in the **custom package** and add the needed logic to it. The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**" article. Replacing schema properties (Fig. 8):

- [Title] – "Notifications module".
- [Name] – "ReminderNotificationsSchema".

Fig. 8. Properties of the ReminderNotificationsSchema schema



2. Add following source code on the [Source code] tab of the schema:

```
define("ReminderNotificationsSchema", ["ReminderNotificationsSchemaResources"],
    function() {
        return {
            entitySchemaName: "Reminding",
            methods: {
                // Determines of the reminder is related to the lead.
                getIsLeadNotification: function() {
                    return this.get("SchemaName") === "Lead";
                },
                // Gets reminder title.
                getNotificationSubjectCaption: function() {
                    var caption = this.get("Description");
                    return caption;
                }
            },
            // Array of view model notifications.
            diff: [
                // Reminder primary container.
                {
                    "operation": "insert",
                    "name": "NotificationleadItemContainer",
                    "parentName": "Notification",
                    "propertyName": "items",
                    "values": {
                        "itemType": Terrasoft.ViewItemType.CONTAINER,
                        "wrapClass": [
                            "reminder-notification-item-container"
                        ],
                        // Displays only for leads.
                        "visible": {"bindTo": "getIsLeadNotification"},
                        "items": []
                    }
                },
                // Title container.
                {
                    "operation": "insert",
                    "name": "NotificationItemleadTopContainer",
                    "parentName": "NotificationleadItemContainer",
                    "propertyName": "items",
                    "values": {
                        "itemType": Terrasoft.ViewItemType.CONTAINER,
                        "wrapClass": ["reminder-notification-item-top-container"],
                        "items": []
                    }
                },
```

```
                // Image.
                {
                    "operation": "insert",
                    "name": "NotificationleadImage",
                    "parentName": "NotificationItemleadTopContainer",
                    "propertyName": "items",
                    "values": {
                        "itemType": Terrasoft.ViewItemType.BUTTON,
                        "className": "Terrasoft.ImageView",
                        "imageSrc": {"bindTo": "getNotificationImage"},
                        "classes": {"wrapClass": ["reminder-notification-icon-
class"]}
                    }
                },
                // Date display.
                {
                    "operation": "insert",
                    "name": "NotificationDate",
                    "parentName": "NotificationItemleadTopContainer",
                    "propertyName": "items",
                    "values": {
                        "itemType": Terrasoft.ViewItemType.LABEL,
                        "caption": {"bindTo": "getNotificationDate"},
                        "classes": {"labelClass": ["subject-text-labelClass"]}
                    }
                },
                // Reminder text display.
                {
                    "operation": "insert",
                    "name": "NotificationleadSubject",
                    "parentName": "NotificationItemleadTopContainer",
                    "propertyName": "items",
                    "values": {
                        "itemType": Terrasoft.ViewItemType.LABEL,
                        "caption": {"bindTo": "getNotificationSubjectCaption"},
                        "click": {"bindTo": "onNotificationSubjectClick"},
                        "classes": {"labelClass": ["subject-text-labelClass", "label-
link", "label-url"]}
                    }
                }
            ]
        };
    });
```

3. Save the schema.

As a result, bpm’online will create reminders (Fig. 10) for all leads where the owner and actualization date are specified (Fig. 9).

Fig. 9. Lead page where owner and next actualization date fields populated

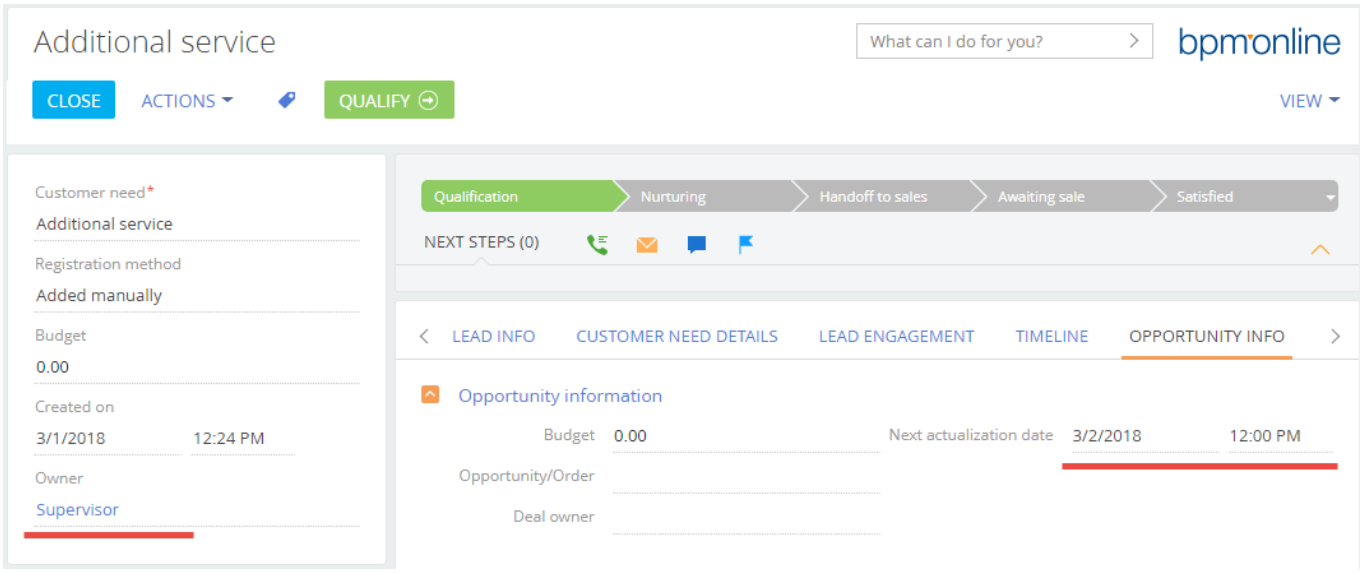
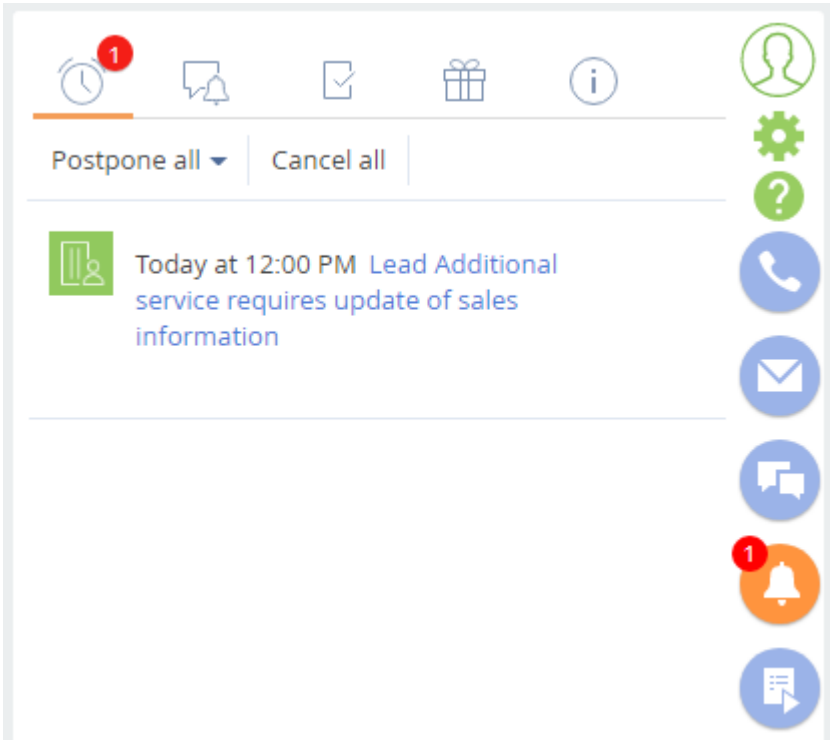
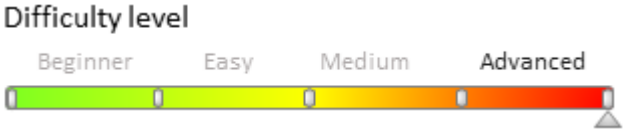


Fig. 10. Lead custom reminder



How to create the [Timeline] tab tiles bound to custom section



Introduction

Starting from version 7.12.0 you can use the [Timeline] tab for quick analysis of customer cooperation, opportunity, case, etc. history in bpm’online. This tab is available by default in the [Contacts], [Accounts], [Leads], [Opportunities] and [Cases] sections. General information about this tab functions is provided in the "**The [Timeline] tab**” article.

You can set up a tile using the *TimelinePageSetting* table settings as shown in the example with the base tile (see "**The [Timeline] tab**"). In such a case you will use the following for your tile:

- the default icon
- the *BaseTimelineItemView* and *BaseTimelineItemViewModel* base view and view model modules
- author field
- tile caption field
- message field

You can use one and the same tile for different sections if needed. However, we recommend to use the *TimelineTileSetting* table and set up your tiles for different sections.

The *TimelineTileSetting* table contains tile configurations that already exist in bpm’online. The section, however, will only display the tiles indicated in the *TimelinePageSetting* table for this particular section.

For example, the *TimelineTileSetting* contains three pre-configured tiles: *Tasks*, *Leads* and *Calls*. The *TimelinePageSetting* table contains the *Tasks* and *Calls* tiles that are pre-configured for usage in the [Accounts] section, and only the *Calls* tile that is pre-configured for usage in the [Contacts] section. The *Leads* tile in this case will not be displayed in any section.

NOTE

It is considered a good practice to differentiate the tile settings. We recommend to use the *TimelineTileSetting* table for setting up tiles, and the *TimelinePageSetting* – for adding the tiles to section timeline.

ATTENTION

If you need to add the existing *Files* tile to a section, the "entitySchemaName” property of the *TimelinePageSetting* table should contain the entity schema name for files in the corresponding section configuration (for example, *AccountFile*, *ContactFile*, etc.). The object schema name (the "entitySchemaName” property) of the *TimelineTileSetting* table should always look as follows: "##ReferenceSchemaName##File".

To add a new pre-configured tile:

1. Add a new section (if needed).

2. Add a module schema to your custom package and determine the tile view class, bound to the new section. The class should be the inheritor of *BaseTimelineItemView*.

3. Add a module schema to your custom package and determine the tile view model class, bound to the new section. The class should be the inheritor of *BaseTimelineItemViewModel*.

4. Add a record with the tile view settings bound to the new section into the *TimelineTileSetting* database table.

5. In the *TimelinePageSetting* table add or edit the record enabling the tile display on the [Timeline] tab in the necessary section.

Case description

Display the tiles bound to the [Books] custom section on the [Timeline] tab of the [Accounts] section. The tiles should contain:

- icon
- name
- author
- book record date
- price
- ISBN number

A short book description should also be displayed when you deploy the tile.

Source code

You can download the package with the [Book] section implementation using the following link. You can download the package with the tile module schema implementation using the following link.

> ATTENTION
>
> You should also edit the database tables to implement the case (see steps 4 and 5).

Case implementation algorithm

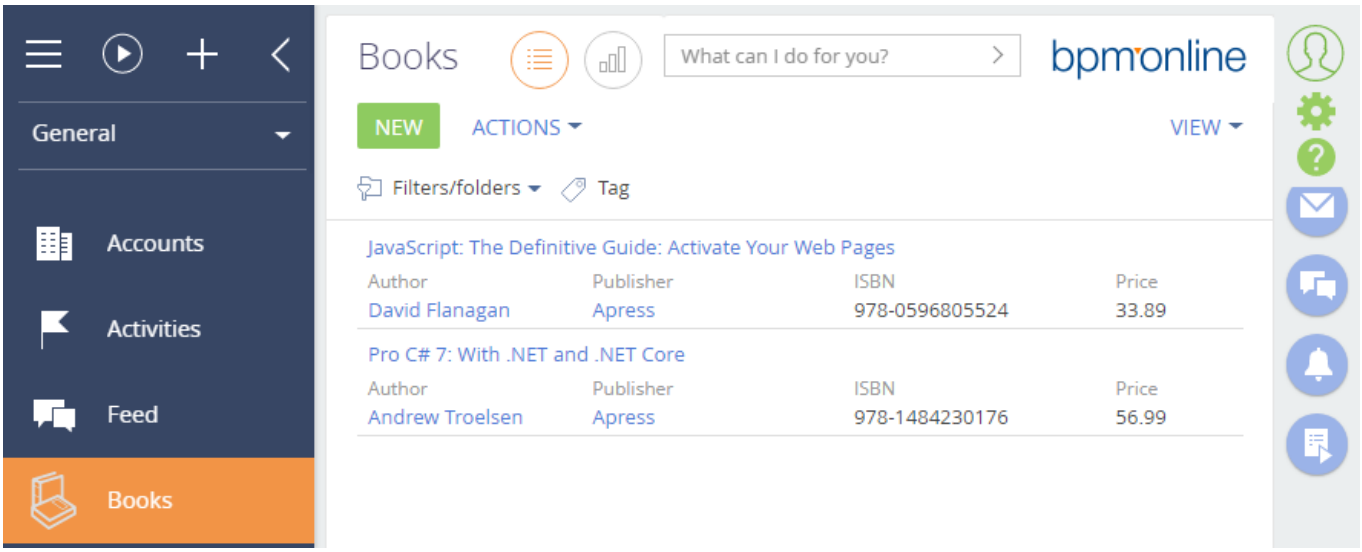
1. Adding a new [Books] section

Use the archive containing the needed function package to add the new [Books] section. Install the package via the marketplace application installation function from the *.zip-archive (see "**Installing marketplace applications from a zip archive**").

> NOTE
>
> You can also add the section via section wizard.

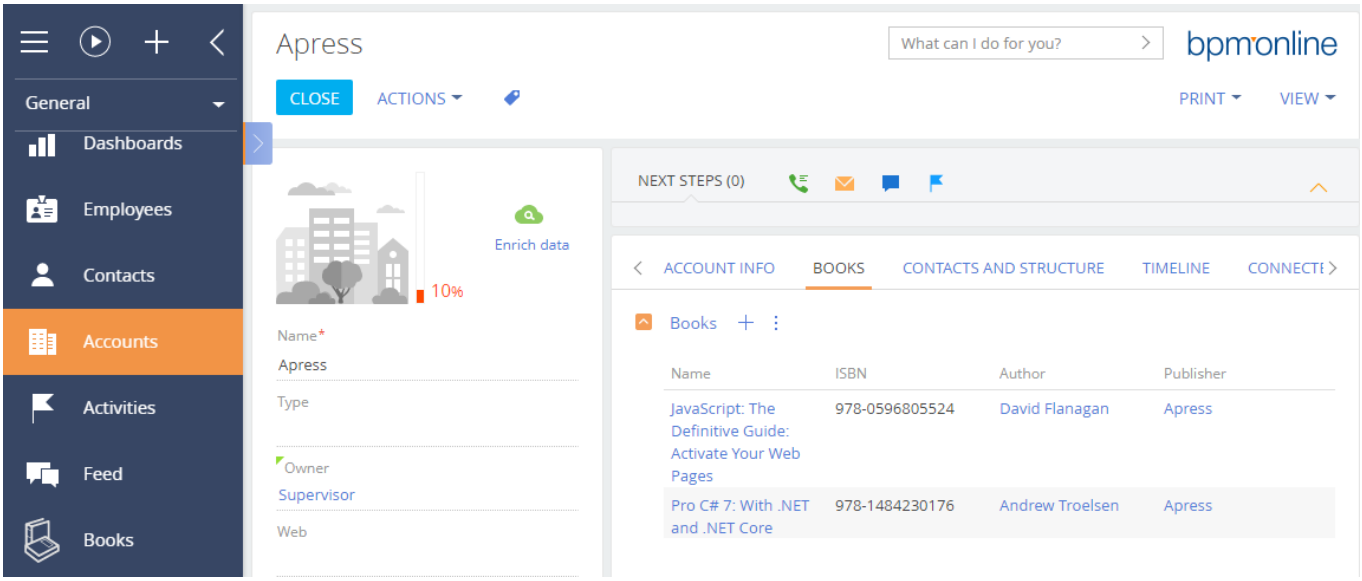
The [Books] section will be available in the [General] workplace after you install the package (Fig.1).

Fig. 1. The [Books] section



You will also see a detail displaying the linked records from the [Books] section on the [Books] tab of the [Accounts] section record edit page (Fig.2).

Fig. 2. The [Books] detail in the [Accounts] section



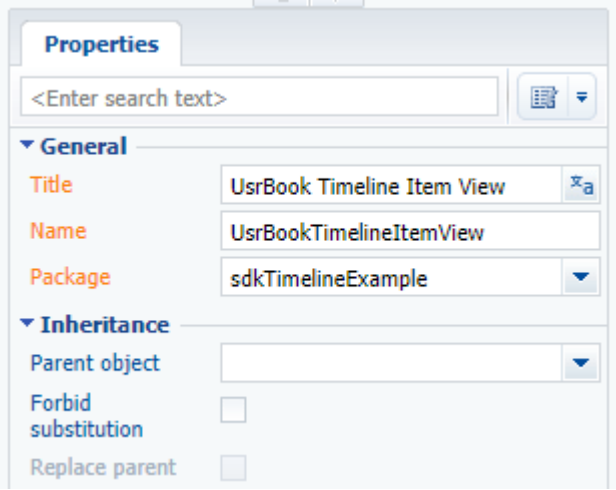
2. Adding a tile view module

Add a client module schema containing dependencies from the Timeline package to the custom package (see "**Creating a custom client module schema**").

For the created module schema specify (Fig. 3):

- [Name] – "UsrBookTimelineItemView"
- [Title] – "UsrBook Timeline Item View"

Fig. 3. Tile view module schema properties



Add the following module source code to the [Source code] tab of the schema:

```
// Defining the module and its dependencies.
define("UsrBookTimelineItemView", ["UsrBookTimelineItemViewResources",
"BaseTimelineItemView"], function() {
    // Defining the tile view class.
    Ext.define("Terrasoft.configuration.UsrBookTimelineItemView", {
        extend: "Terrasoft.BaseTimelineItemView",
        alternateClassName: "Terrasoft.UsrBookTimelineItemView",
        // Method returning the [UsrISBN] additional tile field configuration.
        getUsrISBNViewConfig: function() {
            return {
                // Field name.
                "name": "UsrISBN",
                // Field type — label.
```

```
                "itemType": Terrasoft.ViewItemType.LABEL,
                // Caption.
                "caption": {
                    "bindTo": "UsrISBN"
                },
                // Visibility.
                "visible": {
                    // Binding to the tile linked entity column.
                    "bindTo": "UsrISBN",
                    // Visibility setup.
                    "bindConfig": {
                        // A field is visible if its value is populated.
                        "converter": "checkIsNotEmpty"
                    }
                },
                // CSS field styles.
                "classes": {
                    "labelClass": ["timeline-text-light"]
                }
            };
        },
        // Method returning the [UsrPrice] additional tile field configuration.
        getUsrPriceViewConfig: function() {
            return {
                "name": "UsrPrice",
                "itemType": Terrasoft.ViewItemType.LABEL,
                "caption": {
                    "bindTo": "UsrPrice"
                },
                "visible": {
                    "bindTo": "UsrPrice",
                    "bindConfig": {
                        "converter": "checkIsNotEmpty"
                    }
                },
                "classes": {
                    "labelClass": ["timeline-item-subject-label"]
                }
            };
        },
        // Redefined method returning the [Message] tile field configuration.
        getMessageViewConfig: function() {
            // Receiving standard settings.
            var config = this.callParent(arguments);
            // Visibility setup. Visible if the tile is deployed.
            config.visible = {
                "bindTo": "IsExpanded"
            };
            return config;
        },
        // Redefined method returning general tile configuration.
        getBodyViewConfig: function() {
            // Receiving standard settings.
            var bodyConfig = this.callParent(arguments);
            // Adding additional field configurations.
            bodyConfig.items.unshift(this.getUsrISBNViewConfig());
            bodyConfig.items.unshift(this.getUsrPriceViewConfig());
            return bodyConfig;
        }
    });
});
```

Here you can define the configuration of the [UsrISBN] and [UsrPrice] fields that are additionally displayed on the tab. The standard configuration is defined in the *BaseTimelineItemView* module.

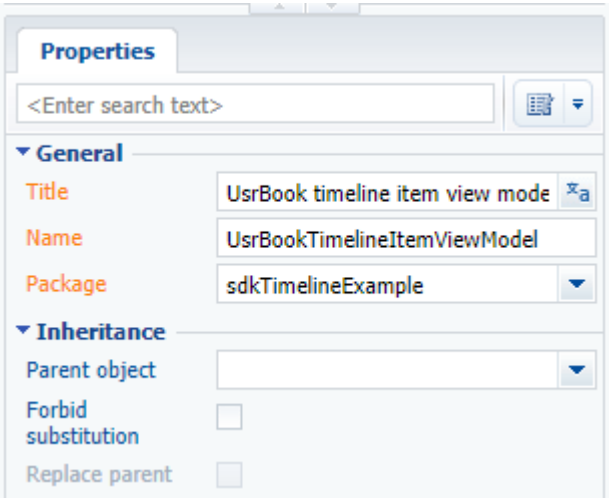
3. Adding a tile view model module

Add a client module schema containing dependencies from the Timeline package to the custom package (see "**Creating a custom client module schema**").

For the created module schema specify (Fig. 3):

- [Name] – "UsrBookTimelineItemViewModel"
- [Title] – "UsrBook timeline item view model"

Fig. 4. Module schema properties of the tile view model



Add the following module source code to the [Source code] tab of the schema:

```
define("UsrBookTimelineItemViewModel", ["UsrBookTimelineItemViewModelResources",
"BaseTimelineItemViewModel"],
    function() {
        Ext.define("Terrasoft.configuration.UsrBookTimelineItemViewModel", {
            alternateClassName: "Terrasoft.UsrBookTimelineItemViewModel",
            extend: "Terrasoft.BaseTimelineItemViewModel"
        });
    });
```

You define the *Terrasoft.configuration.UsrBookTimelineItemViewModel* class here. Since this class is defined as the inheritor of *Terrasoft.BaseTimelineItemViewModel*, it enables using all functions of the base class.

4. Adding the record with tile view settings to the TimelineTileSetting table

The TimelineTileSetting table is used to set up the timeline tile properties. The purpose of primary columns of this table is provided in table 2 of the "**The [Timeline] tab**" article.

Add a new record to the TimelineTileSetting table. You can add a new record via the following SQL query:

```
INSERT INTO TimelineTileSetting (CreatedOn, CreatedById, ModifiedOn, ModifiedById,
Name, Data, Image)
  VALUES (GETUTCDATE(), NULL, GETUTCDATE(), NULL, 'UsrBooks', NULL, NULL);
```

Since data in the *Data* and *Image* columns are stored in the *varbinary(max)* format, use specific editors (such as dbForge Studio Express for SQL Server) to modify them. To do this (Fig.5):

1. Select a table.

2. Select the necessary record column and click the edit button.

3. Enter the text data display mode in the data editor.

4. Add necessary data.

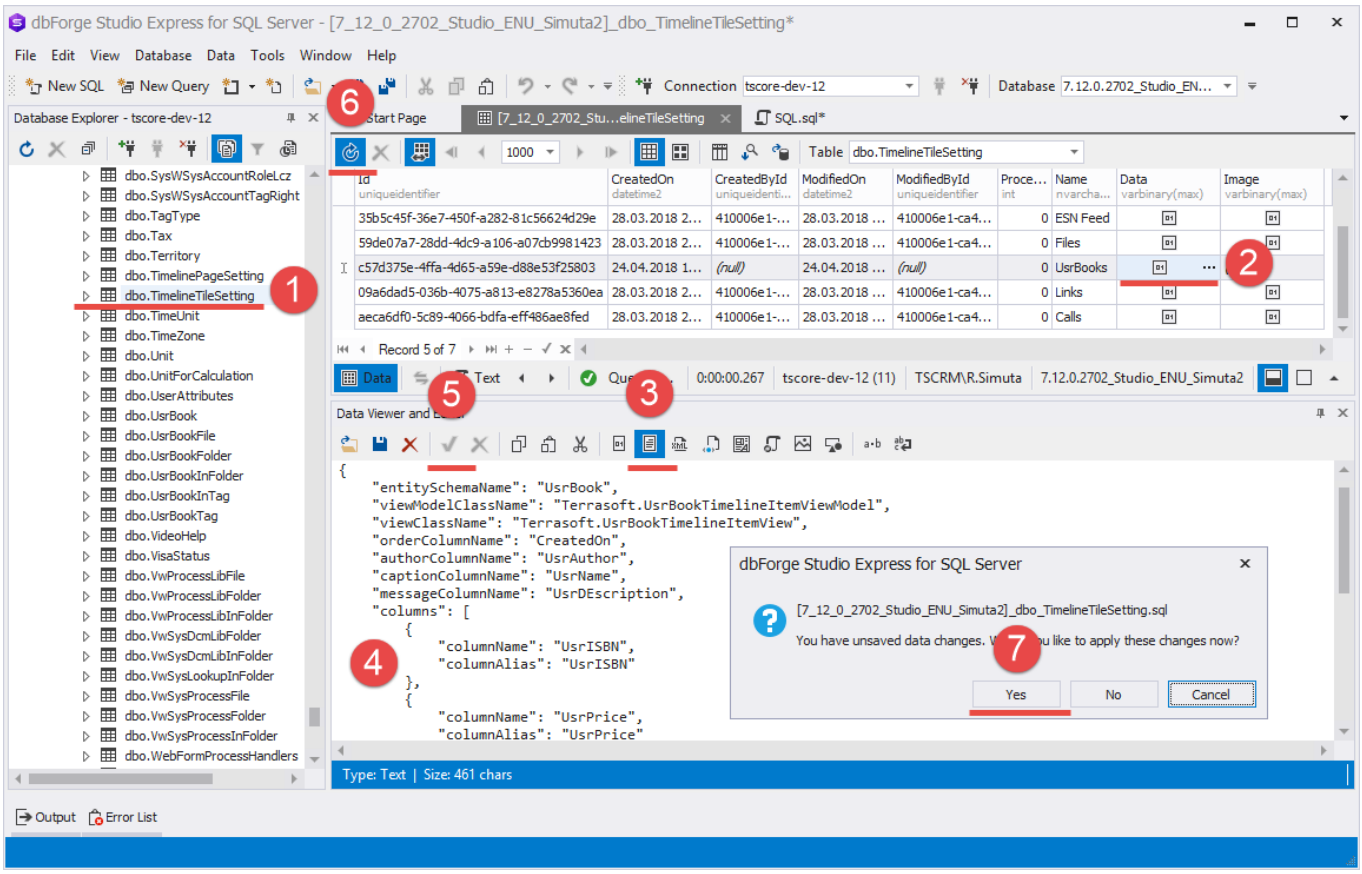
5. Save the changes in the data editor.

6. Click the data update button.

7. Click OK in the popped up checkout window to apply the modifications.

> **ATTENTION**
>
> This method is only good for the development environments deployed on-site. Since the modifications are implemented directly in the database, they are not bound to any package. That is why the modifications will not be implemented in the database if the package with the view models and the tile view models is installed into another application. For correct transfer of the developed functions you need to bind the SQL-scripts that implement the corresponding modifications in the database when installing the package.

Fig. 5. Editing data via the dbForge Studio Express for SQL Server



Add the following configuration object to the Data column using the above mentioned algorithm:

```
{
    "entitySchemaName": "UsrBook",
    "viewModelClassName": "Terrasoft.UsrBookTimelineItemViewModel",
    "viewClassName": "Terrasoft.UsrBookTimelineItemView",
    "orderColumnName": "CreatedOn",
    "authorColumnName": "UsrAuthor",
    "captionColumnName": "UsrName",
    "messageColumnName": "UsrDEscription",
    "columns": [
        {
            "columnName": "UsrISBN",
            "columnAlias": "UsrISBN"
        },
        {
            "columnName": "UsrPrice",
            "columnAlias": "UsrPrice"
```

```
        }
    ]
}
```

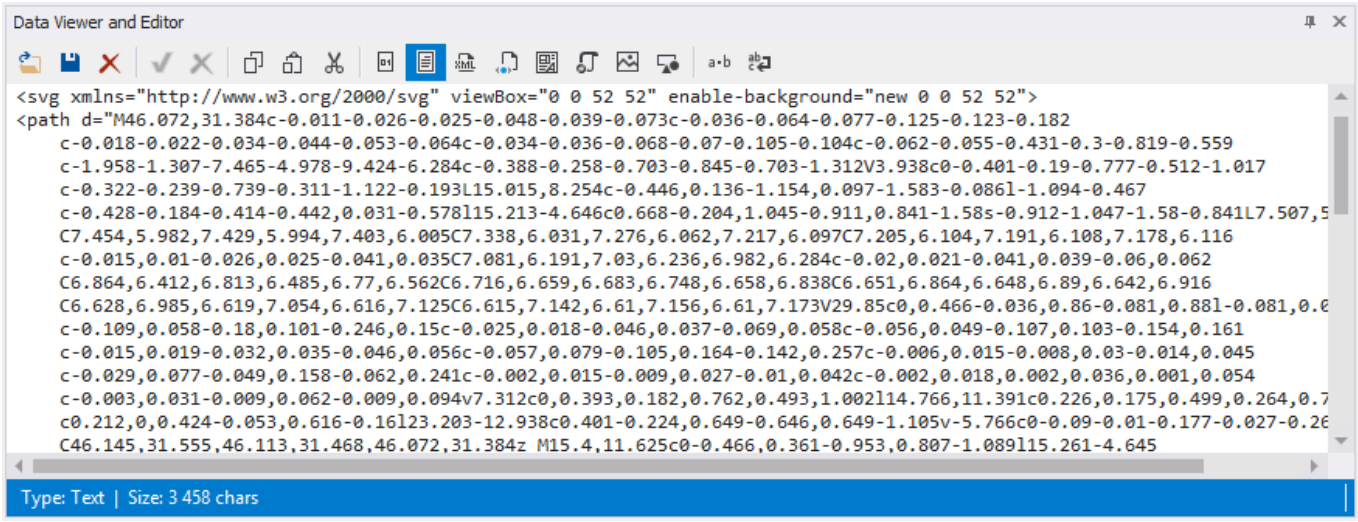
You need to indicate the additional field array whose display is configured in the *UsrBookTimelineItemView* view model in addition to the primary fields inherited from the base tile (see step 2).

Add SVG-format data to the *Image* column to display the icon that corresponds to the section icon (Fig.6).

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 52 52" enable-background="new 0
0 52 52">
<path d="M46.072,31.384c-0.011-0.026-0.025-0.048-0.039-0.073c-0.036-0.064-0.077-
0.125-0.123-0.182
    c-0.018-0.022-0.034-0.044-0.053-0.064c-0.034-0.036-0.068-0.07-0.105-0.104c-0.062-
0.055-0.431-0.3-0.819-0.559
    c-1.958-1.307-7.465-4.978-9.424-6.284c-0.388-0.258-0.703-0.845-0.703-
1.312V3.938c0-0.401-0.19-0.777-0.512-1.017
    c-0.322-0.239-0.739-0.311-1.122-0.193L15.015,8.254c-0.446,0.136-1.154,0.097-
1.583-0.086l-1.094-0.467
    c-0.428-0.184-0.414-0.442,0.031-0.578l15.213-4.646c0.668-0.204,1.045-0.911,0.841-
1.58s-0.912-1.047-1.58-0.841L7.507,5.961

C7.454,5.982,7.429,5.994,7.403,6.005C7.338,6.031,7.276,6.062,7.217,6.097C7.205,6.104,
7.191,6.108,7.178,6.116
    c-0.015,0.01-0.026,0.025-0.041,0.035C7.081,6.191,7.03,6.236,6.982,6.284c-
0.02,0.021-0.041,0.039-0.06,0.062

C6.864,6.412,6.813,6.485,6.77,6.562C6.716,6.659,6.683,6.748,6.658,6.838C6.651,6.864,6
.648,6.89,6.642,6.916

C6.628,6.985,6.619,7.054,6.616,7.125C6.615,7.142,6.61,7.156,6.61,7.173V29.85c0,0.466-
0.036,0.86-0.081,0.88l-0.081,0.036
    c-0.109,0.058-0.18,0.101-0.246,0.15c-0.025,0.018-0.046,0.037-0.069,0.058c-
0.056,0.049-0.107,0.103-0.154,0.161
    c-0.015,0.019-0.032,0.035-0.046,0.056c-0.057,0.079-0.105,0.164-0.142,0.257c-
0.006,0.015-0.008,0.03-0.014,0.045
    c-0.029,0.077-0.049,0.158-0.062,0.241c-0.002,0.015-0.009,0.027-0.01,0.042c-
0.002,0.018,0.002,0.036,0.001,0.054
    c-0.003,0.031-0.009,0.062-
0.009,0.094v7.312c0,0.393,0.182,0.762,0.493,1.002l14.766,11.391c0.226,0.175,0.499,0.2
64,0.773,0.264
    c0.212,0,0.424-0.053,0.616-0.16l23.203-12.938c0.401-0.224,0.649-0.646,0.649-
1.105v-5.766c0-0.09-0.01-0.177-0.027-0.261
    C46.145,31.555,46.113,31.468,46.072,31.384z M15.4,11.625c0-0.466,0.361-
0.953,0.807-1.089l15.261-4.645
    c0.446-0.136,0.807,0.132,0.807,0.598v14.63c0,0.467-0.314,0.635-0.702,0.376l-
1.127-0.752c-0.361-0.24-0.819-0.278-1.216-0.104
    l-13.059,5.805c-0.426,0.189-0.771-0.034-0.771-
0.501C15.4,25.943,15.4,11.625,15.4,11.625z M28.851,23.579
    c0.425-0.189,1.085-0.134,1.473,0.125l11.43,7.62c0.388,0.259,0.368,0.644-
0.045,0.86l-18.404,9.662
    c-0.412,0.216-1.047,0.163-1.418-0.121l-11.789-9.001c-0.371-0.283-0.326-0.665,0.1-
0.854L28.851,23.579z M9.142,9.932
    c0-0.466,0.348-0.695,0.776-
0.512l2.174,0.929c0.429,0.183,0.776,0.708,0.776,1.175v2.158c-1.57-0.068-2.894-0.916-
3.727-1.61
    L9.142,9.932L9.142,9.932z
M9.142,13.152c0.931,0.671,2.22,1.323,3.727,1.372v7.633c-1.57-0.066-2.894-0.915-3.727-
1.609
    C9.142,20.548,9.142,13.152,9.142,13.152z
M9.142,21.627c0.931,0.671,2.22,1.323,3.727,1.372v3.992c0,0.466-0.35,0.985-0.782,1.16
    l-2.163,0.876c-0.432,0.175-0.782-0.061-0.782-0.527V21.627z
```

```
M43.666,36.101c0,0.467-0.33,1.027-0.737,1.255L22.578,48.702
    c-0.407,0.228-1.036,0.18-1.405-0.104L8.897,39.127c-0.369-0.284-0.668-0.893-0.668-
1.358v-2.444c0-0.466,0.3-0.614,0.671-0.332
    l12.764,9.748c0.225,0.171,0.496,0.26,0.768,0.26c0.201,0,0.403-0.048,0.588-
0.146l19.899-10.447
    c0.413-0.217,0.747-0.015,0.747,0.452V36.101z" style="fill:#6c91de;"/>
<path d="M33.81,34.064c0.072,0.049,0.155,0.073,0.239,0.073c0.072,0,0.145-0.018,0.209-
0.055l4.505-2.575
    c0.126-0.072,0.207-0.204,0.212-0.349c0.006-0.146-0.063-0.283-0.183-0.365l-9.011-
6.192c-0.118-0.08-0.268-0.097-0.399-0.042
    l-5.157,2.123c-0.143,0.059-0.243,0.191-0.259,0.346c-
0.017,0.154,0.053,0.304,0.181,0.392L33.81,34.064z M29.492,25.426
    l8.269,5.682l-3.692,2.111l-8.803-6.052L29.492,25.426z" style="fill:#6c91de;"/>
</svg>
```

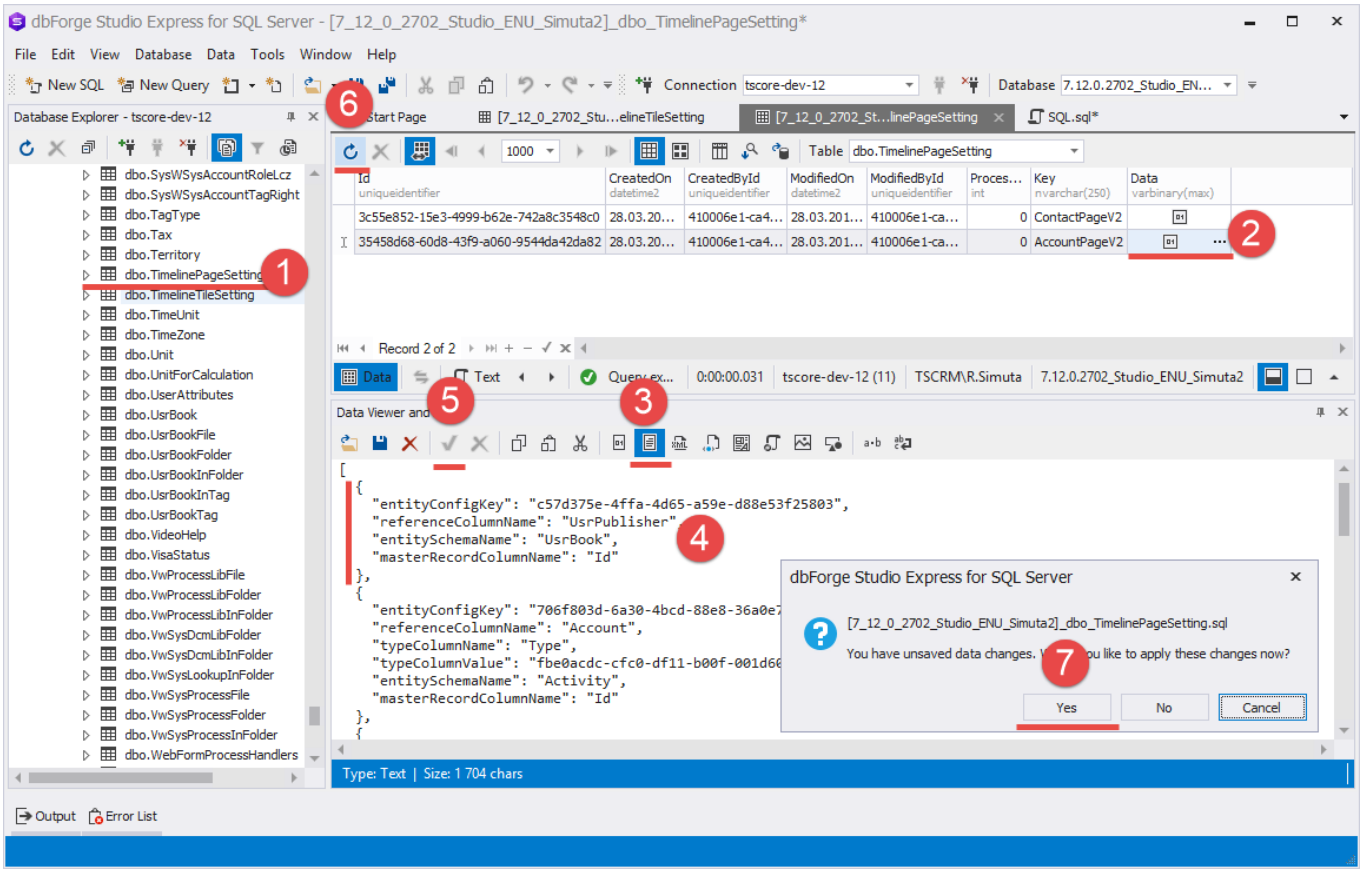
Fig. 6. Editing the Image column data via the dbForge Studio Express for SQL Server



5. Editing the record that enables the tile display on the [Timeline] tab of the account page in the TimelinePageSettings table.

For the [Accounts] section there already exists a record in the *TimelineTileSettings* table with settings of tiles bound to other sections. This is the record containing the "AccountPageV2" value in the *Key* column (Fig.7).

Fig. 7. Editing the Image column data via the dbForge Studio Express for SQL Server



> **ATTENTION**
>
> Since there are several tiles used on the [Accounts] section page timeline, the array of configuration objects enabling the corresponding tile is stored in the Data column.

Using the algorithm mentioned in step 4, change the configuration object array by adding a new record to it.

```
[
  {
    "entityConfigKey": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
    "referenceColumnName": "UsrPublisher",
    "entitySchemaName": "UsrBook",
    "masterRecordColumnName": "Id"
  },
  ...
]
```

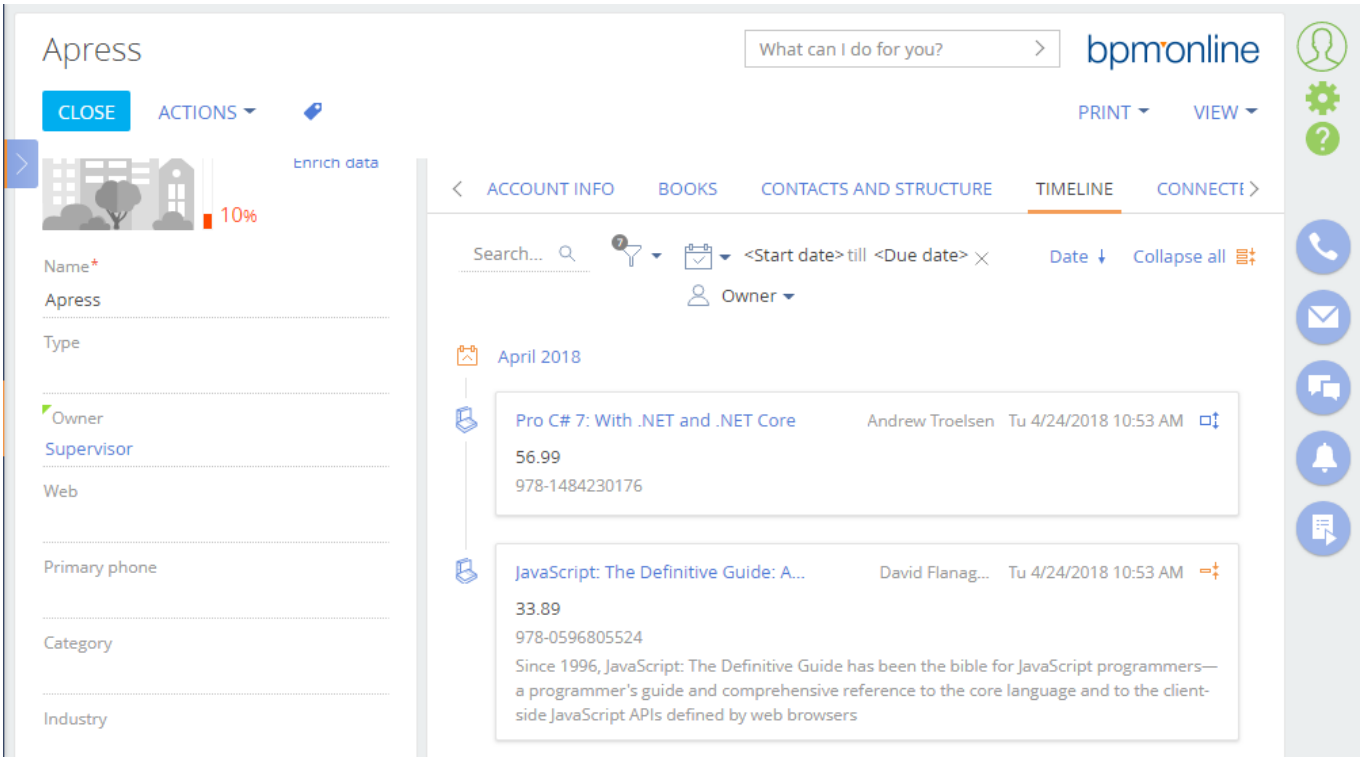
Here the "entityConfigKey" property: "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx" should contain the TimelineTileSettings table record identifier created on step 4. In our case it is the "c57d375e-4ffa-4d65-a59e-d88e53f25803" value (Fig.5 and Fig.7).

> **ATTENTION**
>
> The "entityConfigKey" should be obligatory indicated. It should match the *Id* column value of the record containing settings of the necessary tile in the *TimelineTileSettings* table.
>
> Since the identifiers of the added records are generated at random, the generated identifier in your database will be different from the one we have in our case, when you repeat step 4.

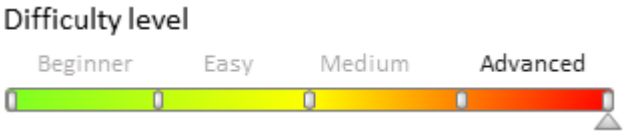
> **ATTENTION**
>
> Be careful when modifying the *Data* column value. Incorrect modifications can disrupt the operation of all existing timeline tiles in a section.

As a result of case implementation you will have the tiles bound to the [Book] custom section displayed on the [Timeline] tab of the [Accounts] section page. These tiles contain all the fields we described in our case conditions. The short book description will only be displayed when you deploy the tile (Fig.8).

Fig. 8. Case result



Adding multi-language email templates to a custom section



Introduction

You can set up custom logic for selecting languages of multi-language email templates. You can select email templates in the needed language using the action dashboard of a section record. The selection is based on special rules that can be specified for the section. If special rules are not defined, the selection is based on the contact, bound to the edited record (the [Contact] column). If a section object does not have a column for connecting with a contact, the *DefaultMessageLanguage* system setting value is used.

To add custom logic for selecting multi-language templates (localization):

1. Create a class or classes inherited from *BaseLanguageRule* and define the language selection rules (one class defines one rule).

2. Create a class inherited from *BaseLanguageIterator*. Define the *LanguageRules* property in the class constructor as a class instance array created on the previous step. The sequence corresponds to the rule priority.

3. Create a class inherited from *AppEventListenerBase* that will bind the class defining the language selection rules to the section.

4. Add the necessary multi-language templates to the [Email templates] lookup.

Case description

Add logic of selecting an email template language to a custom section based on the *UsrContact* column of the primary section object. Use English and Spanish languages.

Source code

You can download the package with case implementation using the following link.

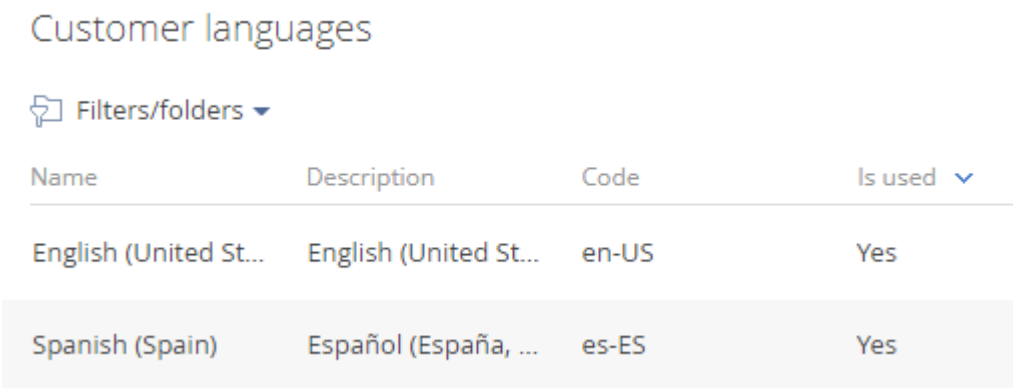
> ⚠ ATTENTION
>
> You can install the package for bpm’online products, containing the *EmailTemplates* package. Make sure all the below described preliminary settings are performed after you install the package.

Preliminary settings

For correct case implementation:

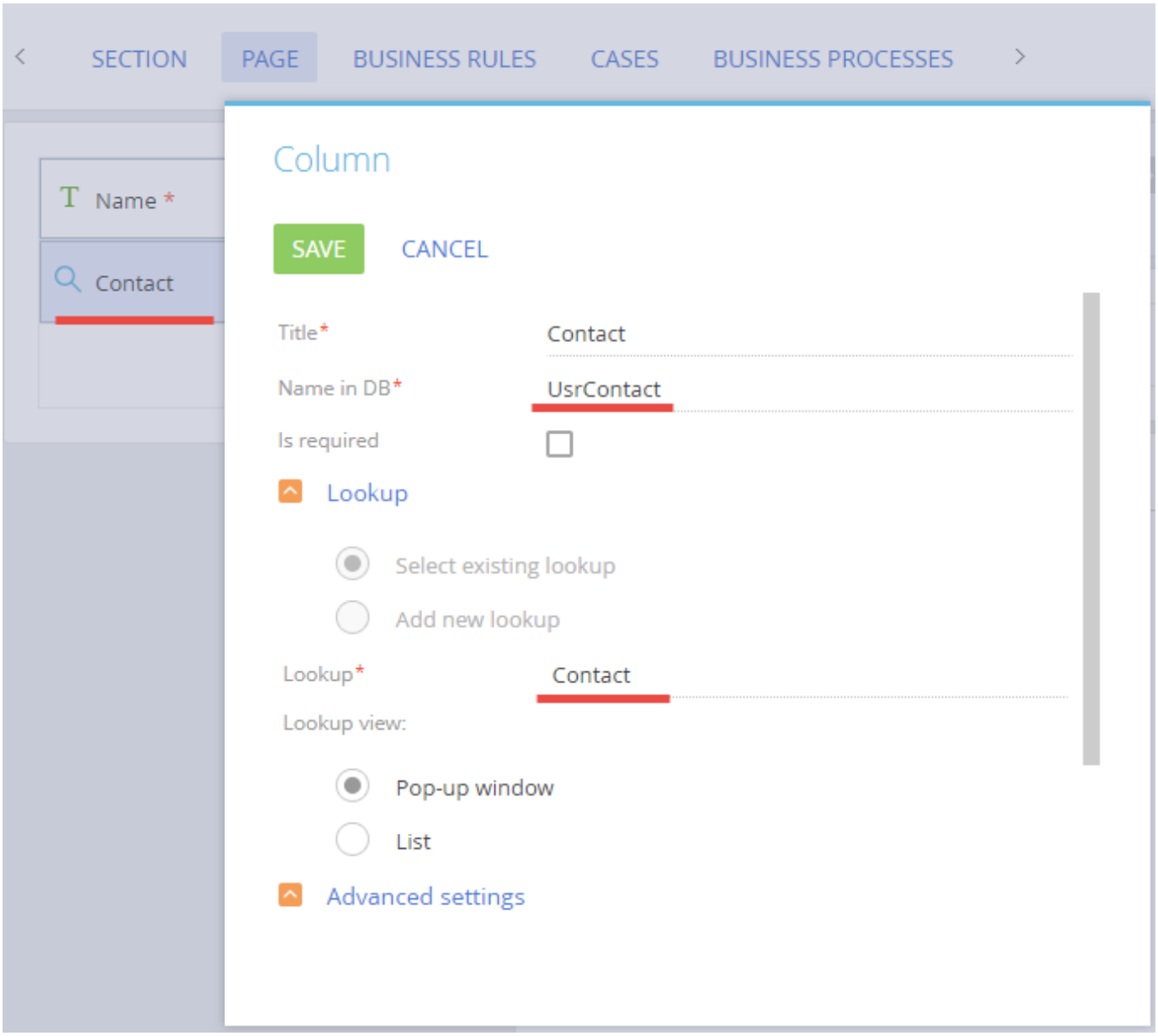
1. Make sure that the [Customer languages] lookup contains English and Spanish languages (Fig.1).

Fig. 1. [Customer languages] lookup



2. Use the section wizard to check that there is the *UsrContact* column bound to the [Contact] lookup on the edit page of the custom section record (Fig.2).

Fig. 2. The UsrContact column



Case implementation algorithm

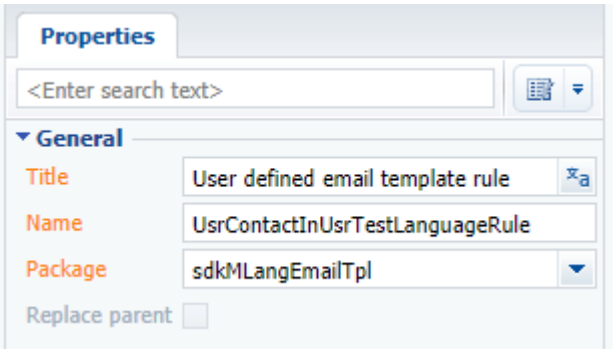
1. Adding a language selection rule

Create a [Source code] schema in the custom package (see “**Creating the [Source code] schema**”).

For the created schema specify (Fig. 3):

- [Name] – "UsrContactInUsrTestLanguageRule"
- [Title] – "User defined email template rule”

Fig. 3. The [Source code] schema properties



Add the following source code to the schema:

```
namespace Terrasoft.Configuration
{
    using System;
    using Terrasoft.Core;
    using Terrasoft.Core.Entities;
    public class ContactInUsrTestLanguageRule : BaseLanguageRule
    {
        public ContactInUsrTestLanguageRule (UserConnection userConnection) :
base(userConnection)
        {
        }
        // Defines the user preferred language identifier.
        // recId – current record identifier.
        public override Guid GetLanguageId(Guid recId)
        {
            // Creating the EntitySchemaQuery instance for the custom section primary
object.
            var esq = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"UsrMLangEmailTpl");
            // Defining the contact language column name.
            var languageColumnName = esq.AddColumn("UsrContact.Language.Id").Name;
            // Obtaining current record instance.
            Entity usrRecEntity = esq.GetEntity(UserConnection, recId);
            // Obtaining the value of user preferred language identifier.
            Guid languageId = usrRecEntity.GetTypedColumnValue<Guid>
(languageColumnName);
            return languageId;
        }
    }
}
```

Publish the schema.

2. Defining the order sequence of language selection rules

Create a [Source code] schema in the custom package (see "**Creating the [Source code] schema**").

For the created schema specify (Fig. 3):

- [Name] – "UsrTestLanguageIterator"
- [Title] – "User defined language iterator"

Add the following source code to the schema:

```
namespace Terrasoft.Configuration
{
    using Terrasoft.Core;
    public class UsrTestLanguageIterator: BaseLanguageIterator
    {
        public UsrTestLanguageIterator(UserConnection userConnection):
```

```
base(userConnection)
        {
            // Language selection rule array.
            LanguageRules = new ILanguageRule[] {
                // Custom rule.
                new ContactInUsrTestLanguageRule (UserConnection),
                // Default rule.
                new DefaultLanguageRule(UserConnection),
            };
        }
    }
}
```

DefaultLanguageRule. is the second array element The rules uses the *DefaultLanguage* system setting for obtaining the language and is used by default if the language was not detected by higher priority rules.

Publish the schema.

3. Binding language selection iterator to the section

Create a [Source code] schema in the custom package (see "**Creating the [Source code] schema**").

For the created schema specify (Fig. 3):

- [Name] – "UsrTestMLangBinder"
- [Title] – "UsrTestMLangBinder"

Add the following source code to the schema:

```
namespace Terrasoft.Configuration
{
    using Terrasoft.Core.Factories;
    using Terrasoft.Web.Common;
    public class UsrTestMLangBinder: AppEventListenerBase
    {
        public override void OnAppStart(AppEventContext context)
        {
            // Calling the basic logics.
            base.OnAppStart(context);
            // Binding iterator to a custom section.
            // UsrMLangEmailTpl — name of the section primary object.
            ClassFactory.Bind<ILanguageIterator, UsrTestLanguageIterator>
("UsrMLangEmailTpl");
        }
    }
}
```

Publish the schema.

4. Adding the necessary multi-language templates

Add a new record (Fig.4) to the [Email Templates] lookup and define the email templates in the necessary languages (Fig.5).

Fig. 4. A new record in the [Email templates] lookup

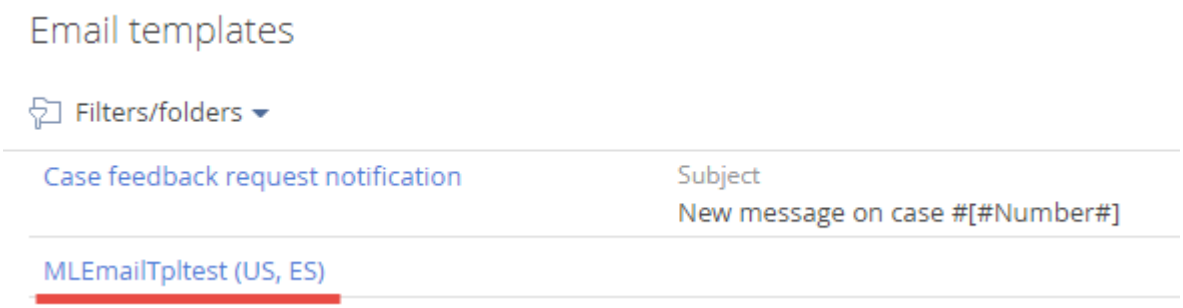
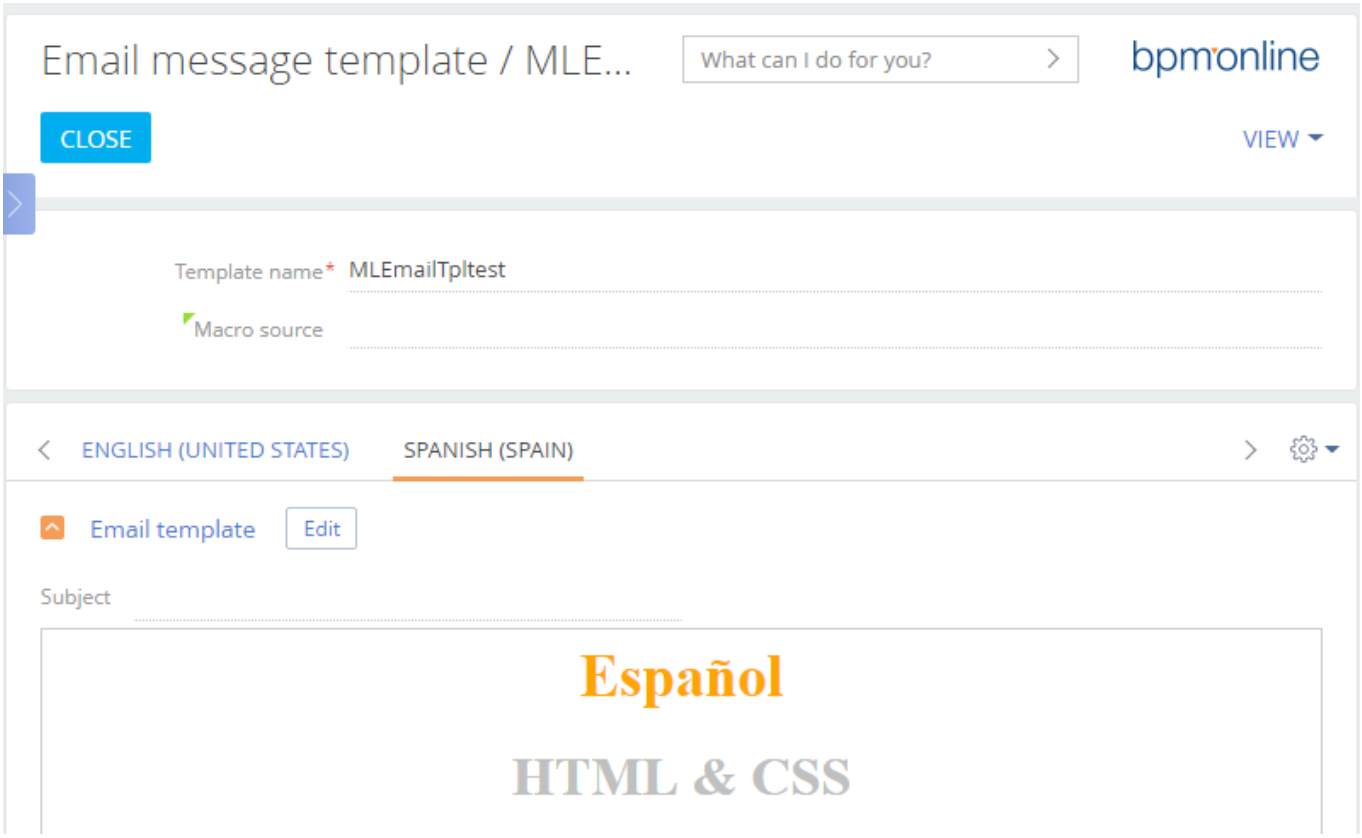


Fig. 5. Adding templates in the necessary languages



As a result of case implementation, in the action dashboard panel (Fig. 6. 1) of the custom section record edit page (Fig.6) the email templates (Fig. 6. 2) will be selected automatically in the language (Fig. 6. 3) specified as the contact’s preferred language (Fig.7).

Fig. 6. Case result

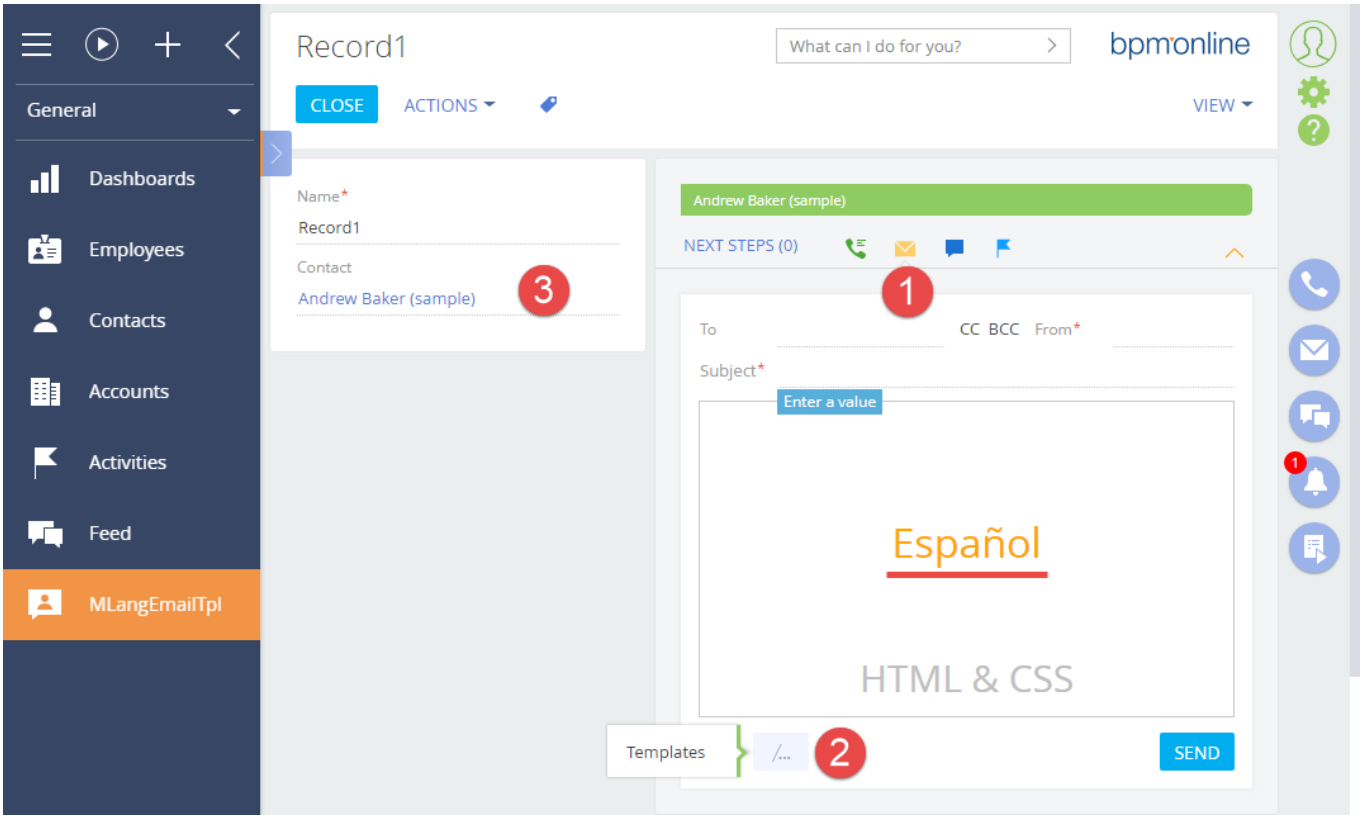
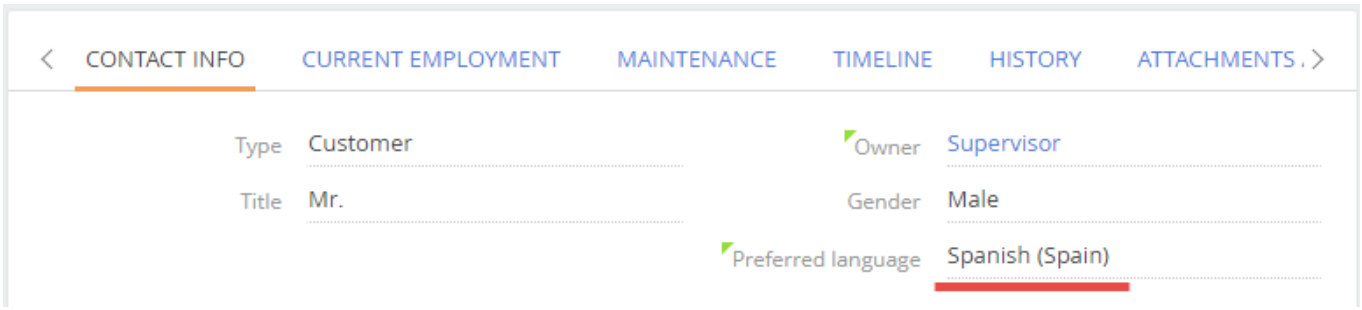


Fig. 7. Contact preferred language

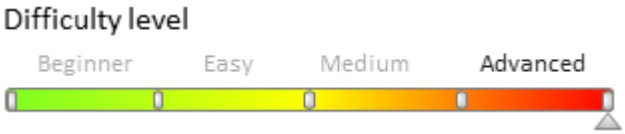


Analytics

Contents

- **How to create macros for a custom report in Word**

How to create macros for a custom report in Word



Introduction

You can set up a printable using standard BPMonline MS Word Report Designer tools. MS Word printable setup is covered in the "Setting up MS Word printables in bpm'online" article.

To implement specific printable setup tasks, you need to use macros. Basic macros are covered in the "**Basic**

macros in the MS Word printables” article. A macro for setting up a printable is a class implementing the *IExpressionConverter* interface (see the *ExpressionConverter* schema of the *NUI* package).

To be able to call a custom macro from the printable template, it must be marked by the *ExpressionConverterAttribute* attribute with a specified name. Example:

```
[ExpressionConverterAttribute("CurrentUser")]
```

The *Evaluate(object value, string arguments = "")* interface method must be implemented in the class. The method accepts a printable template field value as an argument and returns the *string* type value that will be inserted instead of this field in the ready printable.

Algorithm of creating a custom macro for a printable

1. Create a printable and add the [Id] column to the list of printable columns that will be the incoming parameter for the macro.
2. Add the source code module to the custom package, where you need to describe the class-inheritor of the *IExpressionConverter* interface. The class must be marked by the ExpressionConverterAttribute attribute with the name of the macro. Implement the *Evaluate(object value, string arguments = "")* method in it.
3. Add a tag with the name of the macro in the [#MacrosName#] format to the [Id] column in the printable template.
4. Publish the source code module and download the updated printable template into the printable.

Case description

Create the [Account summary] printable for the [Accounts] section edit page that will contain the [Name], [Type] and [Primary contact] general fields and the [Additional information] field, which will display the annual turnover for the [Customer] type accounts, and the number of employees for the [Partner] type accounts. Besides, the printable must contain information about the date of creation and the name of employee who created it.

Source code

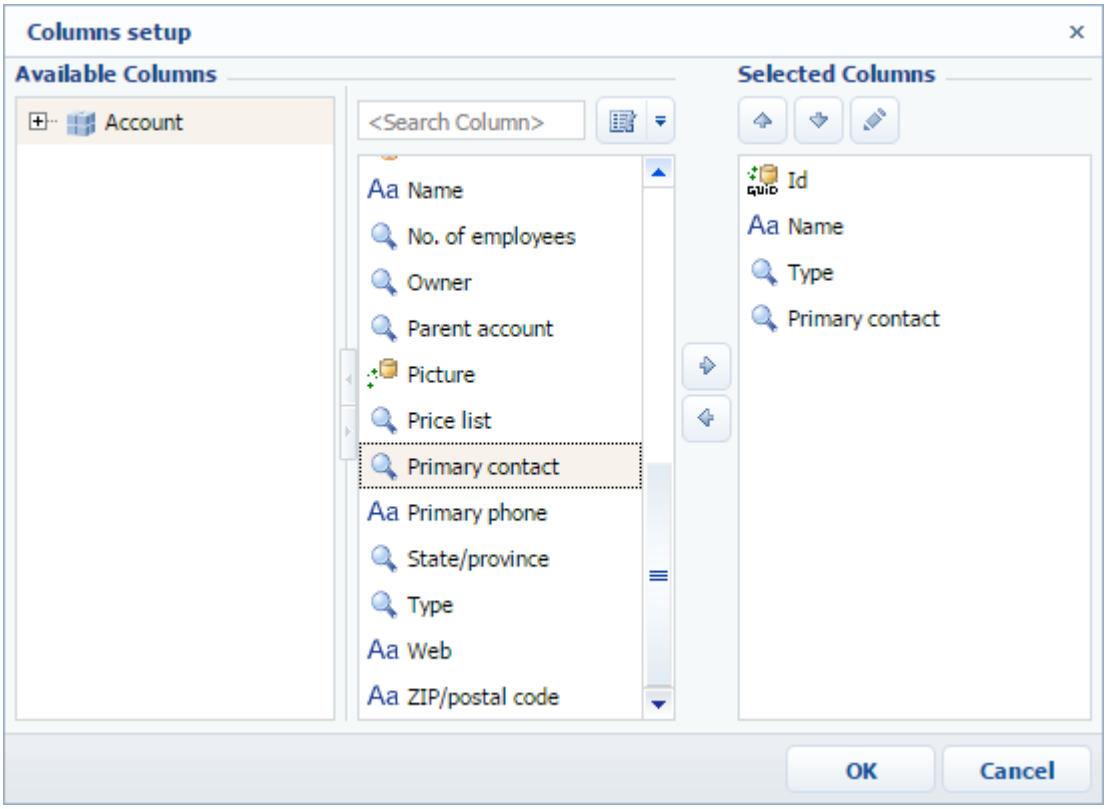
You can download the package with case implementation using the following llink.

Case implementation algorithm

1. Creating the [Account summary] printable for the [Accounts] section edit page

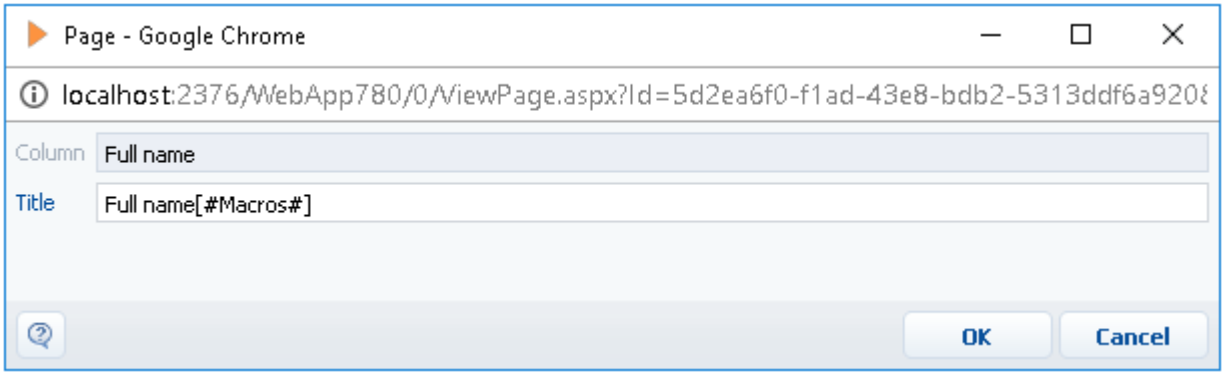
Add the [Id], [Name], [Type], [Primary contact] columns to the list of columns.

Fig. 1. Column list of the printable



You can add the necessary macros to columns at the stage of their setup. To do this, go to the [Selected columns] field and select the necessary column. Click the edit button and add a macro in the opened column edit window (fig.2).

Fig. 2. Adding a macro to the column

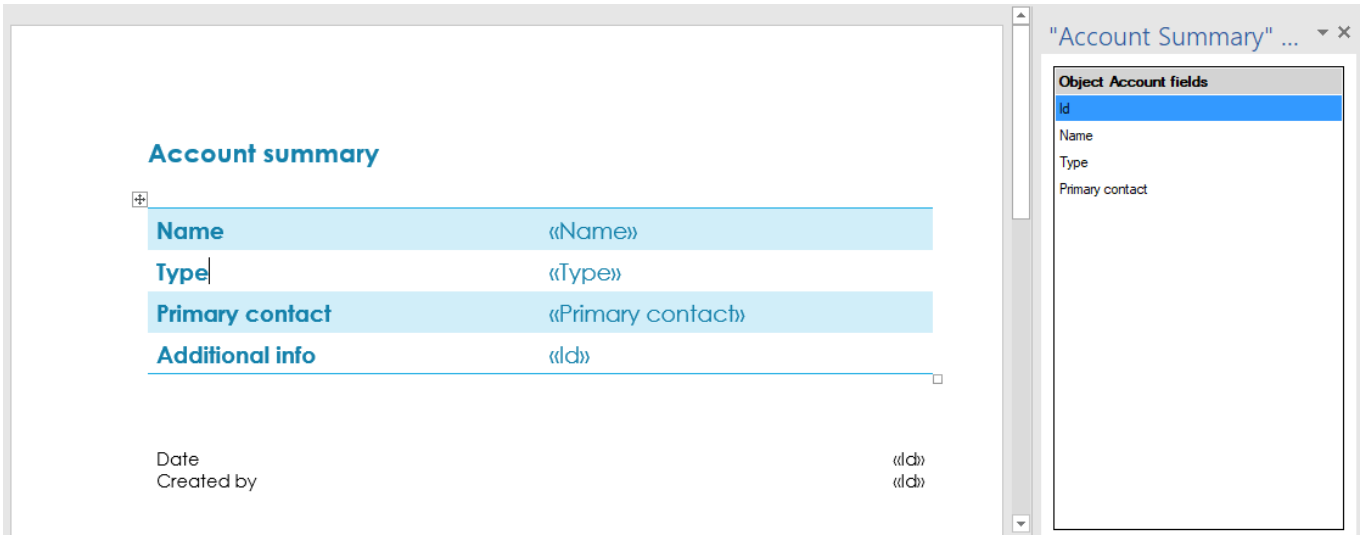


Creating the MS Word printable is covered in the “Setting up MS Word printables in bpm’online” article.

2. Exporting the printable template and locating the fields

After you add the fields, your template may look as shown in fig.3. Editing the MS Word printable is covered in the “Setting up MS Word printables in bpm’online” article.

Fig. 3. Locating fields in the template



3. Adding source code modules to the custom package that would implement macros

3.1 Macro of receiving additional information depending on the account type

Add the source code type schema (see **Creating the [Source code] schema**) with the following properties:

- [Name] – "UsrAccountInfoByTypeConverter"
- [Title] – "AccountInfoByTypeConverter"

Add localizable strings, whose properties are listed in table 1 to the schema.

Table 1. Localizable string properties

Name	Value
PartnerAdditional	"Number of employees {0} persons"
CustomerAdditional	"Annual turnover {0}"

Create a macro class in the schema source code for receiving additional information depending on the account type:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.CodeDom.Compiler;
    using System.Collections.Generic;
    using System.Data;
    using System.Linq;
    using System.Runtime.Serialization;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Web;
    using Terrasoft.Common;
    using Terrasoft.Core;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Entities;
    using Terrasoft.Core.Packages;
    using Terrasoft.Core.Factories;

    // The [AccountInfoByType] attribute with the macro name.
    [ExpressionConverterAttribute("AccountInfoByType")]
    // The class must implement the IExpressionConverter interface.
```

```
    class AccountInfoByTypeConverter : IExpressionConverter
    {
        private UserConnection _userConnection;
        private string _customerAdditional;
        private string _partnerAdditional;
        // Calling the values of localizable strings
        private void SetResources() {
            string sourceCodeName = "UsrAccountInfoByTypeConverter";
            _customerAdditional = new
LocalizableString(_userConnection.ResourceStorage, sourceCodeName,
                "LocalizableStrings.CustomerAdditional.Value");
            _partnerAdditional =  new
LocalizableString(_userConnection.ResourceStorage, sourceCodeName,
                "LocalizableStrings.PartnerAdditional.Value");
        }
        // Implementing the Evaluate method of the IExpressionConverter interface.
        public string Evaluate(object value, string arguments = "")
        {
            try
            {
                _userConnection =
(UserConnection)HttpContext.Current.Session["UserConnection"];
                Guid accountId = new Guid(value.ToString());
                return getAccountInfo(accountId);
            }
            catch (Exception err)
            {
                return err.Message;
            }
        }
        // Method of receiving additional infomration depending on the account type.
        // The account Id is used as the incoming parameter.
        private string getAccountInfo(Guid accountId)
        {
            try
            {
                // Saving the EntitySchemaQuery class object with the  [Account] root
schema.
                EntitySchemaQuery esq = new
EntitySchemaQuery(_userConnection.EntitySchemaManager, "Account");
                // Adding the [Name] column to the schema from the [Type] lookup
field.
                var columnType = esq.AddColumn("Type.Name").Name;
                // Adding the [Name] column to the schema from the [EmployeesNumber]
lookup field.
                var columnNumber = esq.AddColumn("EmployeesNumber.Name").Name;
                // Adding the [Name] column to the schema from the[AnnualRevenue]
lookup field.
                var columnRevenue = esq.AddColumn("AnnualRevenue.Name").Name;
                // Records are filtered by the account Id.
                var accountFilter = esq.CreateFilterWithParameters(
                    FilterComparisonType.Equal,
                    "Id",
                    accountId
                );
                esq.Filters.Add(accountFilter);
                // Receiving the entity collection.
                EntityCollection entities = esq.GetEntityCollection(_userConnection);
                // If the collection contains elements, the method returns
correspondent information
                // depending on the account type.
                if (entities.Count > 0)
```

```
                {
                    Entity entity = entities[0];
                    var type = entity.GetTypedColumnValue<string>(columnType);
                    switch (type)
                    {
                        case "Customer":
                            return String.Format(_customerAdditional,
entity.GetTypedColumnValue<string>(columnRevenue));
                        case "Partner":
                            return String.Format(_partnerAdditional,
entity.GetTypedColumnValue<string>(columnNumber));
                        default:
                            return String.Empty;
                    }
                }
                return String.Empty;
            }
            catch (Exception err)
            {
                throw err;
            }
        }
    }
}
```

Publish the module.

3.2 Macro of receiving the current date

Add the source code type schema (see **Creating the [Source code] schema**) with the following properties:

- [Name] – “UsrCurrentDateConverter”
- [Title] – “CurrentDateConverter”

Create a macro class in the schema source code for receiving the current date:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.CodeDom.Compiler;
    using System.Collections.Generic;
    using System.Data;
    using System.Linq;
    using System.Runtime.Serialization;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Web;
    using Terrasoft.Common;
    using Terrasoft.Core;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Entities;
    using Terrasoft.Core.Packages;
    using Terrasoft.Core.Factories;

    // The  [CurrentDate] attribute with the macro name.
    [ExpressionConverterAttribute("CurrentDate")]
    // The class must implement the IExpressionConverter interface.
    class CurrentDateConverter : IExpressionConverter
    {
        private UserConnection _userConnection;
```

```
        // Implementing the Evaluate method of the IExpressionConverter interface.
        public string Evaluate(object value, string arguments = "")
        {
            try
            {
                _userConnection =
(UserConnection)HttpContext.Current.Session["UserConnection"];
                // The method returns the current date.
                return
_userConnection.CurrentUser.GetCurrentDateTime().Date.ToString("d MMM yyyy");
            }
            catch (Exception err)
            {
                return err.Message;
            }
        }
    }
}
```

Publish the module.

3.3 Macro of receiving the current user

Add the source code type schema (see **Creating the [Source code] schema**) with the following properties:

- [Name] – “UsrCurrentUserConverter”
- [Title] – “CurrentUserConverter”

Create a macro class in the schema source code for receiving the current user:

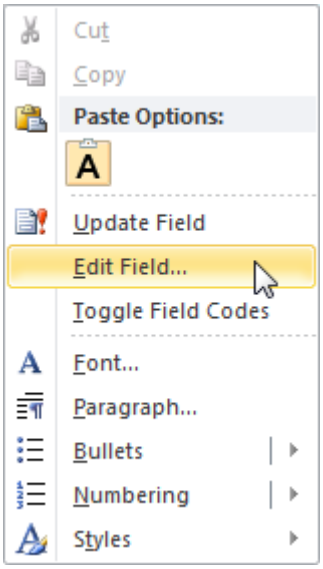
```
namespace Terrasoft.Configuration
{
    using System;
    using System.CodeDom.Compiler;
    using System.Collections.Generic;
    using System.Data;
    using System.Linq;
    using System.Runtime.Serialization;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Web;
    using Terrasoft.Common;
    using Terrasoft.Core;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Entities;
    using Terrasoft.Core.Packages;
    using Terrasoft.Core.Factories;

    // The [CurrentUser] attribute with the macro name.
    [ExpressionConverterAttribute("CurrentUser")]
    // The class must implement the IExpressionConverter interface.
    class CurrentUserConverter : IExpressionConverter
    {
        private UserConnection _userConnection;
        // Implementing the Evaluate method of the IExpressionConverter interface.
        public string Evaluate(object value, string arguments = "")
        {
            try
            {
```

```
                _userConnection =
(UserConnection)HttpContext.Current.Session["UserConnection"];
                // The method returns the contact of the current user.
                return _userConnection.CurrentUser.ContactName;
            }
            catch (Exception err)
            {
                return err.Message;
            }
        }
    }
}
```

4. Adding tags with the macro names to the report template fields

Select the [Edit Field] option in the template field context menu (Fig. 4).

Fig. 4. A template field context menu



Add tags with macro names to the field names (fig. 5, 6, 7).

> NOTE
>
> You can use any template field to receive data about the current date and user, since the handler-macros for this data do not use incoming parameters. You can use the [Id] field in both cases of the case.

Fig. 5. Adding the AccountInfoByType tag for receiving additional information about the account

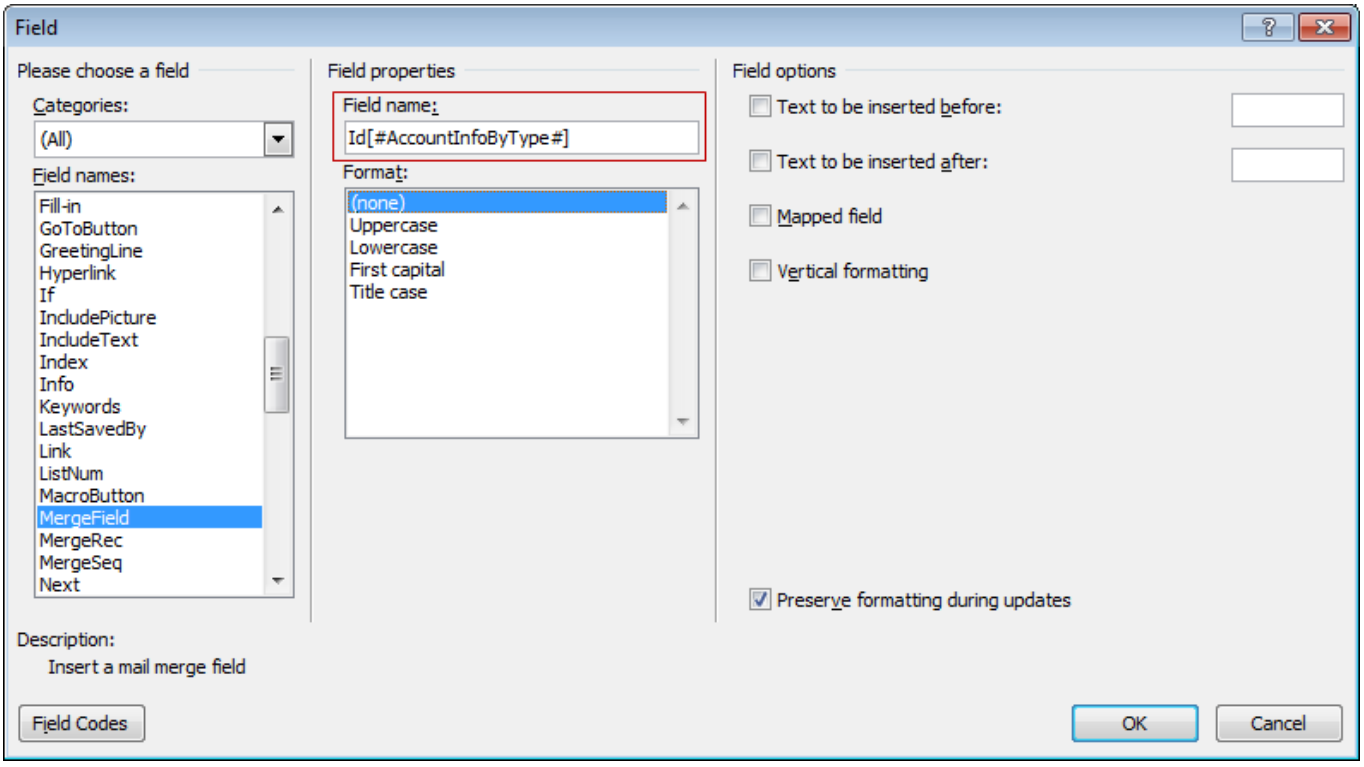


Fig. 6. Adding the CurrentDate tag for receiving the current date

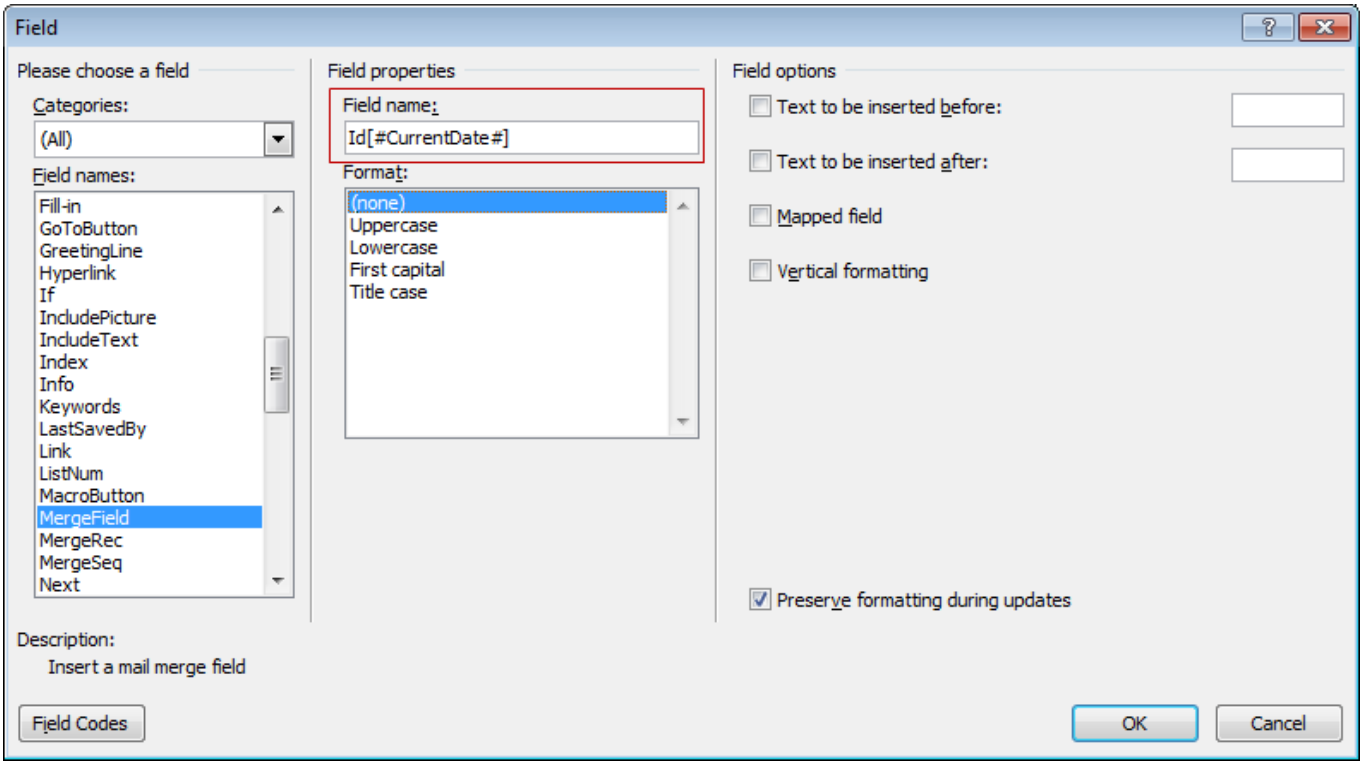
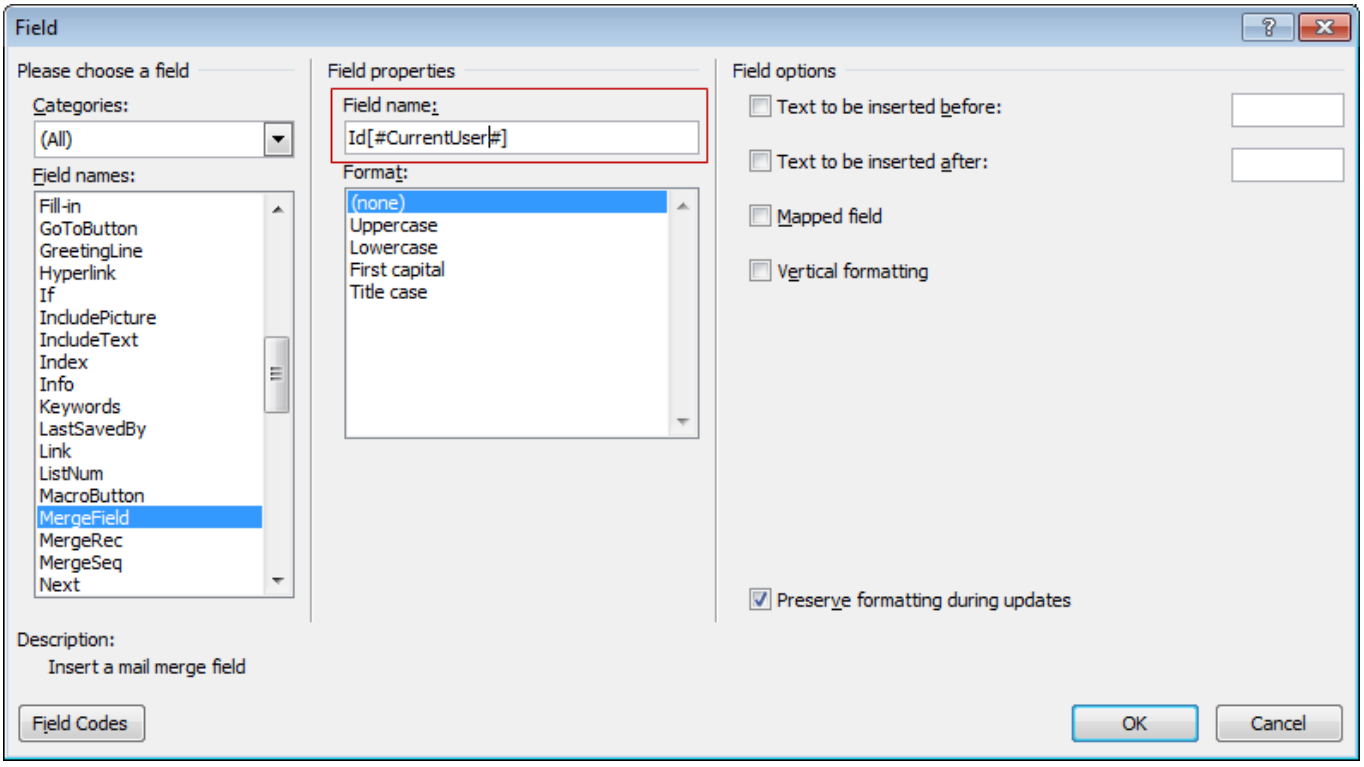
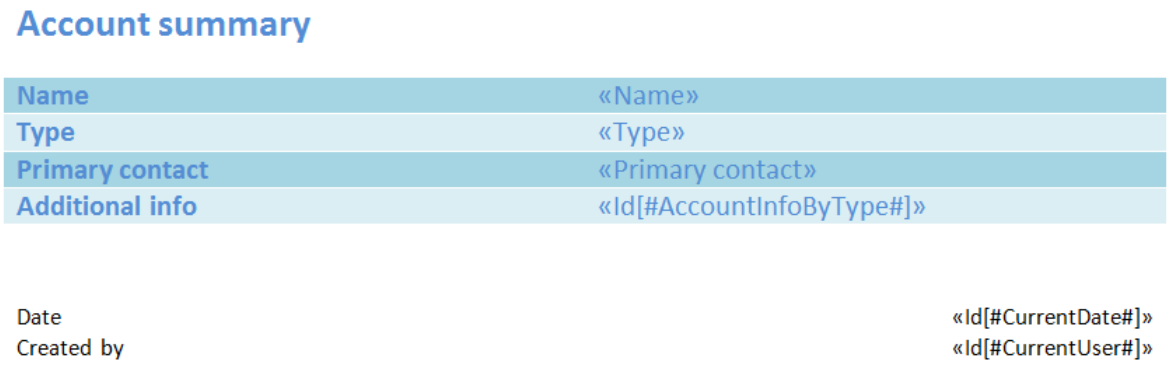


Fig. 7. Adding the CurrentUser tag for receiving the current user



As a result, the template will look as shown in Fig. 8.

Fig. 8. A printable template.

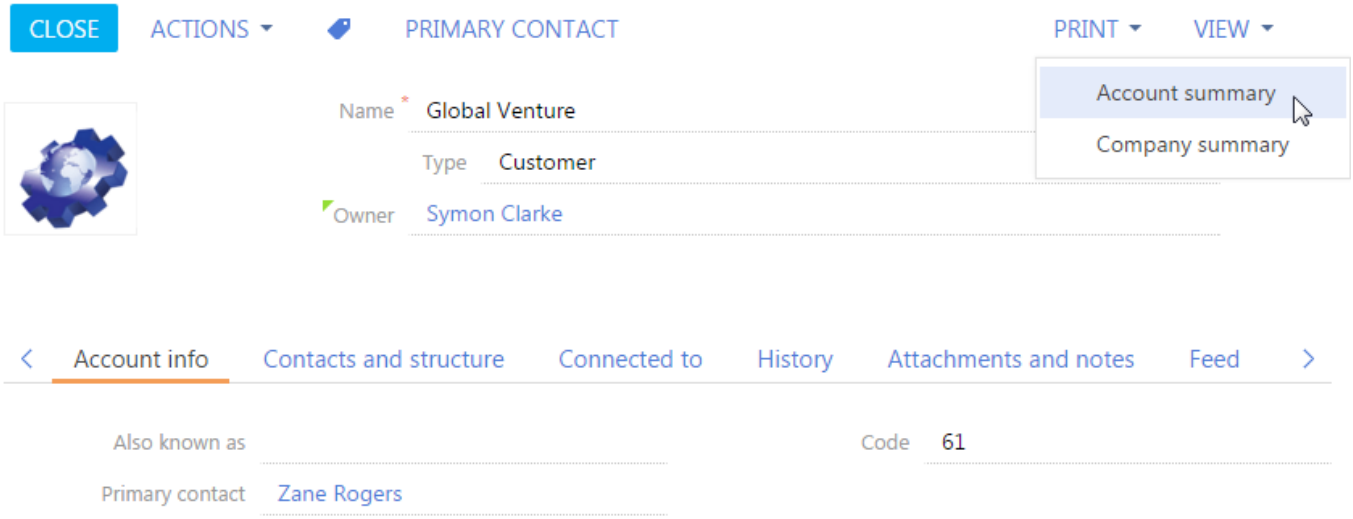


5. Saving the template and downloading it into the printable

Downloading the template into a printable is covered in the "Setting up MS Word printables in bpm'online" article.

The new [Account summary] printable will be displayed in the [Print] button menu of the account edit page after you download the template (fig.9).

Fig. 9. The new printable in the [Accounts] section



Information in the printable will depend on the account type (fig.10, 11).

Fig. 10. Printable for the [Customer] type account

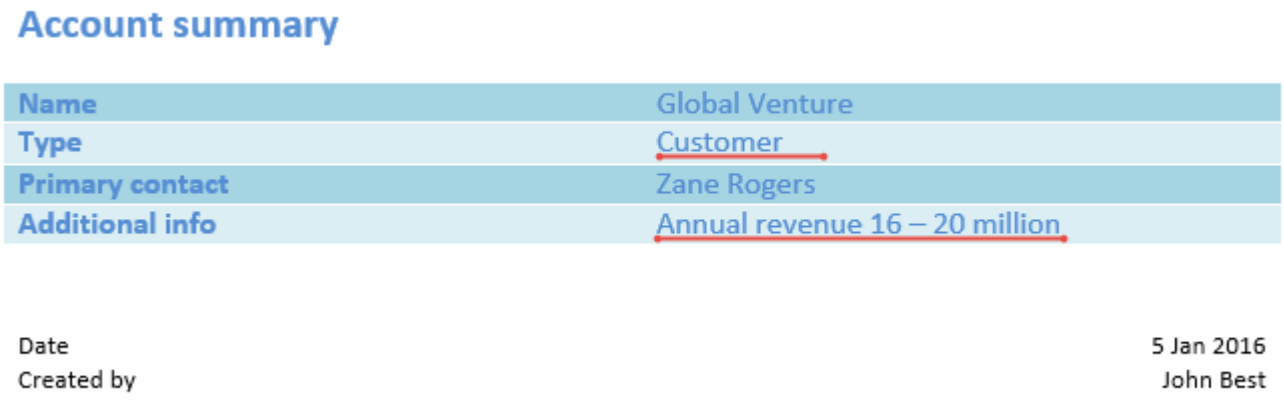
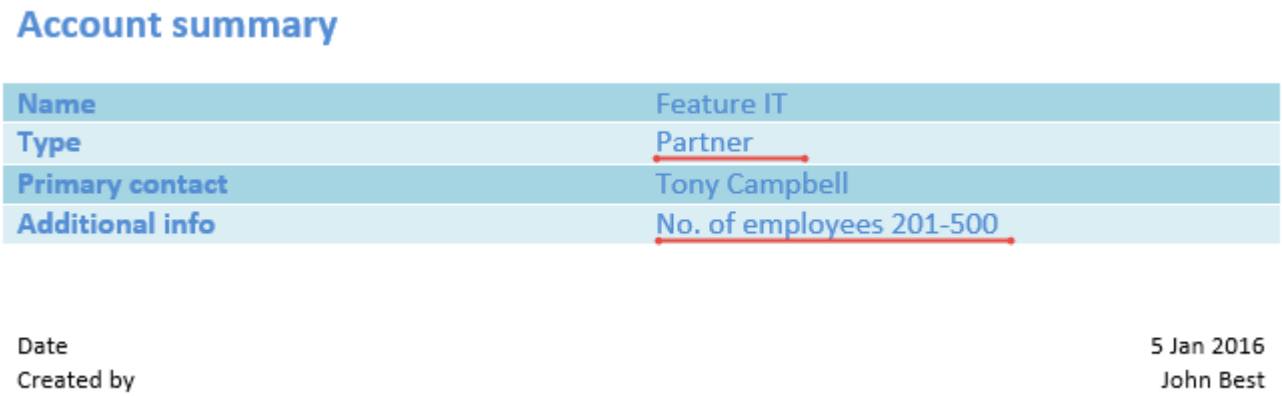


Fig. 11. Printable for the [Partner] type account



Working with data

Contents

- **CRUD-operations in configuration**
- **CRUD-operations on server side**
- **Web-services in configuration**
- **Reading multilingual data with EntitySchemaQuery**
- **Views localization**
- **Working with the localized data via Entity**
- **Adding a multilingual terminator to an object schema**
- **Using the DBExecutor for working with the database**

CRUD-operations in configuration

Contents

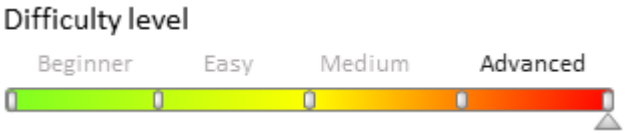
- **The use of EntitySchemaQuery implementation on client**

The use of EntitySchemaQuery implementation on client

Contents

- **Building of paths to columns relative to root schema**
- **Adding columns to a query**
- **Getting query result**
- **EntitySchemaQuery filters handling**

Building of paths to columns relative to root schema



Building of paths to columns relative toroot schema. Examples

The starting point of the EntitySchemaQuery building mechanism is *a root schema* and *feedback principle* (for more details see article **The use of EntitySchemaQuery for creation of queries in database**).

In order to add a column from a table to a query you must build the path to this column. There are different variants for adding columns to queries. Examples of the name formation of columns in each variant are shown below.

1) Root schema column

In this case, the column name is built as **[Column name in root schema].**

- Root schema: Contact
- Example: column with contact address
- Column name: Address
- Example of creation of the EntitySchemaQuery query that returns values of this column:

Example 1

```
// Let's create [EntitySchemaQuery] class instance with [Contact] root schema.
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
```

```
});
// Add [Address] column then add [Address] alias to it.
esq.addColumn("Address", "Address");
```

2) Schema column, lookup column of current schema refers to

The column name is built on the principle **[Lookup column name].[Schema column name, lookup refers to]**.

Country schema are jointed to City root schema by the JOIN operator (LEFT OUTER JOIN by default) in a resultant query. A joint condition (On condition of JOIN operator) is formed on the following principle:
[Name of joinable schema].[Id] = [Root schema name].[Name of column that refers to joinable schema + Id]

In common cases you can continue to build a feedback chain.

- Root schema: Contact
- Example: column with account name, column with name of main contact of account
- Column names: Account.Name, Account.PrimaryContact.Name
- Example of creation of EntitySchemaQuery query that returns values of these columns:

Example 2

```
//Let's create [EntitySchemaQuery] class instance with [Contact] root schema.
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
// Add [Account] lookup column.
// Then add [Name] column from [Account] schema,
// to which [Account] lookup column refers, and assign [AccountName] alias to it .
esq.addColumn("Account.Name", "AccountName");
// Add [Account] lookup column.
// Then add [PrimaryContact] lookup column from [Account] schema,
// to which [Account] lookup column refers.
// Add [Name] column from [Contact] schema,
// to which [PrimaryContact] lookup column refers and assign [PrimaryContactName]
alias to it.
esq.addColumn("Account.PrimaryContact.Name", "PrimaryContactName");
```

3) Schema column on random external key

Column name is built on the following principle **[Name of _ joinable_schema: Name of _column_for_linking of_joinable_schema:Name of_column_for_ linking of _current_schema]**.

If ID column is used as column for linking in current schema, it can be omitted, i.e. column name will have the following view:
[Name of_ joinable _ schema:Name of _ column_ for _linking of _joinable _schema].

In general, you can build the column names by the chains of reverse connections of any length.

- Example: column with name of the contact that has added city
- Column name: [Contact:Id:CreatedBy].Name
- Example of creation of EntitySchemaQuery of returning value of this column:

Example 3

```
// Let's create [EntitySchemaQuery] class instance with root schema [Contact].
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
// Add one more [Contact] schema to [Owner] column
// and select [Name] column from it. Assign [OwnerName] alias to it.
esq.addColumn("[Contact:Id:Owner].Name", "OwnerName");
// Let's joint [Contact] schema to [Acount] lookup column on [PrimaryContact] column
// and select [Name] column from it.
// Assign [PrimaryContact] alias to it.
esq.addColumn("Account.[Contact:Id:PrimaryContact].Name", "PrimaryContactName");
```

Adding columns to a query

Difficulty level

Terrasoft.EntitySchemaQuery query column is Terrassoft.EntityQueryColumn class instance. You can specify main characteristics of column instance in its properties: title, display value, checkboxes, sorting order and direction etc.

addColumn() method that returns instance of column, added to query, is designed for adding columns to queries. The column name relative to root schema is formed in addColumn() methods in accordance with rules, described in **Building of paths to columns relative to root schema**. This method has several variants that allow for adding columns with different parameters to a query (table 1).

Table 1. — Method of adding columns to query

Method		
addColumn*(column,[columnAlias])* Creates and adds Terrasoft.Entity.QueryColumn column instance to query column collection.		
column	String/Terrasoft.BaseQueryColumn	Is a column adding path (is specified relative torootSchema) or query column instance Terrasoft.BaseQueryColumn.
columnAlias	String (optional)	Column alias.
addAggregationSchemaColumn*(columnPath, aggregationType, [columnAlias], aggregationEvalType)* Creates and adds Terrasoft.FunctionQueryColumn functional column instance with set aggregation type (Terrasoft.FunctionType.AGGREGATION) to query column collection.		
columnPath	String	Is a column adding path (it is specified relative torootSchema).
aggregationType	Terrasoft.AggregationType	Is a type of used aggregation function.
columnAlias	String (optional)	Is a column alias.
aggregationEvalType	Terrasoft.AggregationEvalType	Is an application field of aggregation function.
Aggregation types (Terrasoft.AggregationType)		
AVG	Is an average value of all times.	
COUNT	Is a number of all items.	
MAX	Is a maximum value among all items.	
MIN	Is a minimum value among all items.	
NONE	Means that the type of aggregation function is not determined.	
SUM	Is a sum of the values of all items.	
Application field of aggregation function (Terrasoft.AggregationEvalType)		
NONE	Means that application field of aggregation function is not determined.	
ALL	Means that this function is applied to all items.	
DISTINCT	Means that this function is applied to unique values.	
addParameterColumn*(paramValue, paramDataType, [columnAlias])* It creates and adds Terrasoft.ParemeterQueryColumn parameter column instance to column correlation.		
paramValue	Mixed	Is a parameter value. The value should correspond to data type.

paramDataType	Terrasoft.DataValueType	Is a parameter data type.
columnAlias	String (optional)	Is a column alias.

addFunctionColumn*(columnPath, functionType, [columnAlias])*
It creates and adds Terrasoft.FunctionQueryColumn function column instance to column collection.

columnPath	String	Is a column adding path (it is specified relative torootSchema).
functionType	Terrasoft.FunctionType	Is a function type.
columnAlias	String (optional)	Is a column alias.

Function type (Terrasoft.FunctionType)

NONE	Means that functional expression type is not determined.
MACROS	Is a macro substitution.
AGGREGATION	Is an aggregation function.
DATE_PART	Is a date part.
LENGTH	Is a length of byte value.

addDatePartFunctionColumn*(columnPath, datePartType, [columnAlias])*
It creates and adds Terrasoft.FunctionQueryColumn function column instance with [Date Part] type (Terrasoft.FucntionType.DATE_PART) to query column collection.

columnPath	String	Is a column adding path (it is specified relative torootSchema).
datePartType	Terrasoft.DatePartType	Is a data part, used as a value.
columnAlias	String (optional)	Column alias.

Data part(Terrasoft.DatePartType)

NONE	Is a blank field.
DAY	Is a day.
WEEK	Is a week.
MONTH	Is a month.
YEAR	Is a year.
WEEK_DAY	Is a week day.
HOUR	Is an hour.
HOUR_MINUTE	Is a minute.

addMacrosColumn*(macrosType, [columnAlias])*
It creates and adds Terrasoft.FunctionQueryColumn function column instance with [Macros] type (Terrasoft.FunctionType.MACROS) that doesn't require parameterization (for example, current month, current user, primary column etc.) to column collection.

macrosType	Terrasoft.QueryMacrosType	Is a column macros type.
columnAlias	String (optional)	Is a column alias.

Macros column types (Terrasoft.QueryMacrosType)

NONE	Means that macros type is not determined.
CURRENT_USER	Means current user.
CURRENT_USER_CONTACT	Means current user contact.

YESTERDAY	Means yesterday.
TODAY	Means today.
TOMORROW	Means tomorrow.
PREVIOUS_WEEK	Means previous week.
CURRENT_WEEK	Means current week.
NEXT_WEEK	Means next week.
PREVIOUS_MONTH	Means previous month.
CURRENT_MONTH	Means current month.
NEXT_MONTH	Means next month.
PREVIOUS_QUARTER	Means previous quarter.
CURRENT_QUARTER	Means current quarter.
NEXT_QUARTER	Means next quarter.
PREVIOUS_HALF_YEAR	Means previous half year.
CURRENT_HALF_YEAR	Means current half year.
NEXT_HALF_YEAR	Means next half year.
PREVIOUS_YEAR	Means previous year.
CURRENT_YEAR	Means current year.
PREVIOUS_HOUR	Means previous hour.
CURRENT_HOUR	Means current hour.
NEXT_HOUR	Means next hour.
NEXT_YEAR	Means next year.
NEXT_N_DAYS	Means next N days. It requires parameterization.
PREVIOUS_N_DAYS	Means previous N days. It requires parameterization.
NEXT_N_HOURS	Means next N hours. It requires parameterization.
PREVIOUS_N_HOURS	Means previous N hours. It requires parameterization.
PRIMARY_COLUMN	Means primary column.
PRIMARY_DISPLAY_COLUMN	Means primary display column.
PRIMARY_IMAGE_COLUMN	Means primary image column.

addDatePeriodMacrosColumn*(macrosType, [macrosValue], [columnAlias])*
It creates and adds Terrasoft.FunctionQueryColumn function column instance with [Macros] type (Terrasoft.FucntionType.MACROS) to query column columns. The function adds column with macros type that requires parameterization. For example, next N days, the 3d quarter of the year etc.

macrosType	Terrasoft.QueryMacrosType	Is a macros column type.
macrosValue	Number/Date (optional)	Is an auxiliary variable for macros.
columnAlias	String (optional)	Is a column alias.

Examples of addition of columns to query

Example 1. — Adding query column from root schema to query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addColumn("DurationInMinutes", "ActivityDuration");
```

Example 2. — Adding aggregation column query with SUM aggregation type, applied to all table records, to query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addAggregationSchemaColumn("DurationInMinutes", Terrasoft.AggregationType.SUM,
"ActivitiesDuration", Terrasoft.AggregationEvalType.ALL);
```

Example 3. — Adding aggregation column query with COUNT aggregation type, applied to table unique records, to query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addAggregationSchemaColumn("DurationInMinutes", Terrasoft.AggregationType.COUNT,
"UniqueActivitiesCount", Terrasoft.AggregationEvalType.DISTINCT);
```

Example 4. — Adding parameter column with TEXT data type to query column type

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addParameterColumn("DurationInMinutes", Terrasoft.DataValueType.TEXT,
"DurationColumnName");
```

Example 5. — Adding function column with LENGTH function type (value size in bytes) to query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addFunctionColumn("Photo.Data", Terrasoft.FunctionType.LENGTH, "PhotoLength");
```

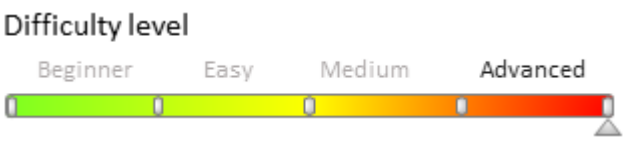
Example 6. — Adding function column with Date-Part function type (date part) to query column collection. Week day is used as a value

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addDatePartFunctionColumn("StartDate", Terrasoft.DatePartType.WEEK_DAY,
"StartDay");
```

Example 7. — Adding function column with MACROS type that don't require parameterization, i.e. PRIMARY_DISPLY-COLUMN (Primary Display Column), to query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addMacrosColumn(Terrasoft.QueryMacrosType.PRIMARY_DISPLAY_COLUMN,
"PrimaryDisplayColumnValue");
```

Getting query result



The EntitySchemaQuery query result is a bpm'online property collection. Each instance of a collection is a string of a data set, returnable by query. You can get query results in the following ways:

- Get a definite string of a data set by a primary key through calling the getEntity method (example 1).

- Get entire resultant data set by calling getEntityCollection method (example 2).

Example 1. — Getting a definite data set string

```
// Get [Id] of card object.
var recordId = this.get("Id");
// Create Terrasoft.EntitySchemaQuery class instance with [Contact] root schema.
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
// Add column with name of main contact of accounts that refers to given contact.
esq.addColumn("Account.PrimaryContact.Name", "PrimaryContactName");
// Get one record from selection on the basis of [Id] of card object and display it
// in an info window.
esq.getEntity(recordId, function(result) {
    if (!result.success) {
        // error processing/logging, for example
        this.showInformationDialog("Data query error");
        return;
    }
    this.showInformationDialog(result.entity.get("PrimaryContactName"));
}, this);
```

Example 2. — Getting entire data set

```
var message = "";
// Create Terrasoft.EntitySchemaQuery class instance with [Contact] root schema.
var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
// Add column with account name that refers to given account.
esq.addColumn("Account.Name", "AccountName");
// Add column with name of main contact account that refers to given contact.
esq.addColumn("Account.PrimaryContact.Name", "PrimaryContactName");
// Get entire record colelction and display it in an infor window.
esq.getEntityCollection(function (result) {
    if (!result.success) {
        // error processing/logging, for example
        this.showInformationDialog("Data query error");
        return;
    }
    result.collection.each(function (item) {
        message += "Account name: " + item.get("AccountName") +
        " - primary contact name: " + item.get("PrimaryContactName") + "\n";
    });
    this.showInformationDialog(message);
}, this);
```

NOTES

When retrieving the lookup columns, *this.get()* method returns object but not the database record identifier. To access identifier, use the "value" property, for example, *this.get('Account').value*.

Table 1. — Query result getting method

Method

getEntity*(primaryColumnValue, callback, scope)*
returns entity instance on set primary key [primaryColumnValue]. It calls [callback] function in [scope] context after data receipt.

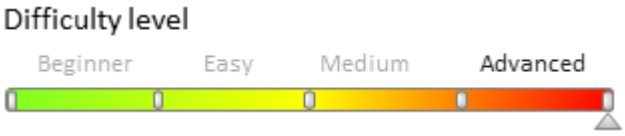
primaryColumnValue	String/Number	Is a primary key value.

callback	Function	Is a function, called upon receipt of server response.
scope	Object	Context where [callback] function will be called.

getEntityCollection*(callback, scope)*
returns collection of entity instances that represent current query results. It calls [callback] function in [scope] context after data receipt.

callback	Function	Is a function that will be called upon receipt of server response.
scope	Object	Is a context where [callback] function will be called.

EntitySchemaQuery filters handling



A filter is a set of conditions, applied to query data display. According to SQL terms, a filter is a separate predicate (condition) of the WHERE operator.

Creation and application of filters in EntitySchemaQuery

To create simple filter in EntitySchemaQuery use CreateFilter() method that returns created Terrasoft.CompareFilter filter object. In addition to simple filters, methods for special filter types are implemented in EntitySchemaQuery (Table 1).

Table 1. — EntitySchemaQuery methods for creation of filters

Filter creation method

createFilter*(comparisonType, leftColumnPath, rightColumnPath)*
Creates instance of Terrasoft.comparefilter class filter for comparing values of two columns.

comparisonType	Terrasoft.ComparisonType	Is a comparison operation type.
leftColumnPath	String	Is a path to verified column relative to root schema *rootSchema*.
rightColumnPath	String	Is a path to column filter relative to root schema *rootSchema*.

createInFilter*(leftExpression, rightExpressions)*
Creates In-filter instance.

leftExpression	Terrasoft.BaseExpression	Is an expression, verified in a filter.
rightExpressions	Terrasoft.BaseExpression[]	Is an array of expressions that will be compared with left expresion.

createBetweenFilter*(leftExpression, rightLessExpression, rightGreaterExpression)*
Creates Between-filter instance.

leftExpression	Terrasoft.BaseExpression	Is an expression, verified in a filter.
rightLessExpression	Terrasoft.BaseExpression	Is an initial expression of filtration range.
rightGreaterExpression	Terrasoft.BaseExpression	Is a final expression of filtration range.

createCompareFilter*(comparisonType, leftExpression, rightExpression)*
Creates Compare-filter instance.

comparisonType	Terrasoft.ComparisonType	Is a type of comparison operation.
leftExpression	Terrasoft.BaseExpression	Is an expression, verified in a filter.
rightExpression	Terrasoft.BaseExpression	Is a filtration expression.

createExistsFilter*(columnPath)*
Creates Exists-filter instance for comparison of [Exists on set condition] types and sets value of expression of column, located on set path, as verified value.

columnPath	String	Path to column, for the expression of which the filter is built.

createIsNotNullFilter*(leftExpression)*
Creates IsNull-filter instance.

leftExpression	Terrasoft.BaseExpression	Is an expression that is verified on IS NOT NULL condition.

createIsNullFilter*(leftExpression)*
Creates IsNull-filter instance.

leftExpression	Terrasoft.BaseExpression	Is an expression that is verified on IS NULL condition.

createNotExistsFilter*(columnPath)*
Creates Exists-filter instance for comparison [Out of set condition] and set expression of the column, located in set path, as verified value.

columnPath	String	Is a path to the verified column, for expression of which the filter is built.

createColumnFilterWithParameter*(comparisonType, columnPath, paramValue)*
Creates Compare-filter instance for comparison of the column with set value.

comparisonType	Terrasoft.ComparisonType	Is a type of comparison operation.
columnPath	String	Is a path to the verified column relative to root schema *rootSchema*.
paramValue	Mixed	Is a parameter value.

createColumnInFilterWithParameters*(columnPath, paramValues)*
Creates Inofilter instance for verification of coincidence of set column value with the value of one of parameters.

columnPath	String	Is a path to the verified column relative to root schema *rootSchema*.
paramValues	Array	Is a parameter value array.

createColumnBetweenFilterWithParameters*(columnPath, lessParamValue, greaterParamValue)*
Creates Between-filter instance that verifies whether the column is within set range.

columnPath	String	Is a path to the verified column relative to root schema *rootSchema*.
lessParamValue	Mixed	Initial value of the filter.
greaterParamValue	Mixed	Final value of the filter.

createColumnIsNotNullFilter*(columnPath)*
Creates IsNull-filter for verification of set column.

columnPath	String	Is a path to the verified column relative to root schema *rootSchema*.

***createColumnIsNullFilter**(columnPath)*
Creates IsNull-filter instance for verification of set column.

columnPath	String	Is a path to verified column relative to root schema *rootSchema*.

***createPrimaryDisplayColumnFilterWithParameter**(comparisonType, paramValue)*
Creates filter object for comparison of primary column for the purpose of displaying with parameter value.

comparisonType	Terrasoft.ComparisonType	Comparison type.
paramValue	Mixed	Parameter value.

The EntitySchemaQuery instance has a filter property that is a collection of filters of a given query. The filter property is the Terrasoft.FilterGroup class instance that, in its turn, is a collection of Terrasoft.BaseFilter items. To add a filter to a query, take the following actions:

- create filter instance for given query (createFilter method (), methods for creation of special type filters);
- add created filter instance or query filters collection (add() method of collection).

All filters added to the Filters collection are interconnected by the AND logical operation. With the LogicalOperation property of the filters collection, the user can specify a logical operation by which filters should be joined. The property takes the following values from Terrasoft.core.enums.LogicalOperatorType list:

- AND
- OR

The possibility for controlling filters, used in building of a resultant data set, is implemented in EntitySchemaQuery. Each filter collection item has the isEnabled property that determines whether this item takes part in building of resultant queries (true means that it takes part and false means that it doesn't take part). Similarly, the isEnabled property is also determined for the entire filter collection. Set this property to false to deactivate filtration for a query. The collection of query filters will remain unchanged. If a query filter collection is created initially, you can use different combinations for filtering queries in the future while not introducing changes directly into the collection.

Example of control of filters in query is shown below (example 1).

Example 1

```
// Creation of query instance with "Contact" root schema.
var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
esq.addColumn("Name");
esq.addColumn("Country.Name", "CountryName");

// Creation of the first filter instance.
var esqFirstFilter =
esq.createColumnFilterWithParameter(Terrasoft.ComparisonType.EQUAL, "Country.Name",
"Mexico");

// Creation of the second filter instance.
var esqSecondFilter =
esq.createColumnFilterWithParameter(Terrasoft.ComparisonType.EQUAL, "Country.Name",
"USA");

// Filters will be updated by OR logical operator in query filters collection.
esq.filters.logicalOperation = Terrasoft.LogicalOperatorType.OR;

// Adding created filters to collection.
esq.filters.add("esqFirstFilter", esqFirstFilter);
esq.filters.add("esqSecondFilter", esqSecondFilter);

// This collection will include objects, i.e. query results, filtered by two filters.
```

```
esq.getEntityCollection(function (result) {
    if (result.success) {
        result.collection.each(function (item) {
            // Processing element collection.
        });
    }
}, this);

// It is indicated that the second filter will be used in building of resultant
query.
// This filter is not deleted from query filters collection.
esqSecondFilter.isEnabled = false;

// This collection will include objects, i.e. query results, filtered only by the
first filter.
esq.getEntityCollection(function (result) {
    if (result.success) {
        result.collection.each(function (item) {
            // Processing of collection items.
        });
    }
}, this);
```

Column paths are built in the EntitySchemaQuery filters in accordance with common rules for building paths to columns relative to root schema (described in article **Building of paths to columns relative to root schema**).

Examples of the use of other methods for creating filters are represented below.

Example 2

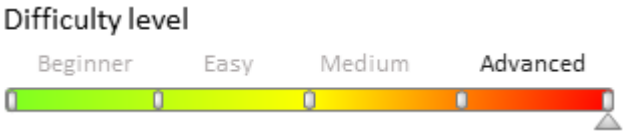
```
// Creation of query instance with "Contact" root schema.
var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
esq.addColumn("Name");
esq.addColumn("Country.Name", "CountryName");

// Select all contacts where country is not specified.
var esqFirstFilter = esq.createColumnIsNullFilter("Country");

// Select all contacts, date of birth of which fall at the period from 1.01.1970 to
1.01.1980.
var dateFrom = new Date(1970, 0, 1, 0, 0, 0, 0);
var dateTo = new Date(1980, 0, 1, 0, 0, 0, 0);
var esqSecondFilter = esq.createColumnBetweenFilterWithParameters("BirthDate",
dateFrom, dateTo);

// Add created filters to query collection.
esq.filters.add("esqFirstFilter", esqFirstFilter);
esq.filters.add("esqSecondFilter", esqSecondFilter);

// This collection will include objects, i.e. query results, filtered by two filters.
esq.getEntityCollection(function (result) {
    if (result.success) {
        result.collection.each(function (item) {
            // Processing of collection items.
        });
    }
}, this);
```

CRUD-operations on server side

Contents

- **Composing add data queries**
- **The use of EntitySchemaQuery for creation of queries in database**
- **Composing modify data queries**
- **Composing delete data queries**

Composing add data queries



The Insert class is used to add data to the bpm'online database.

The Insert class constructor uses the following objects as parameters:

- User connection (*Insert (UserConnection)*).
- Other object *Insert (Insert (Insert))*. This will create a copy of an Insert request included in the parameter.

Here is a number of examples.

Examples of simple requests

Example 1.

C#

```
var insert = new Insert(userConnection).Into("Contact")
        .Set("Name", Column.Parameter("ParameterNameValue"))
        .Set("Address", Column.Parameter("ParameterAddressValue"));
```

MS SQL

```
INSERT INTO [dbo].[Contact] ([Name], [Address]) VALUES (@P1, @P2)
```

Example 2

C#

```
var insert = new Insert(userConnection).Into("Contact")
        .Set("Name", Column.Const("NameValue"))
        .Set("Address", Column.Const("AddressValue"));
```

MS SQL

```
INSERT INTO [dbo].[Contact] ([Name], [Address]) VALUES ('NameValue', 'AddressValue')
```

Example 3

C#

```
var insert = new Insert(userConnection).Into("Contact")
        .Set("Name", Func.IsNull(Column.Parameter(string.Empty),
Column.Parameter("ParameterValue")));
```

MS SQL

```
INSERT INTO [dbo].[Contact] ([Name]) VALUES (ISNULL(@P1, @P2))
```

Examples of requests with conditions

Example 1

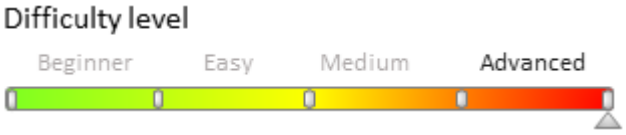
C#

```
var insert = new Insert(userConnection).Into("City")
        .Set("CreatedById",
             new Select(userConnection).Top(1)
                 .Column("Id")
                 .From("Contact")
                 .Where("Name").IsEqual(Column.Parameter("Supervisor")))
        .Set("ModifiedById",
             new Select(userConnection).Top(1)
                 .Column("Id")
                 .From("Contact")
                 .Where("Name").IsEqual(Column.Parameter("User1")));
```

MS SQL

```
INSERT INTO [dbo].[City] ([CreatedById], [ModifiedById])
VALUES((SELECT TOP 1 [Id]
FROM [dbo].[Contact]
WHERE [Name] = @P1),
(SELECT TOP 1 [Id]
FROM [dbo].[Contact]
WHERE [Name] = @P2))
```

The use of EntitySchemaQuery for creation of queries in database



General information

Purpose of the EntitySchemaQuery. Difference between the "Select" and "EntitySchemaQuery"

EntitySchemaQuery – is a high level class whose purpose is to build data select queries to database. To execute the database queries in bpm'online the *Select* class is also used. There is a number of principal differences between the *Select* and *EntitySchemaQuery* classes.

The *Select* class is the standard SQL instruction SELECT, which enables selecting one or more lines or columns from one or more tables in the database. The query is built "as is": columns are added to the query, the data sources are specified, filters and conditions are applied. The results of the query execution are returned as an *IDataReader* instance. Below is the example of using the *Select* class for retrieving a set of contacts (Example 1).

Using directives should be added to the project for successful compilation of parameters below:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Data;
using Terrasoft.Common;
```

```
using Terrasoft.Core;
using Terrasoft.Core.DB;
using Terrasoft.Core.Entities;
```

Example 1

```
// Creation of query instance, adding of columns and data source to query.
Select selectQuery = new Select(UserConnection)
                    .Column("Id")
                    .Column("Name")
                    .From("Contact");
// Execution of query to database and receipt of resulting data set.
using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection())
{
    using (IDataReader reader = selectQuery.ExecuteReader(dbExecutor))
    {
        while (reader.Read())
        {
            // Processing of query results.
        }
    }
}
```

EntitySchemaQuery class is a functionality that is similar to *Select* and extended with additional functions on access right control and bpm'online repository (cache) handling. *EntitySchemaQuery* is a functionally extended add-in of *Select* class. All additional properties and parameters of *EntitySchemaQuery* are projected onto *Select* instance that returns resultant data array.

GetSelectQuery() method returns *Select* instance, associated with definite query *EntitySchemaQuery* (Example 2).

Example 2

```
// Creation of query example EntitySchemaQuery.
EntitySchemaQuery esq = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"SomeSchema");
esq.AddColumn("SomeColumn");

// Receipt of Select instance, associated with created query EntitySchemaQuery.
Select selectEsq = esq.GetSelectQuery(UserConnection);
```

Result of execution of query *EntitySchemaQuery* is a collection of bpm'online entities. i.e. collection of instances of the *Entity* class (*EntityCollection* instance). Each *Entity* instance in a collection is a string of data array, returned by query. You can get query results in the following way (example 3):

- Get entire resultant data array by calling method *GetEntityCollection*.
- Get definite string of data array for set primary key by calling *GetEntity*.

Example 3

```
// Creation of query in City schema, adding of Name column to query.
var esqResult = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");
esqResult.AddColumn("Name");

// Execution of query to database and getting of all resultant data collection.
var entities = esqResult.GetEntityCollection(UserConnection);

// Execution of query to database and getting object with set identifier.
var entity = esqResult.GetEntity(UserConnection, new Guid("100B6B13-E8BB-DF11-B00F-
001D60E938C6"));
```

Main features of EntitySchemaQuery are as follows:

1) Access right support

Data extraction query of *EntitySchemaQuery* is built in a way that considers rights of current user. In other words, only data that are accessed by current user in accordance with its rights will get into resultant array. This is ensured through application of additional filters (conditions) upon formation of resultant query to database. You can adjust conditions of application of rights to linked tables, being available in query (that joint query by JOIN sentence), additionally for *EntitySchemaQuery*. These conditions are determined by the value of *JoinRightState* property of instance of *EntitySchemaQuery*. Conditions for application of rights to linked query tables are described in details below.

2) Caching mechanism

Mechanism for handling of repository (bpm'online cache or random repository, determined by the user) is implemented in *EntitySchemaQuery*). Cache handling optimizes operation effectiveness through access to cached query results without an additional query to the database. Upon execution of query EntitySchemaQuery the data, received from server database, are placed in cache that is determined by *Cache* property with key that is set by *CacheItemName*. Bpm'online cache of session level (data are available only in session of current user) with local data storage is the cache of *EntitySchemaQuery* by default. In general, a random repository, implementing *ICacheStore* can be used as query cache. Example of handling of query cache *EntitySchemaQuery* is represented below (Example 4).

Example 4

```
// Creation of query to City schema, adding Name column to query.
var esqResult = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");
esqResult.AddColumn("Name");

// Determination of the key, under which query execution results will be stored in
cache.
// BPMonlin cache of session level with local data cashing ( since Cache property of
the object is not determined)
// is used as cache.
esqResult.CacheItemName = "EsqResultItem";

// Collection, to which query execution results will be added.
var esqCityNames = new Collection<string>();

//Collection, to which cached query execution results will be added.
var cachedEsqCityNames = new Collection<string>();

// Execution of cache to database and getting resultant collections of objects.
// Query results will be placed in cache after completion of this operation.
var entities = esqResult.GetEntityCollection(UserConnection);

//Processing of query execution results and filling of esqCityNames collection.
foreach (var entity in entities)
{
    esqCityNames.Add(entity.GetTypedColumnValue<string>("Name"));
}

// Getting of link to esqResult query cache on the basis of CacheItemName key in the
form data table in memory.
var esqCacheStore = esqResult.Cache[esqResult.CacheItemName] as DataTable;

// Filling of CachedEsqCityNames collection with values from query cache.
if (esqCacheStore != null)
{
    foreach (DataRow row in esqCacheStore.Rows)
    {
        cachedEsqCityNames.Add(row[0].ToString());
    }
}
```

3) Additional query settings

You can set additional settings that determine parameters for page-by-page output of query execution results and also parameters of building of hierarchical query for *EntitySchemaQuery*. *EntitySchemaQueryOptions* is designed for these purposes. Properties of this class determine the following:

- *HierarchicalColumnName* – column name, used for building of hierarchical query;
- *HierarchicalColumnValue* – initial value of hierarchical column, on the basis of which hierarchy will be built;
- *HierarchicalMaxDepth* – maximum nesting level of hierarchical query;
- *PageableConditionValues* – value of page-by-page output conditions;
- *PageableDirection* – direction of page-by-page output conditions;
- *PageableRowCount* – umber of page records of resultant data array, returned by query.

One and the same *EntitySchemaQueryOptions* instance can be used for getting of different query execution results by transferring it to *GetEntityCollection()* of corresponding query (example 5) as parameter.

Example 5

```
// Creation of query instance with City root schema.
var esqCities = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");
esqCities.AddColumn("Name");

// Creation of query with Country root schema.
var esqCountries = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"Country");
esqCountries.AddColumn("Name");

// Creation of setting instance for returning the first 5 settings through query.
var esqOptions = new EntitySchemaQueryOptions()
{
    PageableDirection = PageableSelectDirection.First,
    PageableRowCount = 5,
    PageableConditionValues = new Dictionary<string, object>()
};

// Getting Cities collection that will contain the first 5 cities of resultant data
array.
var cities = esqCities.GetEntityCollection(UserConnection, esqOptions);

// Getting of countries collection that will contain the first 5 countries o f
resultant data array.
var countries = esqCountries.GetEntityCollection(UserConnection, esqOptions);
```

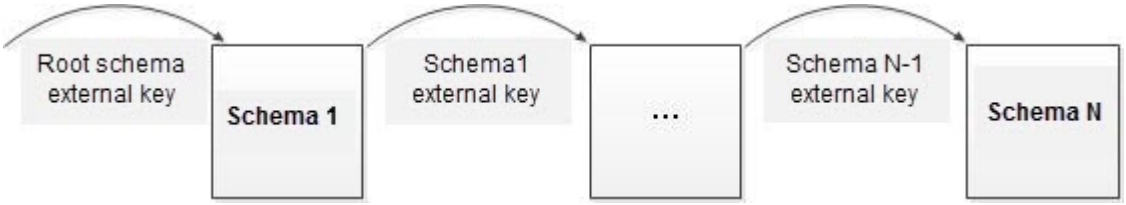
Definition of root schema. Creation of paths to columns against root schema. Examples.

The starting point of mechanisms for creating of *EntitySchemaQuery* is a *root schema*.

Root schema is a schema (table in database), whose paths are created in all columns in query, including columns of added tables.

Upon creation of paths, the feedback principle is applied to columns. The name of a random column, added to a query, can be built in the form of a chain of interconnected links. Each link represents the "context" of definite schema that is connected to a previous one on an external key (figure 1).

Figure 1. Interconnections of schemas on keys



In a common case, the format for building of a user column name built from N schema can be represented in the following form:

[Context of schema 1].[...].[Context of schema N].[Name of _ column]

In order to add column from a random table to a query on feedbacks, build the path to this column correctly. Different variants for adding columns to query and examples of name formation for column in each variant are described below. A city schema is used as a root schema for all examples, listed below.

1) Root schema column

In this case, the column name is built as **[Column name in root schema].**

- Example: column with city name
- Column name: Name
- *EntitySchemaQuery* creation example that returns values of this column:

```
// Creation of EntitySchemaQuery query instance with "City" root schema.
var esqQuery = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

// Adding to the column query with city name.
esqQuery.AddColumn("Name");

// Getting text if resultant sql-query.
string esqSqlText = esqQuery.GetSelectQuery(UserConnection).GetSqlText();
```

- Resultant sql-query (MS SQL):

```
SELECT
[City].[Name] [Name]
FROM
[dbo].[City] [City]
```

2) Schema column, to which lookup column of current schema refers

The column name is built on the basis of **[Lookup column name].[Name of column of schema, to which lookup refers]**.

Joining the operator (Left Outer Join by default) will join the Country schema in a resulting query to the City root schema. Joining a condition (On condition of JOIN operator) is formed by the following principle :

[Name of joinable schema].[Id] = [Name of root schema].[Name of column that refers to joinable schema +Id]

- Example: column with name of the country, to which the city belongs]
- Column name: Country.Name
- *EntitySchemaQuery* creation example that returns values of this column:

```
//Creation of EntitySchemaQuery query instance with "City" root schema.
var esqQuery = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

// Adding of column with country name, to which the city belongs, to query.
esqQuery.AddColumn("Country.Name");

// Getting text of resultant sql-query.
string esqSqlText = esqQuery.GetSelectQuery(UserConnection).GetSqlText();
```

- Resultant sql-query (MS SQL):

```
SELECT
[Country].[Name] [Country.Name]
FROM
[dbo].[City] [City]
LEFT OUTER JOIN [dbo].[Country] [Country] ON ([Country].[Id] = [City].[CountryId])
```

In a common case, you can continue to build the feedback chain.

- Example: contact name that has added country of definite city
- Column name: Country.CreatedBy.Name
- *EntitySchemaQuery* creation example that returns values of this column:

```
// Creation of the example of EntitySchemaQuery query with "City" root schema.
var esqQuery = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

// Adding column with contact name, which has added country of definite city, to
query.
esqQuery.AddColumn("Country.CreatedBy.Name");

// Getting text of resultant sql-query.
string esqSqlText = esqQuery.GetSelectQuery(UserConnection).GetSqlText();
```

- Resultant sql-query (MS SQL):

```
SELECT
[CreatedBy].[Name] [CreatedBy.Name]
FROM
[dbo].[City] [City]
LEFT OUTER JOIN [dbo].[Country] [Country] ON ([Country].[Id] = [City].[CountryId])
LEFT OUTER JOIN [dbo].[Contact] [CreatedBy] ON ([CreatedBy].[Id] =
[Country].[CreatedById])
```

3) Schema column on random external key

Column name is built in accordance with the following principle **[Name_of_joinable_schema:Name_of_column_for_linking_of_joinable_schema:Name of _column_ for_connection_of_current_schema]**.

If the ID column is used as a linking column in the current schema, it can be omitted, i.e. the column name will have the following view:

[Name_of_joinable_schema:Name_of_column_for_linking_of_joinable_schema] .

- Example: column with the name of the contact that has an added city
- Column name: [Contact:Id:CreatedBy].Name
- Example of creation of *EntitySchemaQuery* of returning value of this column:

```
//Creation of instance of EntitySchemaQuery query with "City" root schema.
var esqQuery = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

// Adding column with name of the contact that has added city, to query.
esqQuery.AddColumn("[Contact:Id:CreatedBy].Name");

// Getting text of resultant sql-query.
string esqSqlText = esqQuery.GetSelectQuery(UserConnection).GetSqlText();
```

- Resultant sql-query (MS SQL):

```
SELECT
[Contact].[Name] [Contact.Name]
FROM
[dbo].[City] [City]
LEFT OUTER JOIN [dbo].[Contact] [Contact] ON ([Contact].[Id] = [City].[CreatedById])
```

In general, you can build a column name on the basis of feedback chains of random length. The example below shows an alternative variant of name building for a column with the name of the contact that has added a country of a definite city (see clause 2).

- Example: column with name of the contact that has added country of definite city
- Column name: Country.[Contact: Id:CreatedBy].Name
- Example for creation of *EntitySchemaQuery* that retuns values of this column:

```
// Creation of EntitySchemaQuery instance with "City" root schema.
var esqQuery = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

// Adding column with contact name, which has added country of definite city, to
query.
esqQuery.AddColumn("Country.[Contact:Id:CreatedBy].Name");

// Getting text of resultant sql-query.
string esqSqlText = esqQuery.GetSelectQuery(UserConnection).GetSqlText();
```

- Resultant sql-query (MS SQL):

```
SELECT
[Contact].[Name] [Contact.Name]
FROM
[dbo].[City] [City]
LEFT OUTER JOIN [dbo].[Country] [Country] ON ([Country].[Id] = [City].[CountryId])
LEFT OUTER JOIN [dbo].[Contact] [Contact] ON ([Contact].[Id] =
[Country].[CreatedById])
```

Adding columns to query

The column of the *EntitySchemaQuery* query is an instance of *EntitySchemaQueryColumn* class. You can indicate the main characteristics of a column instance in its properties: titles, view values, checkboxes, sorting order and position etc. A full list of properties and methods of *EntitySchemaQueryColumn* class is shown in the corresponding section of the SDK class description.

The *AddColumn()* method) that returns an instance of a column, added to a query, is designed for adding columns to a query. The column name related to the root schema is formed in the *AddColumn()* method in accordance with above-mentioned rules. This method has some overloads. This allows adding columns with different parameters (table 1) to a query.

Table 1. The list of overloads of the AddColumn() method

Overload	Description
EntitySchemaQuery.AddColumn(String,AggregationTypeStrict,EntitySchemaQuery)	It creates and adds a column into an object schema query in the form of a subquery that returns results of specified aggregating function on path to column schema relating to root schema.
EntitySchemaQuery.AddColumn(EntitySchemaQueryColumn)	Adds a passed column to a columns collection of an object schema query.
EntitySchemaQuery.AddColumn(String)	It creates and adds column into query to object schema on set path to column against root schema.

EntitySchemaQuery.AddColumn(EntitySchemaQueryFunction)	It creates and adds column into query to object schema on transferred function.
EntitySchemaQuery.AddColumn(Object,DataValueType)	It creates and adds column of "parameter" type with set value of definite parameters to query to object schema.
EntitySchemaQuery.AddColumn(EntitySchemaQuery)	It creates and adds transferred instance of subquery EntitySchemaQuery to object schema as query column.

Specific of use of jointed tables

JOIN types of jointed tables

If a column from a schema that differs from the root schema is added, this schema will be added to the query by the JOIN operator. In this case, LEFT OUTER join type is applied by default. *EntitySchemaQuery*indicates the type of joining of this schema to the query upon adding the column of a non-root schema. For this purpose you should indicate the column name in the following form:

[Special join symbol][Column name]

EntitySchemaQuery supports the following types of joins, to which symbols, shown in table 2, correspond.

Table 2. JOIN types of jointed schemas

Join types	Special symbol of join type	Example of column name
INNER JOIN	"="	"=Name"
LEFT OUTER JOIN	">"	">Name"
RIGHT OUTER JOIN	"<"	"<Name"
FULL OUTER JOIN	"<>"	"<>Name"
CROSS JOIN	"*"	"*Name"

Examples of adds to column query with use of different schema joining types are shown below.

Example 6

```
// Creation of instance of query with City root schema.
var esqResult = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

// Country schema with join type LEFT OUTER JOIN will be added to query.
esqResult.AddColumn("Country.Name");

// Country schema with join type INNER JOIN will be added to query.
esqResult.AddColumn("=Country.Name");

// Two schemas will be added to query:
// 1) Country schema with join type LEFT OUTER JOIN ;
// 2) Contact schema with join type RIGHT OUTER JOIN.
esqResult.AddColumn(">Country.<CreatedBy.Name");

// Text of resultant sql-query (MS SQL):

// SELECT
```

```
//             [Country].[Name] [Country.Name],
//             [Country1].[Name] [Country1.Name],
//             [CreatedBy].[Name] [CreatedBy.Name]
// FROM
//             [dbo].[City] [City]
//             LEFT OUTER JOIN [dbo].[Country] [Country] ON ([Country].[Id] = [City].
[CountryId])
//             INNER JOIN [dbo].[Country] [Country1] ON ([Country1].[Id] = [City].
[CountryId])
//             LEFT OUTER JOIN [dbo].[Country] [Country2] ON ([Country2].[Id] =
[City].[CountryId])
//             RIGHT OUTER JOIN [dbo].[Contact] [CreatedBy] ON ([CreatedBy].[Id] =
[Country2].[CreatedById])
```

Application of access rights to jointed schemas

If the root schema of a query is managed by records and there are joinable schemas in the query, access rights of a current user can be applied to these records. Table 3 shows all possible variants of the application of access rights to joinable schemas. These values correspond to enumeration members *Terrasoft.Core.DB.QueryJoinRightLevel*.

Table 3. Variants of application of access rights to joinable query schemas

Value of enumeration member	Procedure for application of access rights
EnabledAlways = 0	Always apply access rights.
EnabledForAdditionalColumns = 1	Transfer rights if columns from jointed schema that differ from primary columns (PrimaryColumn) and primary for view are used in query.
Disabled = 2	Do not apply access right.

The procedure of changing access rights is determined the value of the JoinRightState property of query. The default value of this property is determined by the system setting QueryJoinRightLevel that accepts values in accordance with table 3. If the value of this system setting is not set, its default value is taken to be equal to EnabledForAdditionalColumns.

EntitySchemaQuery filters handling

Notion of EntitySchemaQuery filter and its structure

A filter is a set of conditions, applied to views of a data query. According to SQL terms, a filter is a separate predicate (condition) of the WHERE operator.

Any filter, as a query condition, has its structure (figure 2).

Figure 2. Structure of EntitySchemQuery filter

```
Filter = {[AggregationType] {<LeftExpression> | <LeftExpressionColumnPath>}
<ComparisionType>
{{<RightExpression> | {<RightExpressionColumnPath>,...}} | {<Macros>, [MacrosValue]}}
}
```

Main components of *EntitySchemaQuery* filter:

- *AggregationType* is a type of aggregation function that is expressed in the left side of the condition. It is optional component of the filter. It is set by enumeration values *FilterAggregationType*.
- *LeftExpression* is an expression in the left part of the filter. It is set by type instance *EntitySchemaQueryExpression*.
- *LeftExpressionColumnPath* is a path to column that contains expression of the left part of the filter. It is set by the string value.
- *ComparisionType* is a type of comparison of expressions in the filter. It is assigned by enumeration value *FilterComparisonType*.
- *RightExpression* is an expression in the right part of the filter. It is set by type instance

EntitySchemaQueryExpression.

- *RightExpressionColumnPath* is a path to column that contains expression of the right part of the filter. It is set by the string value.
- *Macros* is a macros that returns expression for the right part of the filter. It is assigned enumeration value *EntitySchemaQueryMacrosType*.
- *MacrosValue* is a value that is assigned to *Macros* as parameter. It is an optional parameter. It is assigned values of different types depending on the type of called macros.

Creation and application of filters in EntitySchemaQuery

The *CreateFilter()* method that returns instance of type *EntitySchemaQueryFilter* is used for creation of simple filter (figure 2) in *EntitySchemaQuery* . Some reloads are implemented for this method in the *EntitySchemaQuery*. This allows creation of filters with different initial parameters. Full list of reloads of the *CreateFilter()* method with examples of their implementation is available in bpm'online SDK.

In spite of simple filters methods for creation of special type filters are implemented in *EntitySchemaQuery* (table 4).

Table 4. EntitySchemaQuery methods for creation of special type filters

Filter creation method	Description
CreateFilterWithParameters()	It created parameterized filter for record retrieval under definite conditions. It is overloaded method.
CreateIsNullFilter()	It creates type comparison filter [Is null in database].
CreateIsNotNullFilter()	It creates type comparison filter [Is not null in database].
CreateExistsFilter()	It creates type comparison filter [Exists on set condition].
CreateNotExistsFilter()	It creates type comparison filter [Does not exists on set condition].

EntitySchemaQuery has *Filters* property) that is represented by collection of filters of given query (instance of *EntitySchemaQueryFilterCollection* class that, in its turn, is a classical typified collection of *IEntitySchemaQueryFilterItem*) items. In order to add filter to query it is necessary to take the following actions:

- create instance of filter for given query (*CreateFilter()* methods, methods for creation of special type filters);
- add created filter instance to query filter collection (*Add()* method of collection).

All filters, added to *Filters* collection, are interconnected through AND logic operation by default. *LogicalOperation* property of *Filters* collection allows user to indicate logic operation, which should be used for joining filters. The property takes values from the list *LogicalOperationStrict*.

The possibility for control of filters, used in building of resultant data array, is implemented in *EntitySchemaQuery* queries. Each item of *Filters* collection has *IsEnabled* property that determines whether this item takes part in building of a resultant query (true means that it take part and false means that it doesn't take part). A similar property *IsEnabled* is also determined for the entire collection of *Filters*. Set this property to “false” to deactivate filtration for a query. In this case, a collection of query filters will remain unchanged. Therefore, if a query filter collection is created initially, you can use different combinations for filtering of this query in the future while not introducing changes directly into a collection. An example of controlling filters in a query is shown below (example 7).

Example 7

```
// Creation of query instance with "City" root schema.
var esqCities = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");
esqCities.AddColumn("Name");

// Creation of the first filter instance.
var esqFirstFilter = esqCities.CreateFilterWithParameters(FilterComparisonType.Equal,
"Name", "New York");
```

```
// Creation of the second filter instance.
var esqSecondFilter =
esqCities.CreateFilterWithParameters(FilterComparisonType.Equal, "Name", "Boston");

// Filters will be joined in filter collection by OR logic operator.
esqCities.Filters.LogicalOperation = LogicalOperationStrict.Or;

// Adding created filters to query collection.
esqCities.Filters.Add(esqFirstFilter);
esqCities.Filters.Add(esqSecondFilter);

// Objects, i.e. query results, filtered by two filters, will be included into this
collection.
var entities = esqCities.GetEntityCollection(UserConnection);

// It is specified for the second filter that it will be used in building of
resultant query.
// In this case the filter is not deleted for query filter collection.
esqSecondFilter.IsEnabled = false;

// It updates Select instance, associated with query in accordance with actual set of
filters.
esqCities.ResetSelectQuery();

// The objects, namely query results , filtered only by the first filter, will be
included into this collection.
var entities1 = esqCities.GetEntityCollection(UserConnection);
```

Column paths in *EntitySchemaQuery* filters are formed in accordance with common rules for building paths to columns against root schema (as described above).

The example below shows how to select activities results for a definite category from the *ActivityCategory* root schema based on feedback. For the purpose of demonstration of use cases of feedbacks for columns in the columns of filters, the activity identifier column, which is built on feedbacks through *ActivityCategoryResultEntry* schema, will be used for selection of a an activity, instead of the Id column of the root schema.

Example 8

```
// Creation of query instance with ActivityCategory root schema.
var esqResult = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"ActivityCategory");

// Adding the activity result column to the request on feedbacks.
esqResult.AddColumn("[ActivityCategoryResultEntry:ActivityCategory].ActivityResult");

// Determining the ID of the activity, for which the results will be selected.
var requiredActivityCategoryId = new Guid("42C74C49-58E6-DF11-971B-001D60E938C6");

// Creation of the filter example for selection of the results for definite activity
category.
var filter = esqResult.CreateFilterWithParameters(FilterComparisonType.Equal,
                    "
[ActivityCategoryResultEntry:ActivityCategory].ActivityCategory.Id",
                    requiredActivityCategoryId);

// Adding filter to the query collection filter.
esqResult.Filters.Add(filter);

// Text of resultant sql-query (MS SQL):

// SELECT
//        [ActivityCate1].[ActivityResultId] [ActivityCate1.ActivityResultId],
```

```
//         [ActivityResult].[Name] [ActivityResult.Name]
//     FROM
//         [dbo].[ActivityCategory] [ActivityCategory]
//         LEFT OUTER JOIN [dbo].[ActivityCategoryResultEntry] [ActivityCate1]
//                 ON ([ActivityCate1].[ActivityCategoryId] = [ActivityCategory].
[Id])
//         LEFT OUTER JOIN [dbo].[ActivityResult] [ActivityResult]
//                 ON ([ActivityResult].[Id] = [ActivityCate1].[ActivityResultId])
//     WHERE
//         EXISTS (
//                 SELECT
//                     [ActivityCategory1].[Id] [Id]
//                 FROM
//                     [dbo].[ActivityCategory] [ActivityCategory1]
//                 WHERE
//                     [ActivityCate1].[ActivityCategoryId] = [ActivityCategory1].[Id]
//                 AND [ActivityCategory1].[Id] = '{42C74C49-58E6-DF11-971B-
001D60E938C6}')
```

Composing modify data queries

Difficulty level

The *Update* class is used to update data in the bpm'online database.

The *Update* class constructor uses the following objects as parameters:

- User connection (*Update (UserConnection)*).
- User connection and schema name in which the data will be updated (*Update (UserConnection,String)*).
- User connection and query source object (*Update (UserConnection,ModifyQuerySource)*).
- Other object *Update* (*Update (Update)*). This will create a copy of an *Update* query included in the parameter.

The following are the examples of using the *Update* class method used to build queries of varying complexity. In each example an *Update* object is created and then an SQL query is provided that will be generated for each different DBMS (MS SQL).

Example 1

C#

```
var update = new Update(userConnection, "Contact")
        .Set("Name", Func.IsNull(Column.SourceColumn("Name"),
Column.Parameter("ParameterValue")));
```

MS SQL

```
UPDATE [dbo].[Contact] SET [Name] = ISNULL([Name], @P1)
```

Example 2

C#

```
var update = new Update(userConnection, "City")
        .Set("CreatedById",
             new Select(userConnection).Top(1)
                 .Column("Id")
                 .From("Contact")
```

```
                .Where("Name").IsEqual(Column.Parameter("Supervisor")))
        .Set("ModifiedById",
             new Select(userConnection).Top(1)
                .Column("Id")
                .From("Contact")
                .Where("Name").IsEqual(Column.Parameter("User1")));
```

MS SQL

```
UPDATE [dbo].[City]
SET [CreatedById] = (SELECT TOP 1 [Id]
FROM [dbo].[Contact]
WHERE [Name] = @P1),
[ModifiedById] = (SELECT TOP 1 [Id]
FROM [dbo].[Contact]
WHERE [Name] = @P2)
```

Composing delete data queries

Difficulty level

The *Delete* class is used to delete data from the bpm'online database.

The *Delete* class constructor uses the following objects as parameters:

- User connection (*Update (UserConnection)*).
- Other object *Delete* (*Delete (Delete)*). This will create a copy of an *Delete* query included in the parameter.

The following are the examples of using the *Delete* class method used to build queries of varying complexity. In each example a *Delete* object is created and then an SQL query is provided that will be generated for each different DBMS (MS SQL).

Example 1

C#

```
var delete = new Delete(userConnection)
        .From("City");
```

MS SQL

```
DELETE FROM [dbo].[City]
```

Example 2

C#

```
var delete = new Delete(userConnection)
        .From("City")
        .Where("Id").IsEqual(Column.Parameter("{210299B0-7A24-4C4D-9B6F-
658FEFEF2E13}"));
```

MS SQL

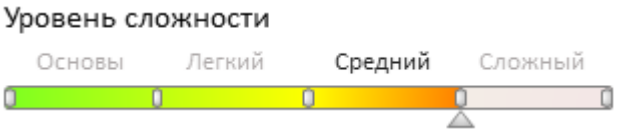
```
DELETE FROM [dbo].[City] WHERE [Id] = @P1
```

Web-services in configuration

Contents

- **How to create custom configuration service**
- **How to call configuration services with ServiceHelper**
- **Creating anonymous web service**
- **How to call configuration services using Postman**

How to create custom configuration service



Introduction

Bpm’online service model implements the base set of web services which you can use for integration of bpm’online with external applications and systems. The example of system services are: the EntityDataService.svc which provides the ability to exchange data with bpm'online over the OData protocol and the ProcessEngineService.svc which provides the **launch of bpm’online business process from external applications**.

Bpm’online enables to create custom web services that can implement specific integration tasks.

Configuration web service is a RESTful service implemented on the basis on WCF technology. The web service is available at following address:

```
[Application Address]/0/rest/[Custom Service Name]/[Custom Service Endpoint]?
[Optional Options]
```

Example:

```
http://mysite.bpmonline.com/0/rest/UsrCustomConfigurationService/GetContactIdByName?
Name=User1
```

> ⚠ ATTENTION
>
> Custom service is available after user authentication via the AuthService.svc (see “**The AuthService.svc authentication service**”).

To create custom web service in the configuration:

1. Create a schema of the [Source code] type in the development package.
2. Create a class of the service in the schema source code. Use the namespace in the *Terrasoft.Configuration* or any namespace embedded in it. Mark the class with the [ServiceContract] and [AspNetCompatibilityRequirement] attributes with necessary parameters.
3. Add the implementation of methods corresponding to the service endpoints in the class. Each method off the service should be marked with the [OperationContract] и [WebInvoke] attributes with necessary parameters. If you need to send the data of complex type (object instances, collections, arrays, etc.) you can implement additional classes which instances will receive and return the method of your service. Each class of that type should be marked with the [DataContract] attribute and the fields of the class should be marked with the [DataMember] attribute.
4. Perform publication of the source code schema.

After publication of the schema, the created web service will be available for call from the source code of the configuration schemas and from the external applications (see “ **How to call configuration services with ServiceHelper** ”).

Case description

Create custom configuration service that returns Id of the contact of the given name. If there are several contacts found, then return the Id of only the first contact found. If the contact is not found, the service should return an empty string.

Source code

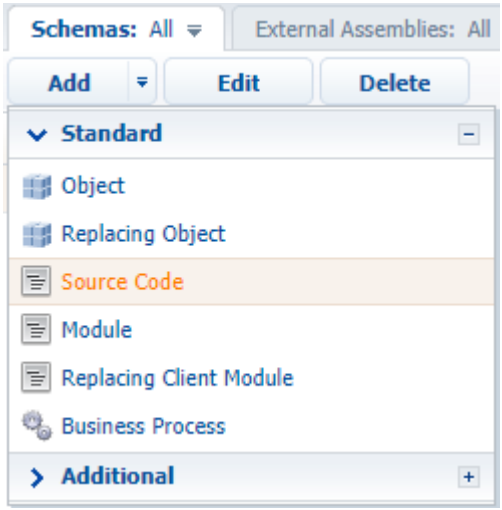
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Creating a [Source code] schema

Perform the [Add] – [Source code] action on the [Schemas] tab of the [Configuration] section.

Fig. 1. Adding the [Source Code] schema



Sett following properties for the schema:

- [Name] – UsrCustomConfigurationService.
- [Title] – UsrCustomConfigurationService.

2. Create class of the service

On the [Source code] tab:

- The namespace nested in the *Terrasoft.Configuration*. The name can be random, for example, the *UsrCustomConfigurationService*.
- Namespaces whose data types will be used in your class. For this use the *using* directive. A complete list of namespaces is provided in the source code below.
- Class, for example the *UsrCustomConfigurationService* class inherited from the *Terrasoft.Nui.ServiceModel.WebService.BaseService*. Mark the class with the [ServiceContract] and [AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequirementsMode.Required)] attributes.

> ⚠ ATTENTION
>
> It is not necessary to inherit the class of the service from other classes. In this example, the class is inherited from the BaseService only to be able to access the *AppConection* property.

The source code with a class declaration is available below:

3. Implement the methods that match the service endpoints

To implement the endpoint of the return of the contact Id by its name, add the *GetContactIdByName(string Name)* public string to the class. The *Name* parameter should receive the name of the contact. After accessing to the database via the **EntitySchemaQuery** the method will return the Id of the first found contact (or empty string) casted to string.

Full source code with the implementation of the service:

```
namespace Terrasoft.Configuration.UsrCustomConfigurationService
{
    using System;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using Terrasoft.Core;
    using Terrasoft.Web.Common;
    using Terrasoft.Core.Entities;

    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class UsrCustomConfigurationService: BaseService
    {
        // Link to the UserConnection instance required to access the database.
        private SystemUserConnection _systemUserConnection;
        private SystemUserConnection SystemUserConnection {
            get {
                return _systemUserConnection ?? (_systemUserConnection =
(SystemUserConnection)AppConnection.SystemUserConnection);
            }
        }

        // A method that returns the contact's ID by its name.
        [OperationContract]
        [WebInvoke(Method = "GET", RequestFormat = WebMessageFormat.Json, BodyStyle =
WebMessageBodyStyle.Wrapped,
        ResponseFormat = WebMessageFormat.Json)]
        public string GetContactIdByName(string Name){
            // The default result.
            var result = "";
            // An EntitySchemaQuery instance that accesses the Contact table of the
database.
            var esq = new EntitySchemaQuery(SystemUserConnection.EntitySchemaManager,
"Contact");
            // Adding columns to the query.
            var colId = esq.AddColumn("Id");
            var colName = esq.AddColumn("Name");
            // Filter the query data.
            var esqFilter =
esq.CreateFilterWithParameters(FilterComparisonType.Equal, "Name", Name);
            esq.Filters.Add(esqFilter);
            // Get the result of the query.
            var entities = esq.GetEntityCollection(SystemUserConnection);
            // If the data is received.
            if (entities.Count > 0)
            {
                // Return the value of the "Id" column of the first record of the
query result.
                result = entities[0].GetColumnValue(colId.Name).ToString();
                // You can also use this option:
                // result = entities [0]. GetTypedColumnValue <string> (colId.Name);
            }
            // Return the result.
```

```
            return result;
        }
    }
}
```

After making changes save and publish the schema.

As a result, the new configuration service *UsrCustomConfigurationService* will be available in the bpm'online. When the *GetContactIdByName* endpoint of this service is called, for example, out of web browser, the contact Id (Fig. 2) or the empty string ("") value (Fig. 3) will be returned.

> **Note**
>
> Pay attention to the format of the call result. In the server response, the object that contains property with the name combined from the name of the called method and the "Return" suffix, will be passed. Value of the object property contains the value of the contact Id (or an empty string) returned by the service.

Fig. 2. Request result The contact Id is found.

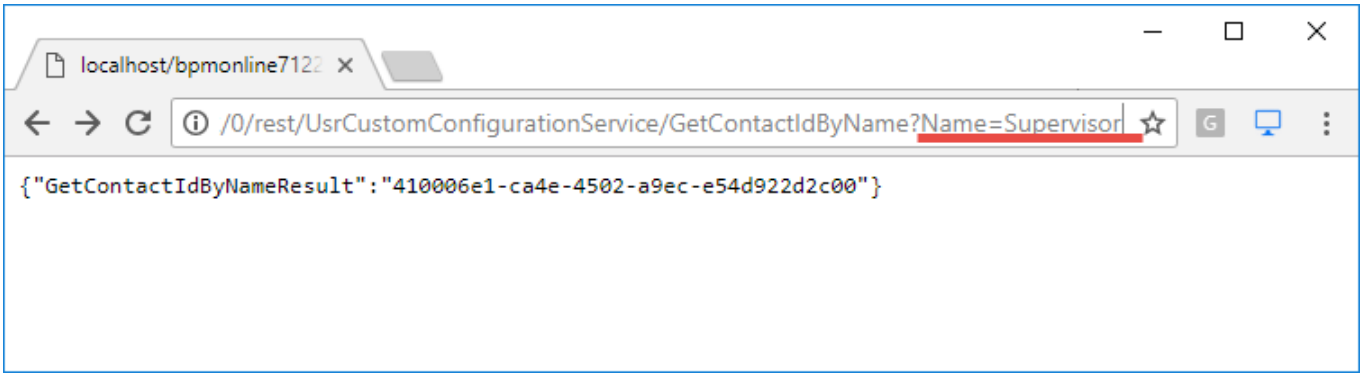
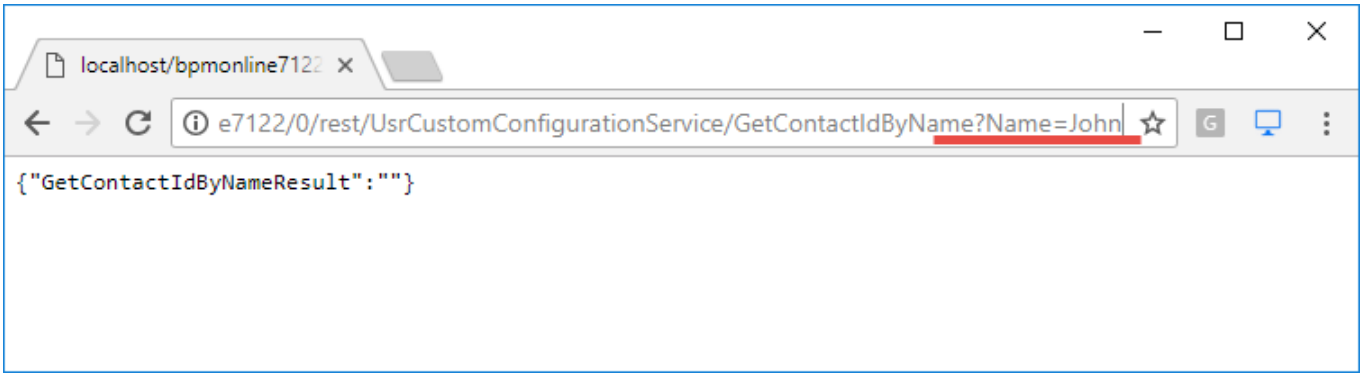
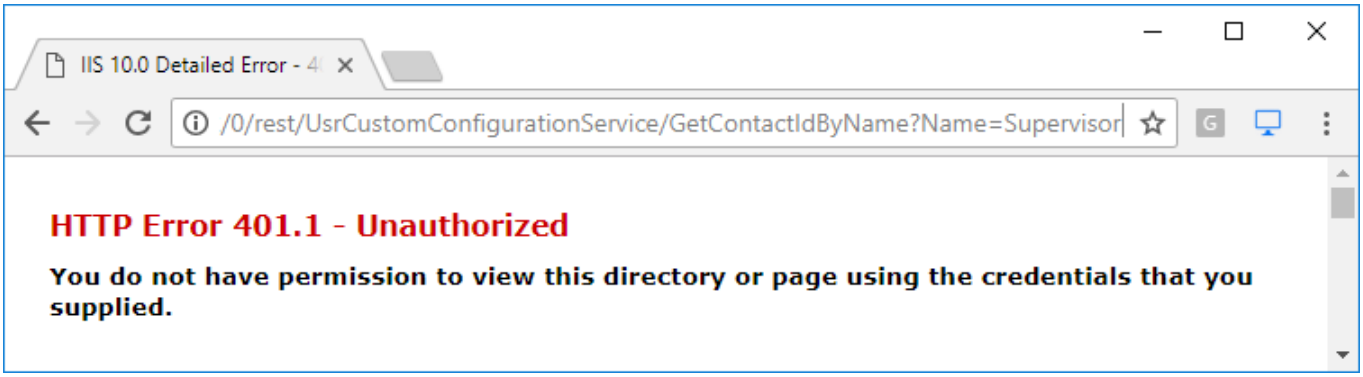


Fig. 3. Request result The contact Id is not found.

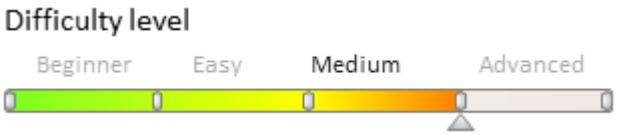


> **ATTENTION**
>
> If the service is called without logging to the application, the authorization error will be displayed (Fig. 4).

Fig. 4. Request result No authorization



How to call configuration services with ServiceHelper



Introduction

To call a configuration web service from the client JavaScript-code:

1. Add the *ServiceHelper* module as a dependency to the module of the page that was used for calling the service. This module is an interface for executing server queries via the *Terrasoft.AjaxProvider* query provider implemented in the client core.
2. Call the *callService(serviceName, serviceMethodName, callback, serviceData, scope)* method from the *ServiceHelper* module by passing the parameters as listed in table 1.

Table 1. The *callService()* method parameters

Parameter	Details
serviceName	Configuration service name. Corresponds to the name of the C# class that implements the service.
servcieMethodName	Name of the configuration service method being called. The method can accept incoming parameters and return resulting values.
callback(response)	The callback function that processes the service response. The function accepts the *response* object as a parameter. If the called service method returns any value, you can receive it on a client via the *response* object property. The name of the property that returns the method output-value is generated according to the following rule: [Service method name] + [Result] For example, if you call the *GetSomeValue()* method, the returned value will be contained in the *response.GetSomeValueResult* property.
serviceData	The object with the initialized incoming parameters for the service method.
scope	Execution context.

NOTE

There is an alternative way of calling the *callService(config)*method, where *config* is a configuration object with the following properties:

- *serviceName* – configuration service name
- *methodName* – name of the configuration service method being called
- *callback* – the callback function processing the service response
- *data* – the object with the initialized incoming parameters for the service method

- *scope* – execution context

ATTENTION

The *ServiceHelper* module only works with the POST requests. Therefore, the **configuration service** methods must be marked by the [WebInvoke] method with the *Method* = *"POST"* parameter.

Case description

Add a button calling the configuration web service to the new contact adding page. As a result, the information window of the page will display the result returned by the web service method.

Source code

You can download the package with case implementation using the following link.

Case implementation algorithm

1. Creating configuration service

The current case uses the web service from the "**How to create custom configuration service**" article. The "Method" parameter of the "WebInvoke" attribute is changed for POST in the service.

Service source code:

```
namespace Terrasoft.Configuration.UsrConfigurationService
{
    using System;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using Terrasoft.Core;
    using Terrasoft.Web.Common;
    using Terrasoft.Core.Entities;

    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class UsrConfigurationService: BaseService
    {
        // Link to the UserConnection instance needed for addressing the database.
        private SystemUserConnection _systemUserConnection;
        private SystemUserConnection SystemUserConnection {
            get {
                return _systemUserConnection ?? (_systemUserConnection =
(SystemUserConnection)AppConnection.SystemUserConnection);
            }
        }

        // Method returning the contact identifier by name.
        [OperationContract]
        [WebInvoke(Method = "POST", RequestFormat = WebMessageFormat.Json, BodyStyle
= WebMessageBodyStyle.Wrapped,
        ResponseFormat = WebMessageFormat.Json)]
        public string GetContactIdByName(string Name){
            // Default result.
            var result = "";
            // The EntitySchemaQuery instance, addressing the Contact database table.
            var esq = new EntitySchemaQuery(SystemUserConnection.EntitySchemaManager,
"Contact");
```

```
            // Adding columns to query.
            esq.AddColumn("Id");
            var colName = esq.AddColumn("Name");
            // Query data filtering.
            var esqFilter =
esq.CreateFilterWithParameters(FilterComparisonType.Equal, "Name", Name);
            esq.Filters.Add(esqFilter);
            // Receiving the query results.
            var entities = esq.GetEntityCollection(SystemUserConnection);
            // If data are received.
            if (entities.Count > 0)
            {
                // Return the "Id" column value of the first query result record.
                result = entities[0].GetColumnValue(colId.Name).ToString();
                // You can also use the folowing variant:
                // result = entities[0].GetTypedColumnValue<string>(colId.Name);
            }
            // Return result.
            return result;
        }
    }
}
```

2. Creating a replacing edit page

Create a replacing client module and specify [Display schema – Contact card] (*ContactPageV2*) as the parent object in the custom package (Fig. 1). Creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Add the *ServiceHelper* module as a dependency to declaring the edit page module.

2. Adding a button to the edit page

Adding a button to the edit page is described in the “**How to add a button to an edit page in the new record add mode**” and “**How to add the button on the edit page in the combined mode**” articles.

Add a localizable string with the button caption to the replacing module schema of the contact edit page, for example:

- [Name] – “GetServiceInfoButtonCaption”
- [Value] – “Call service”

3. Adding the button handler and calling the web service method

Use the *callService()* method of the *ServiceHelper* module to call the web service and pass the following values as parameters:

- name of the configuration service class (*UsrCustomConfigurationService*)
- name of the called service method (*GetContactIdByName*)
- the object with the initialized incoming parameters for the service method (*serviceData*)
- the callback function where processing of service results is executed
- execution context

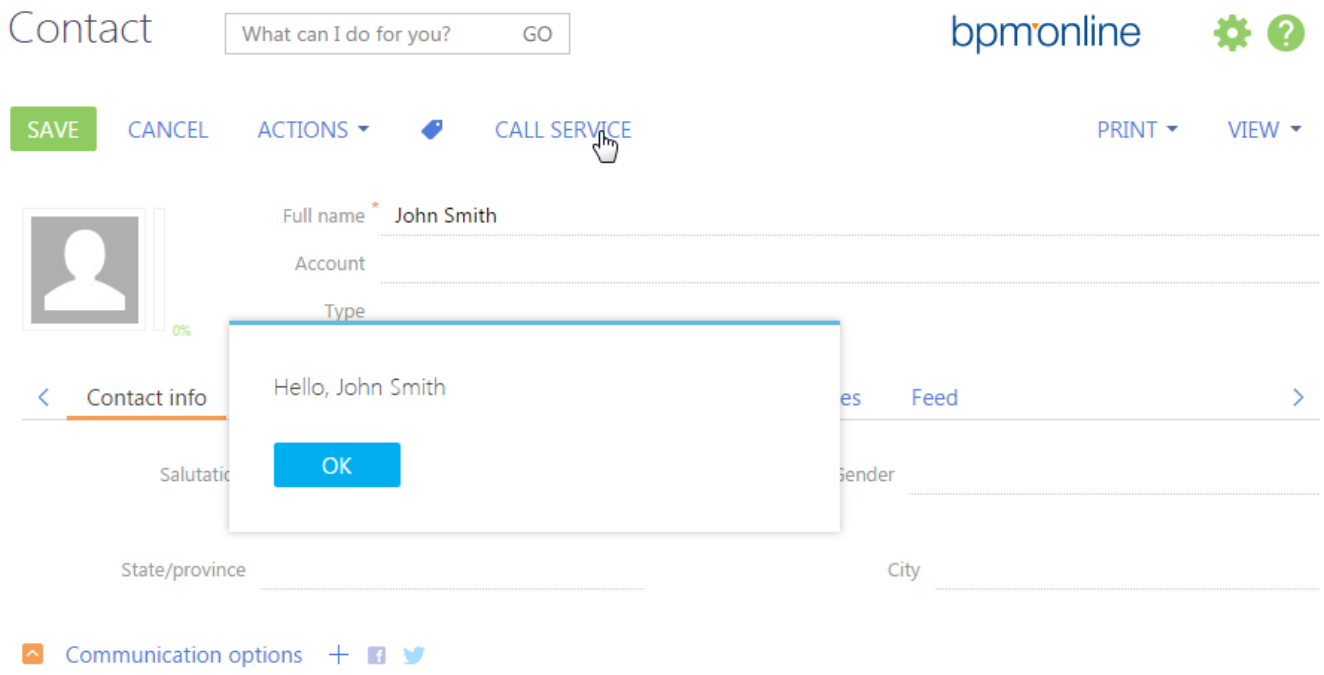
The source code of the edit page replacing module:

```
define("ContactPageV2", ["ServiceHelper"],
    function(ServiceHelper) {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Contact",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            // View model methods of the edit page.
            methods: {
                // Verifies if the [Full name] field is populated on the page.
```

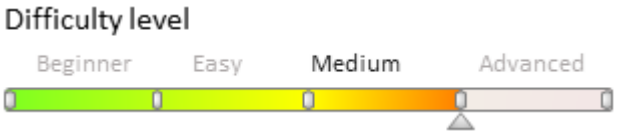
```
                isContactNameSet: function() {
                    return this.get("Name") ? true : false;
                },
                // Handler-method of clicking the button.
                onGetServiceInfoClick: function() {
                    var name = this.get("Name");
                    // Object initializing incoming parameters for the service
method.
                    var serviceData = {
                        // The property name corresponds to the incoming parameter
name of the service method.
                        Name: name
                    };
                    // Calling the web service and processing the results.
                    ServiceHelper.callService("UsrConfigurationService",
"GetContactIdByName",
                        function(response) {
                            var result = response.GetContactIdByNameResult;
                            this.showInformationDialog(result);
                        }, serviceData, this);
                }
            },
            diff: /**SCHEMA_DIFF*/[
                // Metadata for adding a custom button to a page.
                {
                    // Executing the operation of adding the element to page.
                    "operation": "insert",
                    // Name of the parent control element where the button is added.
                    "parentName": "LeftContainer",
                    // The button is added to the control element collection
                    // of the parent element (whose metaname is specified in
parentName).
                    "propertyName": "items",
                    // Name of the added button.
                    "name": "GetServiceInfoButton",
                    // Additional field property.
                    "values": {
                        // The added element type - button.
                        itemType: Terrasoft.ViewItemType.BUTTON,
                        // Binding the button caption to the localizable schema
string.
                        caption: {bindTo:
"Resources.Strings.GetServiceInfoButtonCaption"},
                        // Binding the handler-method of clicking the button.
                        click: {bindTo: "onGetServiceInfoClick"},
                        // Binding the "enabled" property ot the button.
                        enabled: {bindTo: "isContactNameSet"},
                        // Field location setup.
                        "layout": {"column": 1, "row": 6, "colSpan": 2, "rowSpan": 1}
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
    });
```

After you save the schema and update the application page, the [Call service] button will appear on the contact edit page. When you click the button, the configuration service method will be called (fig. 1).

Fig. 1. Case implementation result



Creating anonymous web service



Introduction

Bpm’online service model implements the base set of web services, which you can use for integration of bpm’online with external applications and systems. Examples of system services are: the EntityDataService.svc, which enables exchanging data with bpm'online via the OData protocol and the ProcessEngineService.svc, which provides the **launch of bpm’online business process from external applications**. These services are implemented based on the WCF technology and are managed at IIS level.

There also exist configuration web services in bpm’online that can be called from the custom part of the application. You can implement specific integration tasks via configuration web services. More information about creation of a custom configuration web service can be found in the “**How to create custom configuration service**” article.

Most of WCF-services require preliminary user authentication. However, there exist services that permit anonymous usage. An example of such a service is AuthService.svc.

> ⚠ ATTENTION
>
> Since configuration services are managed directly by the application and not by IIS, you cannot make them anonymous.

To create a WCF-service that would be accessible without user authentication:

1. Create a configuration web service (if needed).
2. Register the WCF-service.
3. Configure WCF-service for the http and https protocols.
4. Set up access to WCF-service for all users.

> ⚠ ATTENTION
>
> You need to change the application configuration files to set up anonymous web-service. When updating the application, all the configuration files are changed by the new ones. Thus, you need to set up the web service

again after the application update.

Case description

Create custom configuration service that returns the Id of a contact by the provided name. If there are several contacts found, it is only necessary to return the Id of the first contact. If the contact is not found, the service should return an empty string.

Note

You can use the service created based on the example covered in the "**How to create custom configuration service**" article as the configuration web service.

Case implementation algorithm

1. Creating configuration service

How to create the configuration service that would meet the case conditions is covered in the "**How to create custom configuration service**" article.

2. Registering the WCF-service.

Create the *UsrCustomConfigurationService.svc* file in the *..\Terasoft.WebApp\ServiceModel* catalog and add the following record into it:

```
<%@ ServiceHost Language="C#" Debug="true"
Service="Terrasoft.Configuration.UsrCustomConfigurationService.UsrCustomConfiguration
Service" %>
```

In the *Service* attribute specify the full name of the configuration service class. Read more about the @ServiceHost WCF-directive in Microsoft documentation.

3. Configuring WCF-service for the http and https protocols.

Add the following record to the *services.config* files located at *..\Terasoft.WebApp\ServiceModel\http* and *..\Terasoft.WebApp\ServiceModel\https* catalogs:

```
<services>
    ...
    <service
name="Terrasoft.Configuration.UsrCustomConfigurationService.UsrCustomConfigurationSer
vice">
        <endpoint name="UsrCustomConfigurationServiceEndPoint"
            address=""
            binding="webHttpBinding"
            behaviorConfiguration="RestServiceBehavior"
            bindingNamespace="http://Terrasoft.WebApp.ServiceModel"

contract="Terrasoft.Configuration.UsrCustomConfigurationService.UsrCustomConfiguratio
nService" />
    </service>
</services>
```

Configure the service here. The <services> element contains a list of configurations of all application services (the <service> nested elements). The *name* attribute contains the name of type (class or interface) implementing the service contract. The <endpoint> nested element requires address, binding and interface that define the service contract specified in the *name* attribute of the <service> element.

You can find detailed description of the service configuration elements in the documentation.

4. Setting up access to WCF-service for all users.

Perform the following changes in the ..*Terasoft.WebApp**Web.config* file:

- Add the <location> element defining the relative path and access rights to the service.
- In the <appSettings> element change the *value* value for the "AllowedLocations" key by adding the relative path to the service into it.

An example of changes in the ..*Terasoft.WebApp**Web.config* file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  ...
  <location path="ServiceModel/UsrCustomConfigurationService.svc">
    <system.web>
      <authorization>
        <allow users="*" />
      </authorization>
    </system.web>
  </location>
  ...
  <appSettings>
    ...
    <add key="AllowedLocations" value="[Предыдущие
значения];ServiceModel/UsrCustomConfigurationService.svc"  />
    ...
  </appSettings>
  ...
</configuration>
```

After reloading the application pool in IIS, the service will become available at:

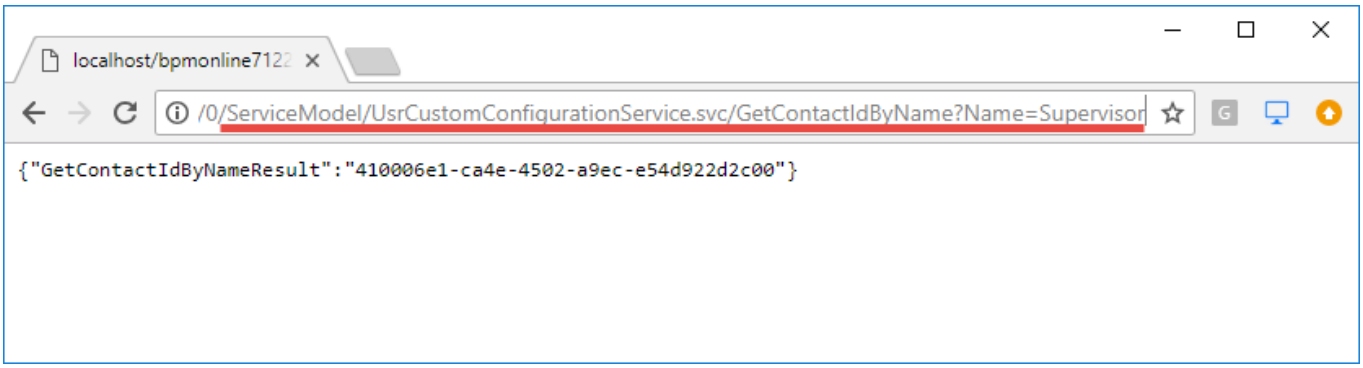
```
[Application Address]/0/ServiceModel/[Custom Service Name].svc/[Custom Service
Endpoint]?[Optional parameters]
```

For example:

```
http://mysite.bpmonline.com/0/ServiceModel/UsrCustomConfigurationService.svc/GetConta
ctIdByName?Name=Supervisor
```

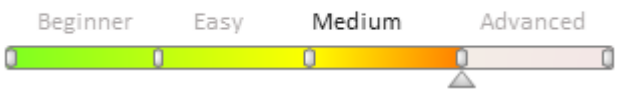
You can address the service, e.g., from a browser (fig.1) either with preliminary login or without it.

Fig. 1. Example of access to the anonymous service from browser.



How to call configuration services using Postman

Difficulty level



Introduction

To integrate with bpm’online **configuration services** you need to execute HTTP requests to these services. Requests can be compiled in any programming language: C#, PHP, etc. However, it is recommended to use HTTP request debugging tools, such as Postman or Fiddler for better understanding of general principles for request formatting. This article provides examples of requests to bpm’online configuration services using Postman.

> Note
>
> An example of a request to bpm’online service using Fiddler is described in the “**Executing OData queries using Fiddler**” article.

Preliminary settings

This article uses a custom configuration service created as described in the “**How to create custom configuration service** article. Use the following link to access the package for setting up configuration service to your application. The package setup procedure is covered in the “**Installing marketplace applications from a zip archive**” article.

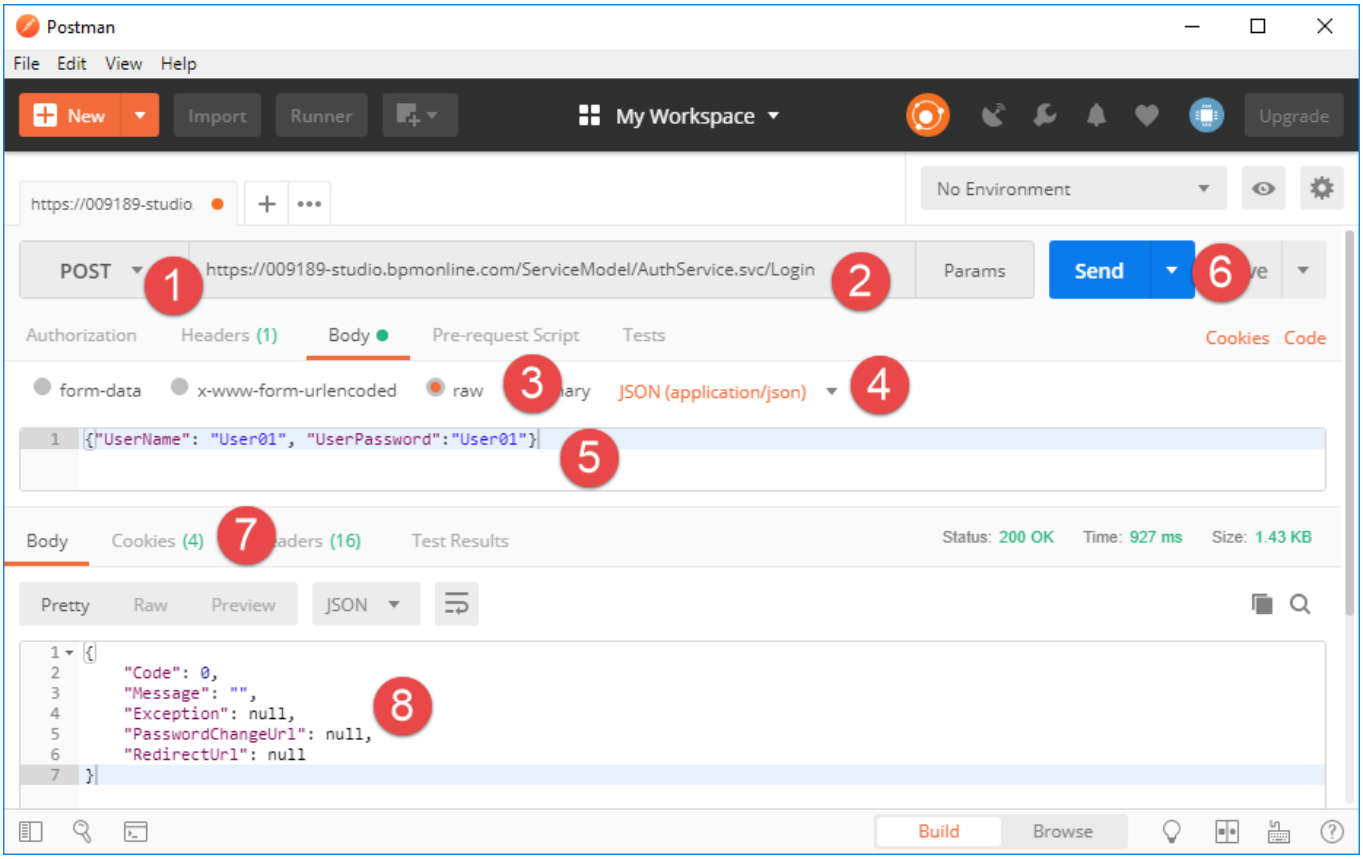
Authentication

Before making requests to configuration service, a third party application must be authenticated and receive the corresponding *cookies*. Bpm’online authentication uses a separate *AuthService.svc* web service (see “**The AuthService.svc authentication service**”).

> NOTE
>
> Authentication is not needed to access the **anonymous services**.

To execute a request to *AuthService.svc* using Postman, do the following (Fig. 1):

Fig. 1. Authentication service request



1. Select the POST HTTP method.

2. Specify the authentication service URL. URL is generated according to the following mask:

```
http(s)://[bpm'online application address]/ServiceModel/AuthService.svc/Login
```

Example:

```
https://009189-studio.bpmonline.com/ServiceModel/AuthService.svc/Login
```

3. Select the "raw" method of request body generation on the [Body] tab.

4. Specify the request body type (JSON (application/json)).

5. Add the request body – a JSON object with the authentication data (login and password):

```
{"UserName": "User01", "UserPassword":"User01"}
```

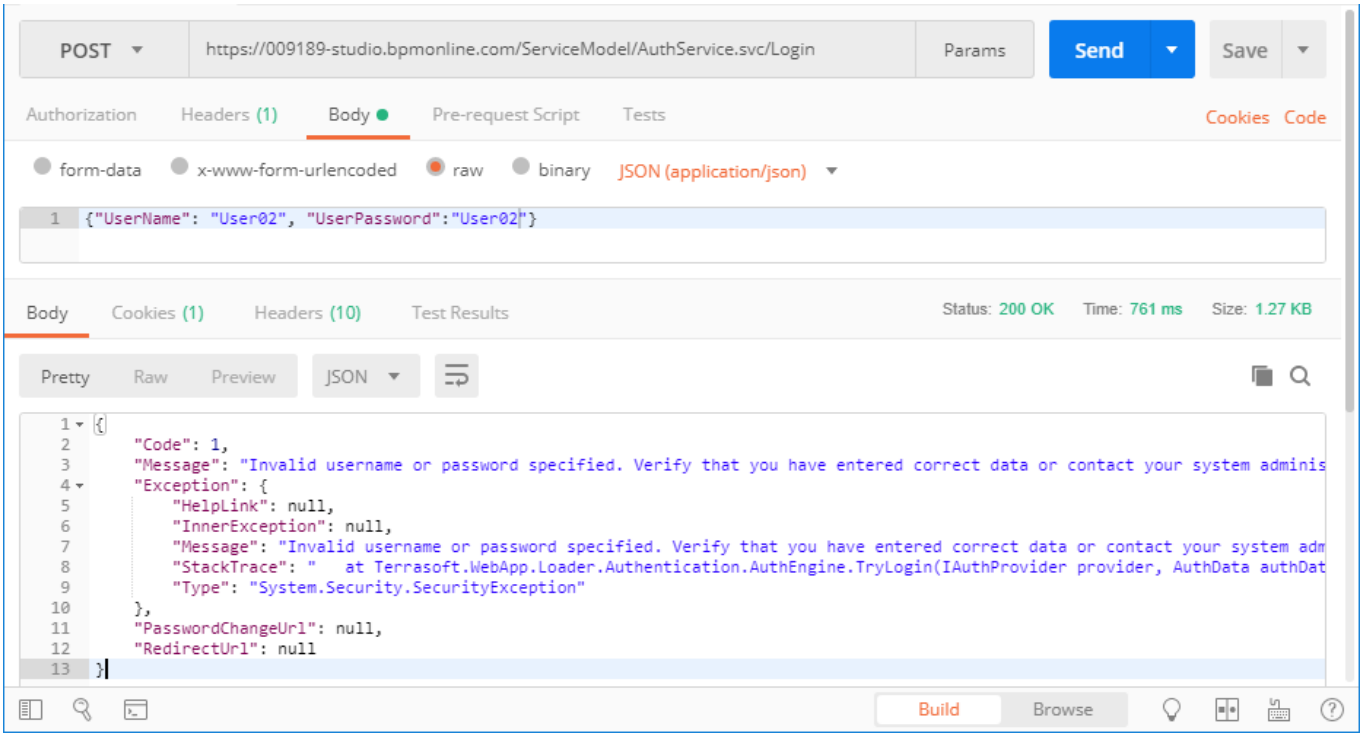
6. Click the [Send] button to send the request to the service.

> ⚠ ATTENTION
>
> The *cookies* received in the HTTP response (BPMLOADER, .ASPXAUTH, BPMCSRF and UserName) are to be used in all further requests to bpm'online services that require authentication data. You can view the received *cookies* on the [Cookies] tab (Fig. 1, 7).
>
> Using the BPMCSRF *cookie* and BPMCSRF token (see below) is required when protection from CSRF attacks is enabled. For more information, see "**Protection from CSRF attacks during integration with bpm'online**".
>
> Protection from CSRF attacks is disabled on bpm'online trial websites. Therefore, there is no need to use both BPMCSRF *cookie* and token in the request titles.

If the authentication has been successful, the response body will contain a JSON object whose *Code* property will be set to "0" (Fig. 1, 8). In case of errors, JSON object properties will contain corresponding code and message. For example, if a user specified in step 5 is not added to the application, the authentication service will return an

incorrect login and password error (Fig. 2).

Fig. 2. Authentication service request containing invalid user data

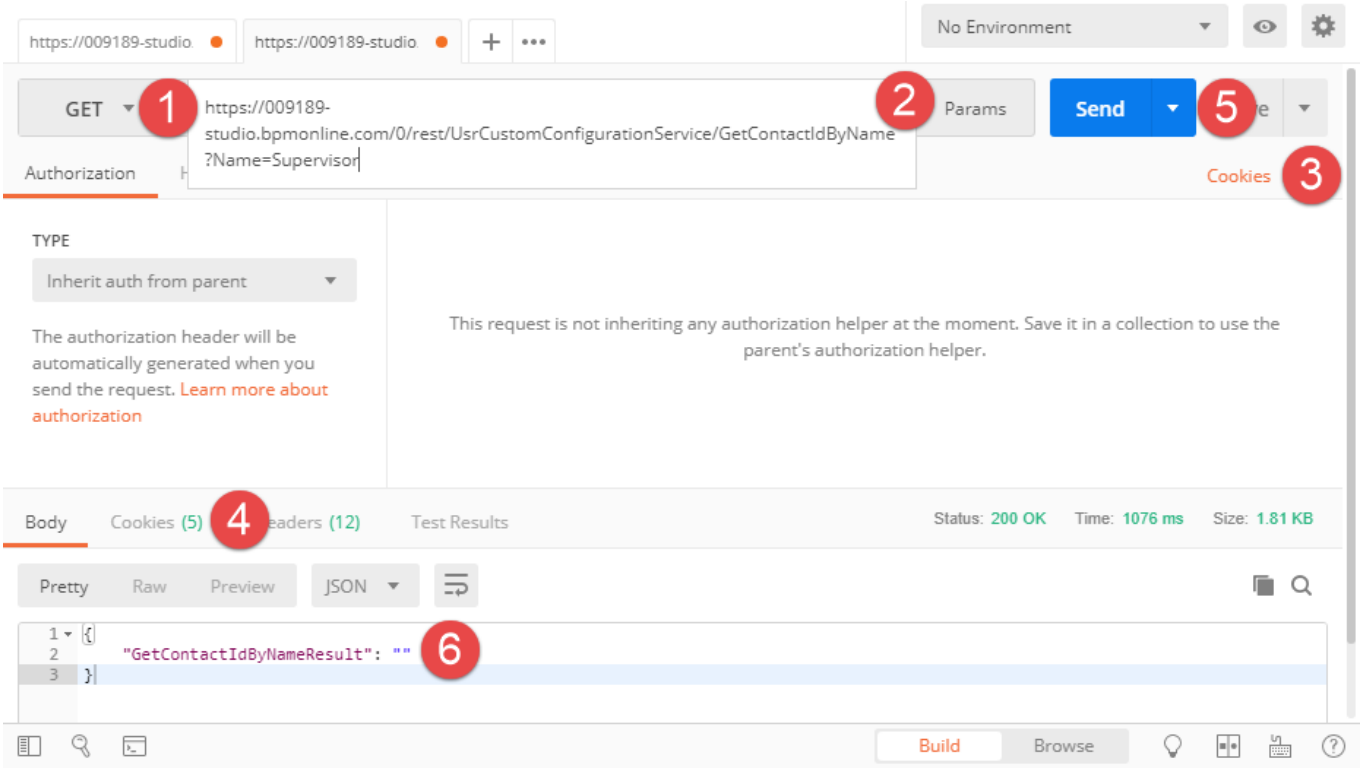


Configuration service request

> **Note**
>
> The *UsrCustomConfigurationService* configuration service used in this article (see “Preliminary settings”, “**How to create custom configuration service**”) can only process HTTP requests via the GET method. Such requests do not have any body. Add the corresponding request body (for example, as described in step 5 of the “Authentication” section), if you need to execute other types of requests.

To generate the *UsrCustomConfigurationService* configuration service request (Fig. 3):

Fig. 3. Configuration service request



1. Select the GET HTTP method.

2. Specify the configuration service URL. URL is generated according to the following mask:

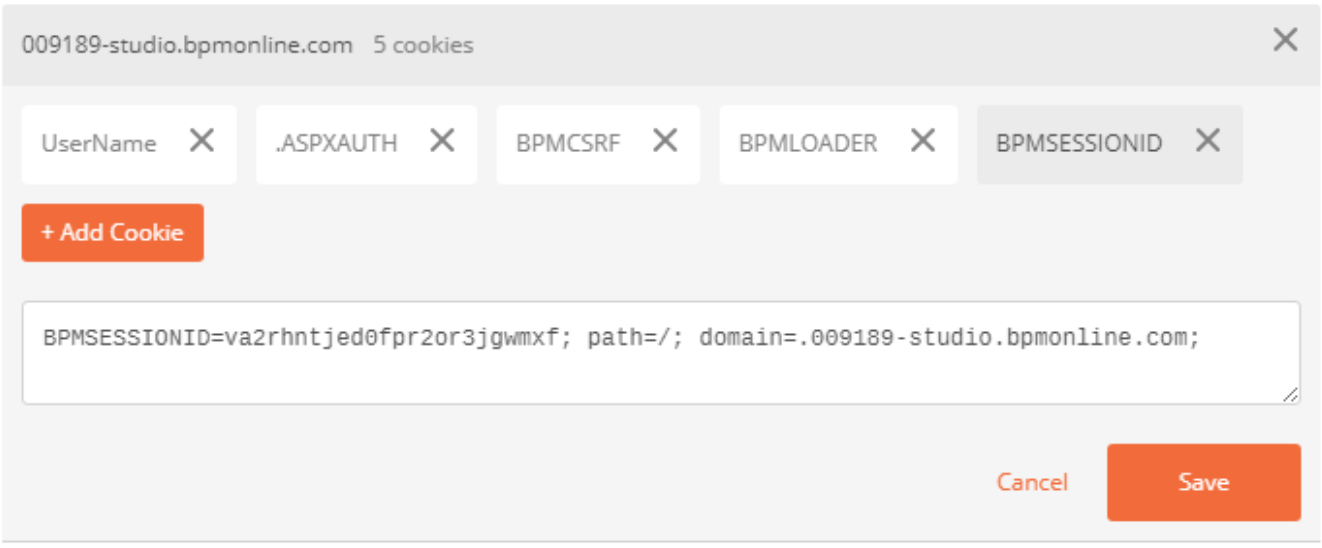
```
[Application Address]/0/rest/[Configuration Service Name]/[Custom Service Endpoint]?
[Optional Parameters]
```

Example:

```
https://009189-
studio.bpmonline.com/0/rest/UsrCustomConfigurationService/GetContactIdByName?
Name=User01
```

3. Add the necessary cookies BPMLOADER, .ASPXAUTH, BPMCSRF and UserName) received in the HTTP authentication service response (Fig. 4).

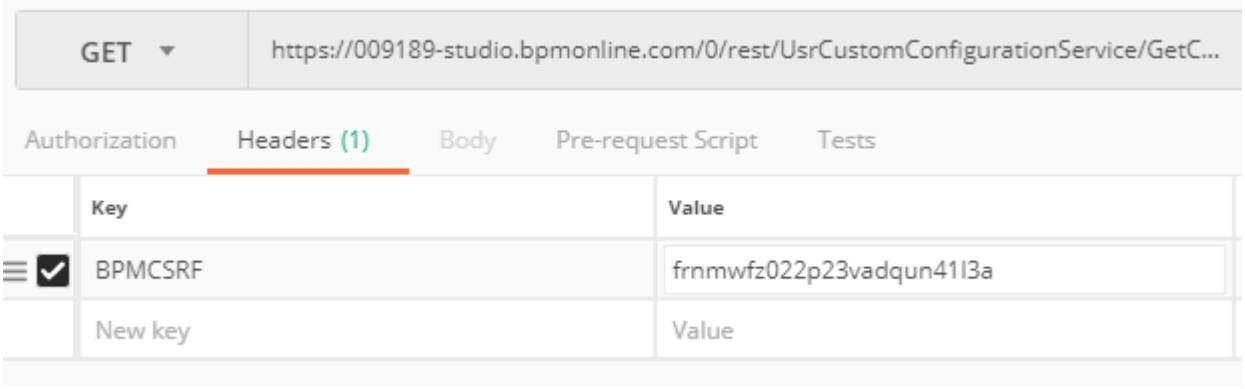
Fig. 4. Adding cookie to a request



ATTENTION

Using the BPMCSRF *cookie* and BPMCSRF token (see below) is required when protection from CSRF attacks is enabled. Add the "key-value" pair to the request caption. Use "BPMCSRF" as a key and the BPMCSRF cookie value as a value (Fig. 5).

Fig. 5. Adding the BPMCSRF token to the request



4. Add the BPMSESSIONID cookie to all requests except for the first one after the authentication.

ATTENTION

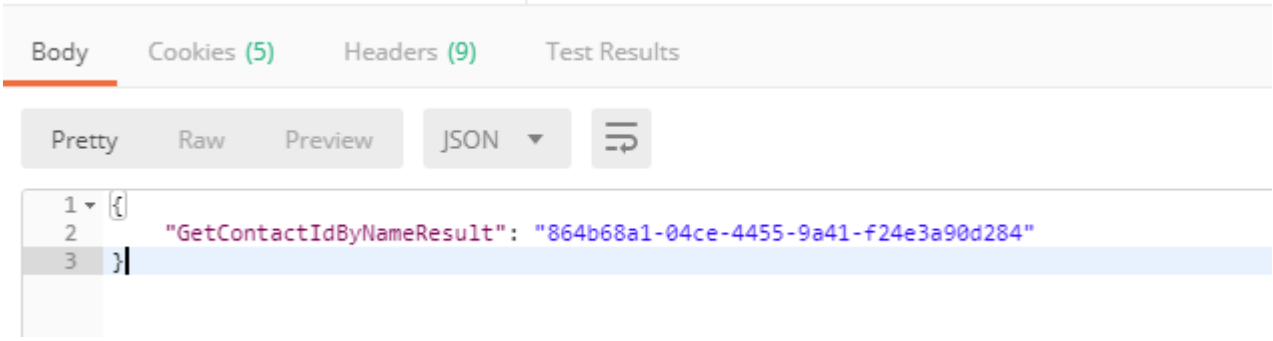
User session will only be created when executing the first configuration service request. The BPMSESSIONID *cookie* will be returned in the HTTP response (see the [Cookies] tab of the HTTP response fig. 3, 4). Therefore, there is no need to add the BPMSESSIONID *cookie* to the title of the first request (Fig. 6, 4).

If you do not add the BPMSESSIONID *cookie* to each subsequent request, each new request will create a new user session. Significant frequency of requests (several or more requests per minute) will increase the RAM consumption which will decrease the performance.

5. Click the [Send] button to send the request to the service.

If the contact specified in the Name parameter is not detected in bpm'online, the "GetContactIdByNameResult" property of the JSON object that was returned in the HTTP response will contain the "" value (fig. 3, 4). If the contact exists, the service will return its identifier (fig. 6).

Fig. 6. Request result



Reading multilingual data with EntitySchemaQuery

Difficulty level

Introduction

Bpm'online has supported multilingual data since version 7.8.3. That means the list data is displayed based on the preferred user language ("culture"). Please refer to the "**Working with data structure**" article for more information on data localization.

Reading multilingual data with EntitySchemaQuery

EntitySchemaQuery (ESQ) is a base mechanism for reading the bpm'online database data. ESQ supports multilingual data by default.

The multilingual data sampling is performed according to the following rules:

- Users with the primary culture (English) receive the main table data.
- Users with additional culture receive the localization table data. If the localization table contains no data for the user's culture, the main table data is returned.

Example of a localized column query generation

A query generation sample code for the localized [Name] column of the [City] object schema on the server side (C#):

```
// User Connection.
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];
// Forming a query.
var esqResult = new EntitySchemaQuery(userConnection.EntitySchemaManager, "City");
// Adding columns to a query.
esqResult.AddColumn("Name");
// Executing a database query and retrieving the entire resulting object collection.
var entities = esqResult.GetEntityCollection(userConnection);
// Retrieving the query text.
var s = esqResult.GetSelectQuery(userConnection).GetSqlText();
// Returning the result.
return s;
```

NOTE

This code can be added to the custom **configuration service** method, for example.

If a default culture is selected in the user profile, the following SQL query will be generated:

```
SELECT
    [City].[Name] [Name]
FROM
    [dbo].[City] [City] WITH(NOLOCK)
```

If any culture other than the primary culture is selected in the user profile, the generated SQL query will take into account the localized values for the seleted culture.

```
SELECT
    ISNULL([SysCityLcz].[Name], [City].[Name])[Name]
FROM
    [dbo].[City] [City] WITH(NOLOCK)
    LEFT OUTER JOIN [dbo].[SysCityLcz] [SysCityLcz] WITH(NOLOCK) ON
([SysCityLcz].[RecordId] = [City].[Id]
    AND [SysCityLcz].[SysCultureId] = @P1)
```

The @P1 parameter takes the record identifier value (*Id*) of the selected culture from the *SysCulture* table.

Disabling the data localization mechanism

To disable the data localization selection mechanism (even if the query is executed on behalf of a user with one of the additional cultures), you must set the ESQ instance to *false* for the *UseLocalization* property.

```
// User Connection.
```

```
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];
// Forming a query.
var esqResult = new EntitySchemaQuery(userConnection.EntitySchemaManager, "City");
// Adding a column to a query.
esqResult.AddColumn("Name");
// Disabling the data localization mechanism.
esqResult.UseLocalization = false;
// Executing a database query and retrieving the entire resulting object collection.
var entities = esqResult.GetEntityCollection(userConnection);
//  Retrieving the query text.
var s = esqResult.GetSelectQuery(userConnection).GetSqlText();
//  Returning the reult.
return s;
```

Regardless of which culture is selected in the user's profile, the following SQL query will be generated:

```
SELECT
    [City].[Name] [Name]
FROM
    [dbo].[City] [City] WITH(NOLOCK)
```

Custom culture data selection

ESQ enables you to select culture data different from the current user culture and the default culture. To select the custom culture data, call the *SetLocalizationCultureId(Guid cultureId)* method in the ESQ instance before data retrieval, and pass the *id* of the culture with the necessary data to it.

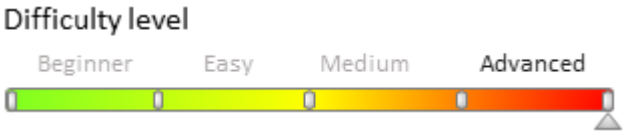
```
// User Connection.
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];

// Retriveing the id of the necessary culture (e.g. italian).
var sysCulture = new SysCulture(userConnection);
if (!sysCulture.FetchPrimaryInfoFromDB("Name", "it-IT"))
    {
    // Error: The record is not found.
    return "The culture is not found";
}
Guid italianCultureId = sysCulture.Id;

// Forming a query.
var esqResult = new EntitySchemaQuery(userConnection.EntitySchemaManager, "City");
// Adding a column to a query.
esqResult.AddColumn("Name");
// Installing the necessary localization.
esqResult.SetLocalizationCultureId(italianCultureId);
// Executing a database query and retrieving the entire resulting object collection.
var entities = esqResult.GetEntityCollection(userConnection);
//  Retrieving the query text.
var s = esqResult.GetSelectQuery(userConnection).GetSqlText();
//  Returning the reult.
return s;
```

As the result, the following SQL inquiry is generated:

```
SELECT
    ISNULL([SysCityLcz].[Name], [City].[Name])[Name]
FROM
    [dbo].[City] [City] WITH(NOLOCK)
    LEFT OUTER JOIN [dbo].[SysCityLcz] [SysCityLcz] WITH(NOLOCK) ON
([SysCityLcz].[RecordId] = [City].[Id]
    AND [SysCityLcz].[SysCultureId] = @P1)
```

The @P1 parameter takes the record identifier value (id) stored in the *italianCultureId* variable.

Views localization



Introduction

Views are often used for data sampling. Views can sample data from localizable columns. Additional configurations of localizable views is required for localized data sampling.

To localize a view:

1. Create a view object schema. Add a multilingual checkbox to the localizable columns. Please refer to “**Adding a multilingual terminator to an object schema**” for more details.
2. Add a new localization view in the database.

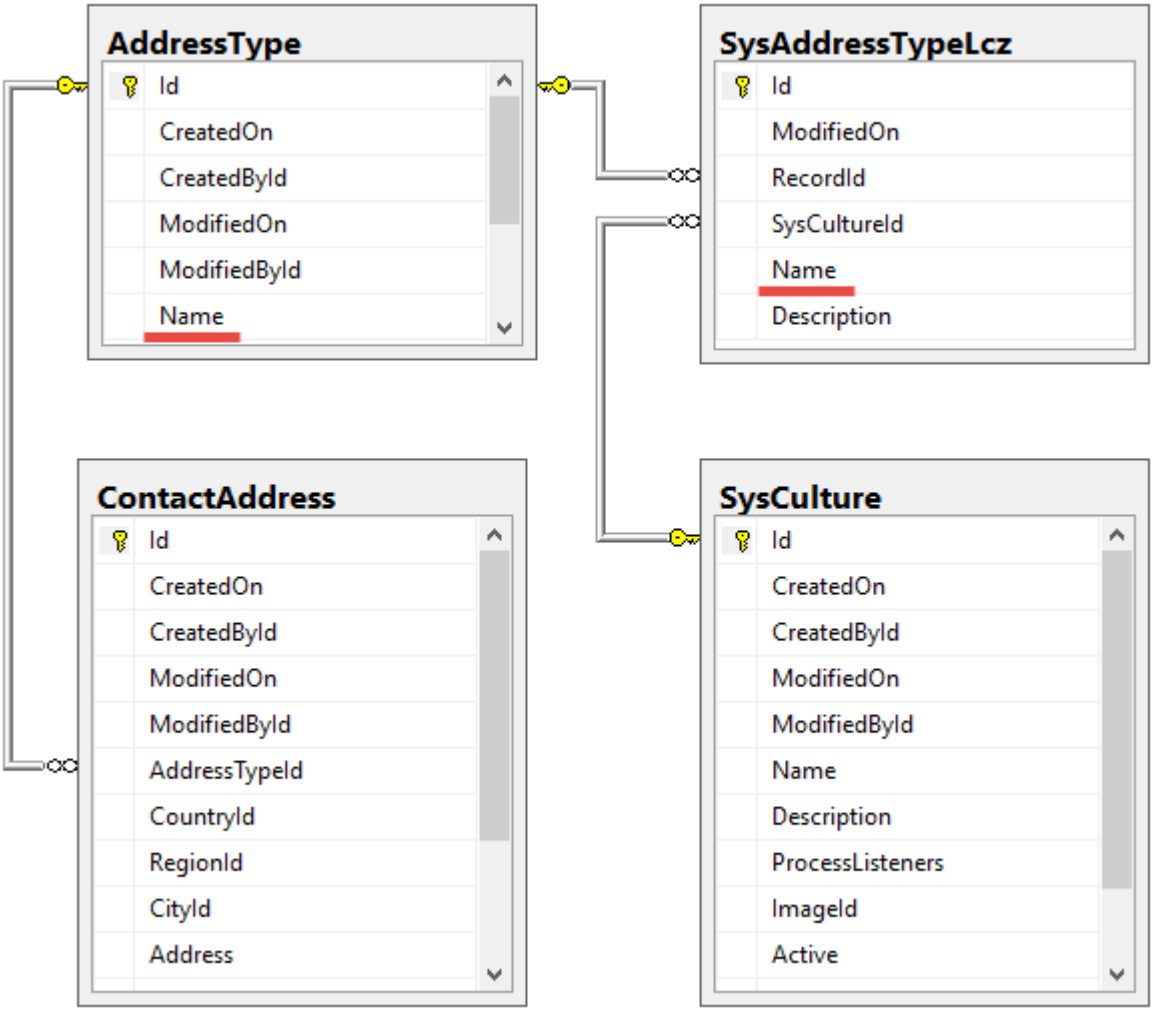
Case description

One of the columns in the *ContactAddress* object schema refers to the *AddressType* lookup (schema). The *Name* column of the AddressType schema is localizable.

Since almost every object schema in bpm'online corresponds to a database table, the table binding structure is (Fig. 1):

- *ContactAddress* – contact address table. Linked to the *AddressType* table through the *AddressTypeId* column.
- *AddressType* – address type table. The *Name* column contains values that match the default user language (“culture”). The values of other cultures are located in the *SysAddressTypeLcz* table.
- *SysAddressTypeLcz* – an auto-generated system table of localizable address type values. Linked to the *AddressType* table through the *RecordId* column, and to the *SysCulture* table through the *SysCultureId* column. The localizable address type values of a culture (specified in the *SysCultureId* column) are located in the *Name* column.
- *SysCulture* – a system table with a list of cultures.

Fig. 1. Structure and relationships of tables for [Contact address], [Address type] schemas and localization tables.



Create a view that samples the following fields:

- *ContactAddress.Address* – contact address.
- *AddressType.Name* – localizable address type name.

Case implementation:

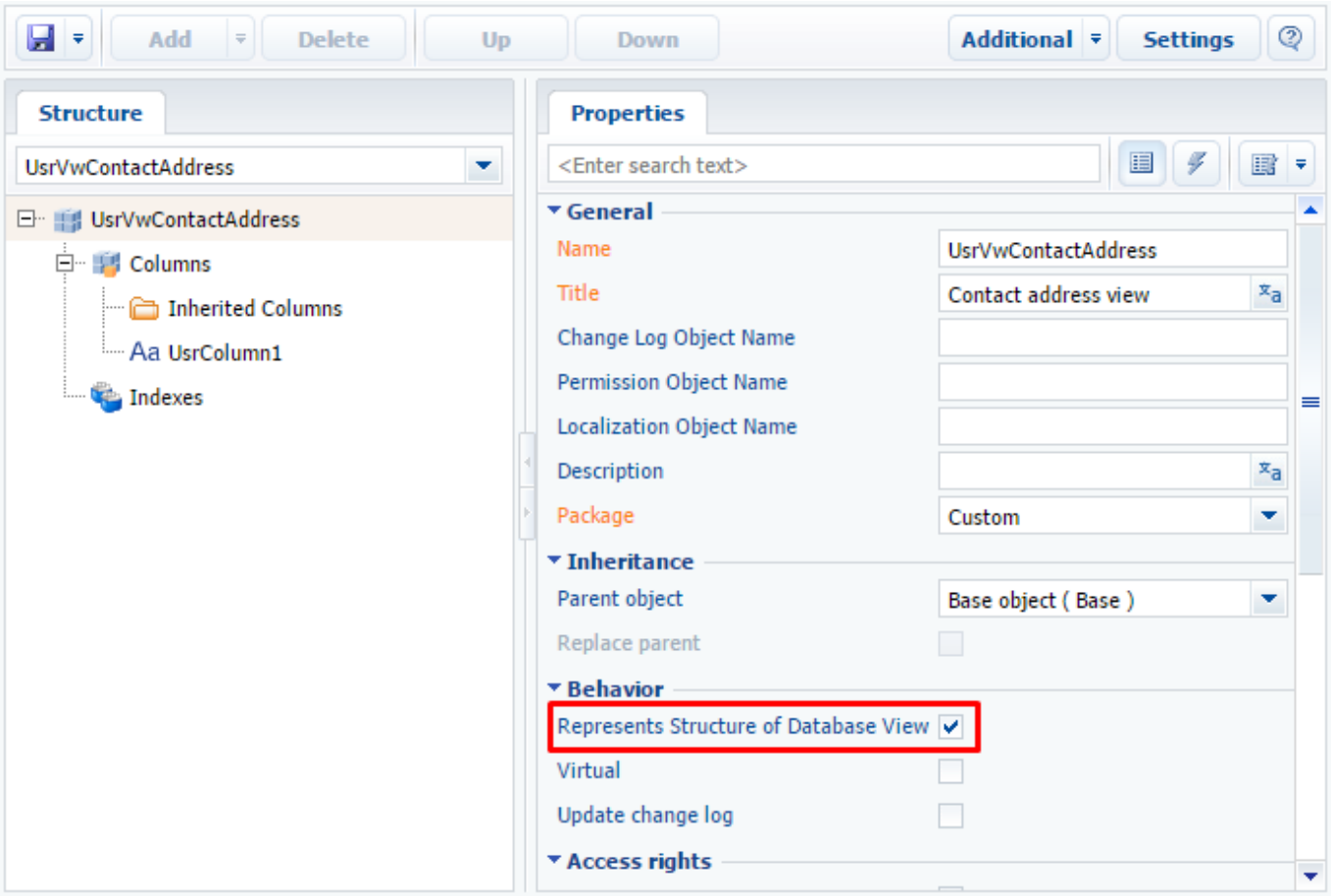
1. Create a view object schema

Learn more about creating object schemas and adding columns to them in the “**Creating the entity schema**” article.

Create an object schema with the following parameters (Fig. 1):

- [Name] – “UsrVwContactAdress”. The *Usr* prefix corresponds to the value of the [Prefix for object name] system setting. The *Vw* prefix (short for *View*) indicates that the schema is a database view.
- [Title] – “Contact address view”.
- [Package] – the name of the package used for development (see: “**Creating and installing a package for development**”).
- [Parent object] – “Base object”.
- [Represents Structure of Database View] – required (Fig. 2).

Fig. 2. The view in the database checkbox



When the values are populated, we recommend saving the schema metadata or pre-publishing the object.

Add two text columns to the created schema.

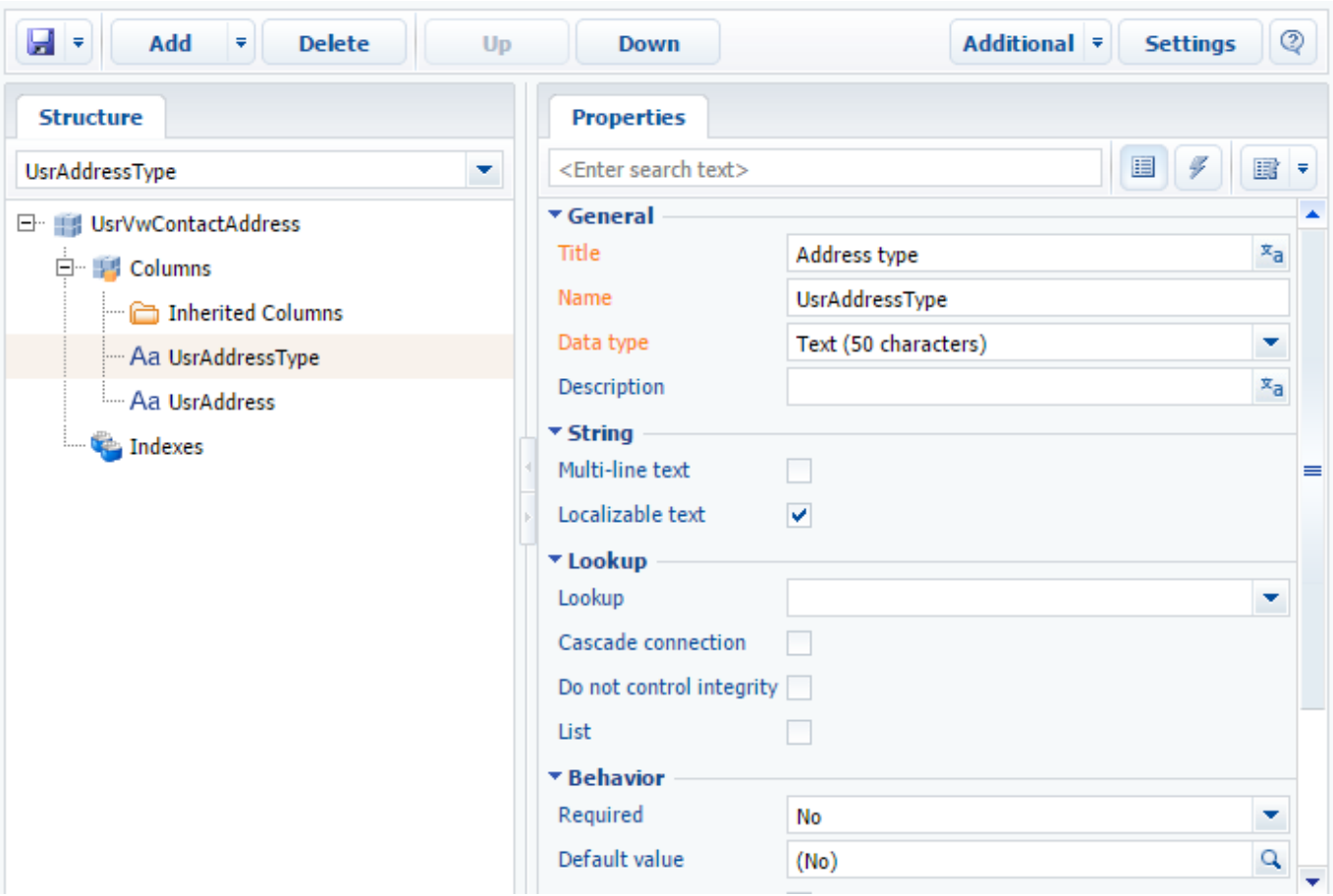
The first column will contain unlocalizaed address values in the default culture. Set the following parameter values:

- [Name] – “UsrAdress”. The *Usr* prefix corresponds to the value of the [Prefix for object name] system setting.
- [Title] – “Address”.
- [Data type] – “Text (50 characters)”. The string may be set to a different maximum length. The amount of memory used in the database depends on the number of characters.

The second column will contain localized address type values. Set the following parameter values:

- [Name] – “UsrAddressType”. The *Usr* prefix corresponds to the value of the [Prefix for object name] system setting.
- [Title] – “Address type”.
- [Data type] – “Text (50 characters)”.
- [Localizable text] – required (Fig. 3). Learn more about multilingual checkboxes in the “**Adding a multilingual terminator to an object schema**” article.

Fig. 3. A multilingual checkbox in a column



NOTE

The [Data type] and [Localizable text] properties are displayed in the extended column properties display mode. (See: “ **Workspace of the Object Designer**”).

2. Creating a database view

Execute the following SQL script to create the *UsrVwContactAddress* view in the database.

```
-- The name of the view must match the name of the schema table.
CREATE VIEW dbo.UsrVwContactAddress
AS
SELECT
    ContactAddress.Id,
    -- View columns should correspond with schema columns.
    ContactAddress.Address AS UsrAddress,
    AddressType.Name AS UsrAddressType
FROM ContactAddress
INNER JOIN AddressType ON ContactAddress.AddressTypeId = AddressType.Id;
```

Execute the following SQL script to create the *UsrVwContactAddress* view in the database.

```
-- The name of the view must match the name of the schema localization table.
CREATE VIEW dbo.SysUsrVwContactAddressLcz
AS
SELECT
    SysAddressTypeLcz.Id,
    ContactAddress.id AS RecordId,
    SysAddressTypeLcz.SysCultureId,
     -- View columns should correspond with schema columns.
    SysAddressTypeLcz.Name AS UsrAddressType
```

```
FROM ContactAddress
INNER JOIN AddressType ON ContactAddress.AddressTypeId = AddressType.Id
INNER JOIN SysAddressTypeLcz ON AddressType.Id = SysAddressTypeLcz.RecordId;
```

The columns of the localized *UsrVwContactAddress* view must match the columns of the localization tables. Learn more about localization tables in the "**Localization tables**" article.

When sampling the data with *EntitySchemaQuery* from the *UsrAddressType* column of the *UsrVwContactAddress* view, the correct values for different languages will be displayed.

Testing case results

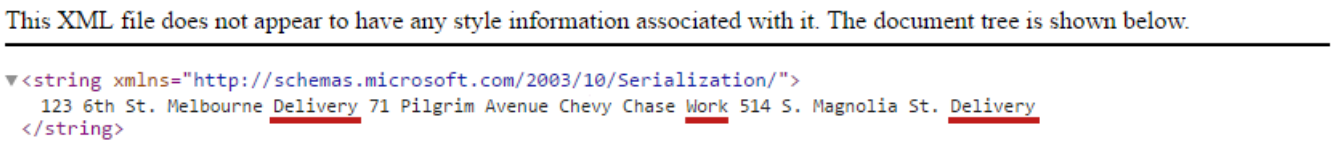
You can use one of the examples in the "**Reading multilingual data with EntitySchemaQuery**" article to check the results. You can use the examples in the "**How to create custom configuration service**" article to check the results of queries.

Implement a method in the created service class that returns a list of addresses and their types from the created non-localized *UsrVwContactAddress* view using the *EntitySchemaQuery* query.

```
[OperationContract]
[WebInvoke(Method = "GET", UriTemplate = "Ex01")]
public string Ex01()
{
    // User connection.
    var userConnection =
(UserConnection)HttpContext.Current.Session["UserConnection"];
    // Forming a query.
    var esqResult = new EntitySchemaQuery(userConnection.EntitySchemaManager,
"UsrVwContactAddress");
    // Adding columns to a query.
    esqResult.AddColumn("UsrAddress");
    esqResult.AddColumn("UsrAddressType");
    // Executing a database query and retrieving the entire resulting objects
collection.
    var entities = esqResult.GetEntityCollection(userConnection);
    // Displaying results.
    var s = "";
    foreach (var item in entities)
    {
        s += item.GetTypedColumnValue<string>("UsrAddress") + Environment.NewLine;
        s += item.GetTypedColumnValue<string>("UsrAddressType") +
Environment.NewLine;
    }
    return s;
}
```

The result of this method is shown in Fig. 4.

Fig. 4. The default localization test results



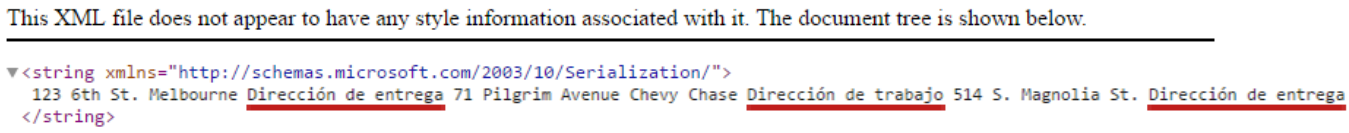
To test the performance of a localized view in the created service class, you must implement the second method:

```
[OperationContract]
[WebInvoke(Method = "GET", UriTemplate = "Ex01")]
public string Ex01()
{
    var userConnection =
```

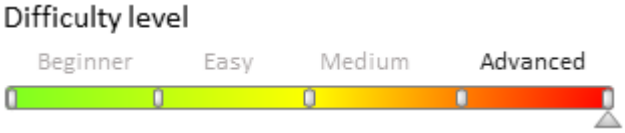
```
(UserConnection)HttpContext.Current.Session["UserConnection"];
    // Retrieving the Id of the necessary culture, e.g. Spanish.
    var sysCulture = new SysCulture(userConnection);
    if (!sysCulture.FetchPrimaryInfoFromDB("Name", "es"))
    {
        return "Culture not found";
    }
    Guid CultureId = sysCulture.Id;
    var esqResult = new EntitySchemaQuery(userConnection.EntitySchemaManager,
"UsrVwContactAddress");
    esqResult.AddColumn("UsrAddress");
    esqResult.AddColumn("UsrAddressType");
    // Selecting the required localization.
    esqResult.SetLocalizationCultureId(CultureId);
    var entities = esqResult.GetEntityCollection(userConnection);
    var s = "";
    foreach (var item in entities)
    {
        s += item.GetTypedColumnValue<string>("UsrAddress") + Environment.NewLine;
        s += item.GetTypedColumnValue<string>("UsrAddressType") +
Environment.NewLine;
    }
    return s;
}
```

The result of this method is shown in Fig. 4.

Fig. 5. The selected localization test results



Working with the localized data via Entity



Introduction

Starting with version 7.9.1, an ability of getting the multilingual data was added to the *Entity.FetchFromDB()* method. The data fetching algorithm is similar to the EntitySchemaQuery algorithm (see " **Reading multilingual data with EntitySchemaQuery**" article):

1. The object will receive the data from the main table if current user culture (language) is the primary culture for the application.
2. The object will receive the data from the **localization table** if current user culture (language) is different from the primary culture. If the localization table contains no data for the user's culture, the main table data is returned.

The examples of using the *Entity.FetchFromDB()* and *Entity.Save()* method overloads and the analysis of their execution for the user with the main (English) and additional (German) cultures (languages) are given below. These methods can be used in the user service methods (see the " **How to create custom configuration service**" article).

Reading the data

The example of source code for getting the data from the *Name* localized column of the *AccountType* schema object on the server side (C#);

```
// A user connection.
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];
// Getting the [Account Type] schema.
EntitySchema schema =
userConnection.EntitySchemaManager.FindInstanceByName("AccountType");
// Creating an instance of the Entity (object).
Entity entity = schema.CreateEntity(userConnection);
// A collection of column names for the fetch.
List<string> columnNamesToFetch = new List<string> {
    "Name",
    "Description"
};
//Get the data for an object with the "Customer" value in the [Name] column.
entity.FetchFromDB("Name", "Customer", columnNamesToFetch);
// Forming and sending a response.
var name = String.Format("Name: {0}", entity.GetTypedColumnValue<string>("Name"));
return name;
```

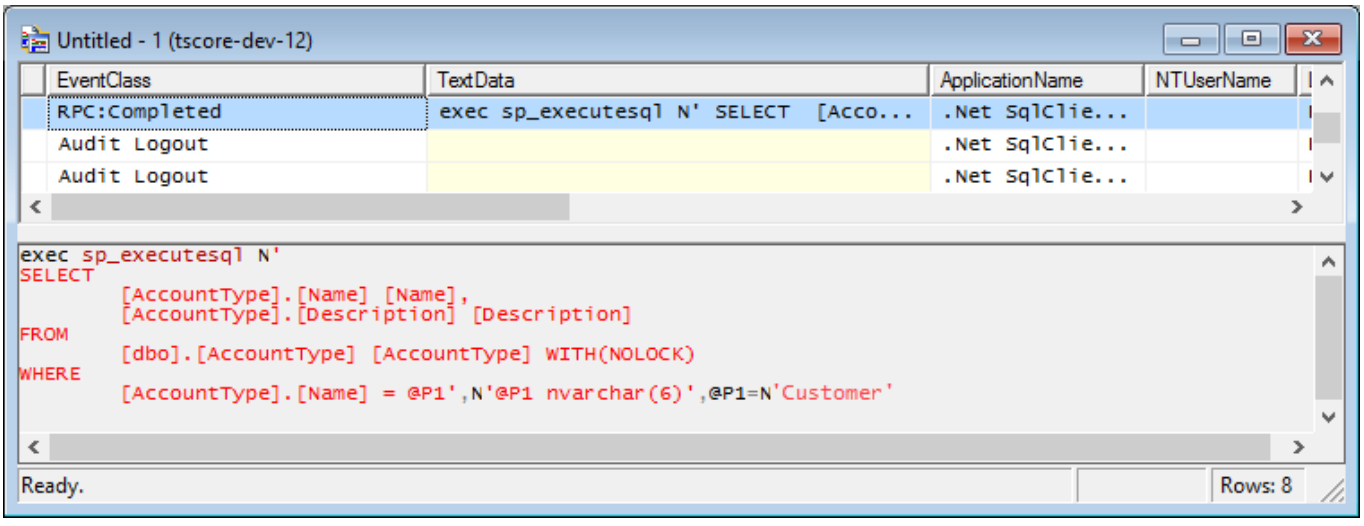
If a user who has a default language selected in the profile executes the method containing this code, the following query will be sent to the database:

```
exec sp_executesql N'
SELECT
    [AccountType].[Name] [Name],
    [AccountType].[Description] [Description]
FROM
    [dbo].[AccountType] [AccountType] WITH(NOLOCK)
WHERE
    [AccountType].[Name] = @P1',N'@P1 nvarchar(6)',@P1=N'Customer'
```

In the above query, the "Customer" value is specified in the @P1 parameter, it determines the corresponding record of the database table.

> NOTE
>
> You can view the request using the SQL Server Profiler (Fig. 1).

Fig. 1. Profiling a query into a database via SQL Server Profiler



If a user with an additional language (such as German) selected in the profile executes the method, the following

query will be sent to the database:

```
exec sp_executesql N'
SELECT
    ISNULL([SysAccountTypeLcz].[Name], [AccountType].[Name]) [Name],
    ISNULL([SysAccountTypeLcz].[Description], [AccountType].[Description])
[Description]
FROM
    [dbo].[AccountType] [AccountType] WITH(NOLOCK)
    LEFT OUTER JOIN [dbo].[SysAccountTypeLcz] [SysAccountTypeLcz] WITH(NOLOCK) ON
([SysAccountTypeLcz].[RecordId] = [AccountType].[Id]
    AND [SysAccountTypeLcz].[SysCultureId] = @P2)
WHERE
    [AccountType].[Name] = @P1',N'@P1 nvarchar(6),@P2
uniqueidentifier',@P1=N'Клиент',@P2='A5420246-0A8E-E111-84A3-00155D054C03'
```

In the above query, the "Customer" value is specified in the @P1 parameter, it determines the corresponding record of the main database table. The indicator of additional culture from the *SysCulture* table will be in the @P2 parameter. It will define the corresponding record from the *SysAcountTypeLcz* localization table.

Thus, for the user with English culture the *name* variable will have the "Customer" value and for the user with German culture it will be the "Kunde" value.

Saving the localized data

The *Entity.SetColumnValue()* method is used for adding and modifying the localized data. This method can accept arguments of *string* and *LocalizableString* types.

Saving the localized data using string argument

The following saving algorithm is used in passing the *string* argument to the *Entity.SetColumnValue()* method:

- when the user with an additional culture creates a new record, the data is added to both the main table and the localization table (for the corresponding culture);
- when the user with an additional culture modifies the existing *Entity* instance, the result is saved only in the localization table (for the corresponding culture);
- when the user with the main culture creates or modifies the *Entity* object, the data will be added or modified in the main table of the object.

The code example of saving the data using string argument:

```
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];
EntitySchema schema =
userConnection.EntitySchemaManager.FindInstanceByName("AccountType");
Entity entity = schema.CreateEntity(userConnection);
// Set the default values for the columns.
entity.SetDefColumnValues();
// Setting the value for the [Name] column.
entity.SetColumnValue("Name", "New customer");
// Saving.
entity.Save();
var name = String.Format("Name: {0}", entity.GetTypedColumnValue<string>("Name"));
return name;
```

When the user with the default (English) culture executes this code, the following query will be executed in the database:

```
exec sp_executesql N'
INSERT INTO [dbo].[AccountType]([Id], [Name], [CreatedOn], [CreatedById],
[ModifiedOn], [ModifiedById], [ProcessListeners], [Description])
    VALUES(@P1, @P2, @P3, @P4, @P5, @P6, @P7, @P8)',N'@P1 uniqueidentifier,@P2
nvarchar(12),@P3 datetime2(7),@P4 uniqueidentifier,@P5 datetime2(7),@P6
uniqueidentifier,@P7 int,@P8 nvarchar(4000)',@P1='3A820BC8-006D-42B7-A472-
```

```
E331FBD73E20',@P2=N'New Customer',@P3='2017-02-10 09:40:23.0909251',@P4='410006E1-
CA4E-4502-A9EC-E54D922D2C00',@P5='2017-02-10 09:40:23.0929256',@P6='410006E1-CA4E-
4502-A9EC-E54D922D2C00',@P7=0,@P8=N''
```

In the above query, the "New customer" value is specified in the @P2 parameter, it is saved in the main database table.

If the user has an additional culture (German) set in their profile, the following code must be executed to save the data with the string argument:

```
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];
EntitySchema schema =
userConnection.EntitySchemaManager.FindInstanceByName("AccountType");
Entity entity = schema.CreateEntity(userConnection);
entity.SetDefColumnValues();
entity.SetColumnValue("Name", "Neue kunden");
entity.Save();
var name = String.Format("Name: {0}", entity.GetTypedColumnValue<string>("Name"));
return name;
```

The query to the *AccountType* main table will be the same as to the main localization, but the "Neue Kunden" value will be specified in the @P2 parameter.

```
exec sp_executesql N'
INSERT INTO [dbo].[AccountType]([Id], [Name], [CreatedOn], [CreatedById],
[ModifiedOn], [ModifiedById], [ProcessListeners], [Description])
    VALUES(@P1, @P2, @P3, @P4, @P5, @P6, @P7, @P8)',N'@P1 uniqueidentifier,@P2
nvarchar(12),@P3 datetime2(7),@P4 uniqueidentifier,@P5 datetime2(7),@P6
uniqueidentifier,@P7 int,@P8 nvarchar(4000)',@P1='94052A88-499D-4072-A28A-
6771815446FD',@P2=N'Neue Kunden',@P3='2017-02-10 10:07:00.3454424',@P4='410006E1-
CA4E-4502-A9EC-E54D922D2C00',@P5='2017-02-10 10:07:00.3454424',@P6='410006E1-CA4E-
4502-A9EC-E54D922D2C00',@P7=0,@P8=N''
```

In addition, the query will be executed in the localization table:

```
exec sp_executesql N'
INSERT INTO [dbo].[SysAccountTypeLcz]([Id], [ModifiedOn], [RecordId], [SysCultureId],
[Name])
    VALUES(@P1, @P2, @P3, @P4, @P5)',N'@P1 uniqueidentifier,@P2 datetime2(7),@P3
uniqueidentifier,@P4 uniqueidentifier,@P5 nvarchar(12)',@P1='911A721A-0E5A-4CC3-B6D9-
9E5FE85FEC64',@P2='2017-02-10 10:07:00.3664442',@P3='94052A88-499D-4072-A28A-
6771815446FD',@P4='A5420246-0A8E-E111-84A3-00155D054C03',@P5=N'Neue Kunden'
```

The "Neue Kunden" value will be specified in the @P5 parameter in the above request.

The value that does not correspond to the default culture will be added to the *AccountType* table.

To avoid this save the localized data using the localized strings.

Saving the localized data using localized string argument

The code example of saving the data using the localized string:

```
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];
EntitySchema schema =
userConnection.EntitySchemaManager.FindInstanceByName("AccountType");
Entity entity = schema.CreateEntity(userConnection);
entity.SetDefColumnValues();

// Creating a localized string with localized values for different cultures.
var localizableString = new LocalizableString();
localizableString.SetCultureValue(new CultureInfo("en-US"), "New customer en-US");
localizableString.SetCultureValue(new CultureInfo("de-DE"), "Neue Kunden de-DE");
```

```
// Seting the value of the column using the localized string.
entity.SetColumnValue("Name", localizableString);
entity.Save();

// The result will be displayed in the current user culture.
var name = String.Format("Name: {0}", entity.GetTypedColumnValue<string>("Name"));
return name;
```

Regardless of the language in the user's profile, the following queries will be sent to the database upon the code execution:

1. The query with the “New customer en-US” value in the @P2 argument will be sent to the main page:

```
exec sp_executesql N'
INSERT INTO [dbo].[AccountType]([Id], [Name], [CreatedOn], [CreatedById],
[ModifiedOn], [ModifiedById], [ProcessListeners], [Description])
    VALUES(@P1, @P2, @P3, @P4, @P5, @P6, @P7, @P8)',N'@P1 uniqueidentifier,@P2
nvarchar(18),@P3 datetime2(7),@P4 uniqueidentifier,@P5 datetime2(7),@P6
uniqueidentifier,@P7 int,@P8 nvarchar(4000)',@P1='5AC81E4A-FCB2-4019-AE5B-
0C485A5F65BD',@P2=N'New Customer en-US',@P3='2017-02-10
10:47:21.7471581',@P4='410006E1-CA4E-4502-A9EC-E54D922D2C00',@P5='2017-02-10
10:47:21.7511578',@P6='410006E1-CA4E-4502-A9EC-E54D922D2C00',@P7=0,@P8=N''
```

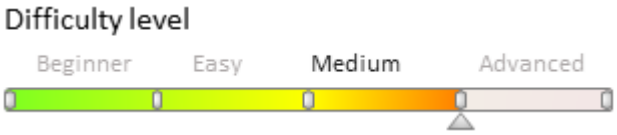
2. The query with the “Neue kunden de-DE” value in the @P5 argument will be sent to the localization page:

```
exec sp_executesql N'
INSERT INTO [dbo].[SysAccountTypeLcz]([Id], [ModifiedOn], [RecordId], [SysCultureId],
[Name])
    VALUES(@P1, @P2, @P3, @P4, @P5)',N'@P1 uniqueidentifier,@P2 datetime2(7),@P3
uniqueidentifier,@P4 uniqueidentifier,@P5 nvarchar(18)',@P1='6EC9C205-7F8B-455E-BC68-
3D9AA6D7B7C0',@P2='2017-02-10 10:47:21.9272674',@P3='5AC81E4A-FCB2-4019-AE5B-
0C485A5F65BD',@P4='A5420246-0A8E-E111-84A3-00155D054C03',@P5=N'Neue Kunden de-DE'
```

Attention!

If the code is be executed by a user who has an additional culture set in the profile and the value for the default culture is not specified in the localization string, the record for the user’s culture will be added to the primary *AccountType* table.

Adding a multilingual terminator to an object schema



Introduction

There is often a need to localize one or more columns of the object schema. As in, certain record data must display in multiple languages according to the culture selected in the user profile.

To create an object schema with localizable columns:

1. Create a new or a replacing object schema.

2. Add the localizable columns, if necessary. Select [Localizable text] in the column properties.

ATTENTION

You can only localize text columns.

Case description

Create the [Localizable object] object schema with the localizable [Name] column.

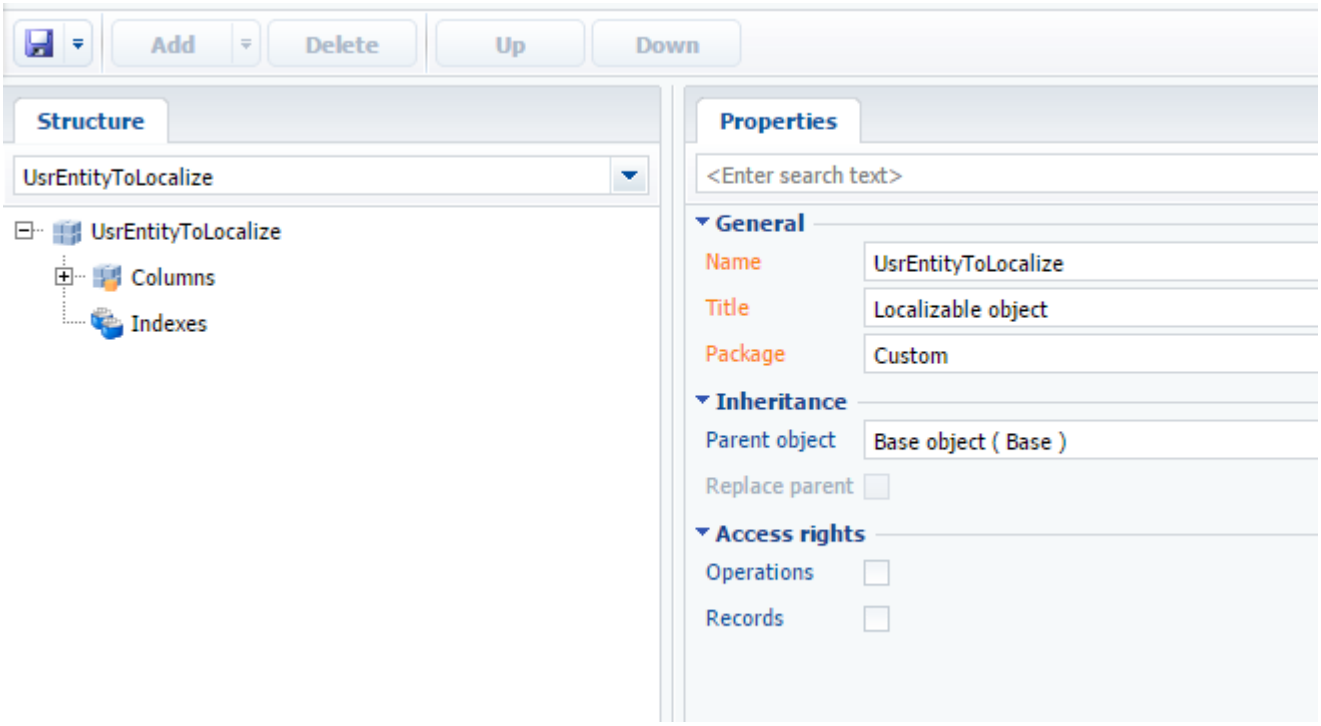
Case implementation algorithm

1. Creating an object schema

Learn more about creating object schemas in the “**Creating the entity schema**” article. According to the case, you need to create an object schema with the following parameters (Fig. 1):

- [Title] – ”Localizable object”
- [Name] – “UsrEntityToLocalize";
- [Parent object] – “Base object (Base)”

Fig. 1. The [Localizable object] schema properties



2. Adding the necessary columns for localization

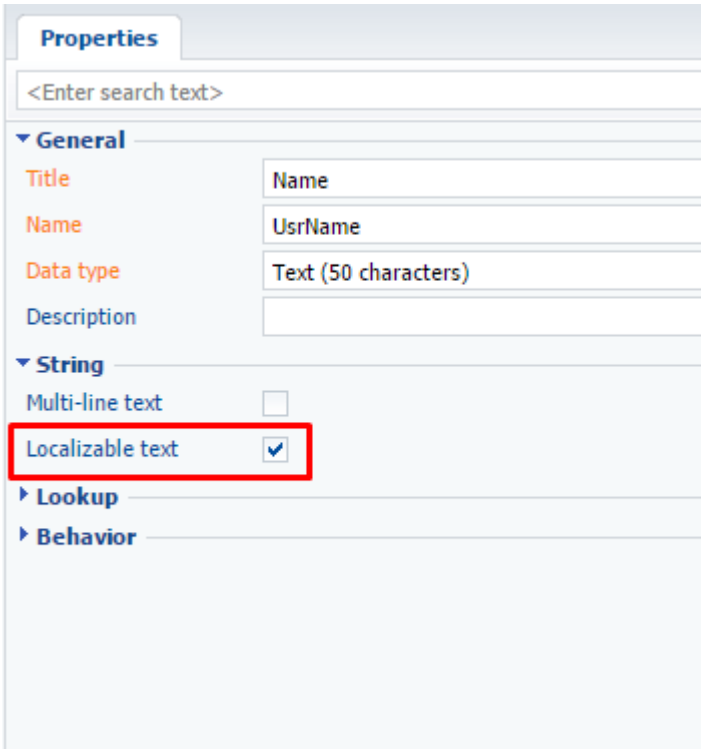
Add the column to the created schema with the following properties:

- [Title] – “Name”;
- [Name] – “UsrName”. The *Usr* prefix must match the [Prefix for object name] system setting;
- [Data type] – “Text (50 characters)”

Learn more about adding an object schema column in the “**Creating the entity schema**” article.

Select [Localizable object] in the added column properties (Fig. 2). The checkbox is only available in the advanced mode of the object designer (see “**Workspace of the Object Designer**”).

Fig. 2. The [Name] column properties

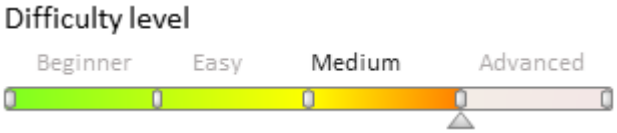


Publish the schema to apply the changes.

A *SysUsrEntityToLocalizeLcz* localization table will be created for the *UsrEntityToLocalize* object schema after publishing. All localized data for all localizable columns will be kept there.

Examples of adding data to localization tables using the *Entity* instance of class and reading such data are described in the “**Working with the localized data via Entity**” article. An example of reading localizable data using *EntitySchemaQuery* is described in the “**Reading multilingual data with EntitySchemaQuery**”.

Using the DBExecutor for working with the database



Introduction

Using a number of streams in working with the database via *UserConnection* may cause to issues in starting synchronization or committing transactions.

ATTENTION

The issue occurs during the work with the database even without direct using of the *DBExecutor*. For example, it can be used indirectly, through the *EntitySchemaQuery*.

ATTENTION

As unmanaged resources are used to work with the database, you need to wrap the *DBExecutor* in the *using* operator. Another way is to call the *Dispose()* method explicitly to release resources. More information about the *using* operator can be found in the "C# Guide” documentation.

An example of misuse

A source code fragment with incorrect usage of the the DBExecutor is given below. You cannot call the DBExecutor instance methods in the parallel threads.

```
// Create a parallel thread.
var task = new Task(() => {
    // Using a DBExecutor instance in a parallel thread.
    using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection()) {
        dbExecutor.StartTransaction();
        //...
        dbExecutor.CommitTransaction();
    }
});
// Running an asynchronous task in a parallel thread.
// The execution of the program in the main thread continues on.
task.Start();
//...
var select = (Select)new Select(UserConnection)
        .Column("Id")
        .From("Contact")
        .Where("Name")
        .IsEqual(Column.Parameter("Supervisor"));
// Using an instance of DBExecutor in the main thread will cause an error,
// because The instance DBExecutor is already used in a parallel thread.
using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection()) {
    using (IDataReader dataReader = select.ExecuteReader(dbExecutor)) {
        while (dataReader.Read()) {
            //...
        }
    }
}
```

An example of correct use

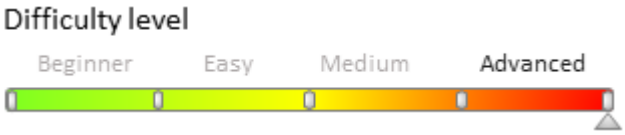
A source code fragment with correct usage of the the DBExecutor is given below. The call of the DBExecutor instance methods is preformed consistently in one thread.

```
// The first use of the instance DBExecutor in the main thread.
using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection()) {
    dbExecutor.StartTransaction();
    //...
    dbExecutor.CommitTransaction();
}
//...
var select = (Select)new Select(UserConnection)
        .Column("Id")
        .From("Contact")
        .Where("Name")
        .IsEqual(Column.Parameter("Supervisor"));
// Reusing the DBExecutor instance in the main thread.
using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection()) {
    using (IDataReader dataReader = select.ExecuteReader(dbExecutor)) {
        while (dataReader.Read()) {
            //...
        }
    }
}
```

Contents

- **How to change the calculation for the "Closed" column in the [Forecasts] section.**
- **Configuration of the editable columns on the product selection page**

How to change the calculation for the "Closed" column in the [Forecasts] section.



Introduction

Use bpm'online to plan your company’s sales turnover and analyze the targets. In the [Forecasts] section, you can generate forecasts using different units of measure registered in the system, and calculate the actual values. This enables to specify the time period for which you want to analyze the sales performance and monitor the overall performance of your department using the summary tables provided in the [Forecasts] section.

More information about the section can be found in the “[Forecasts] section” article.

Case description

Change the logic of calculation of the “Closed” column in the [Forecasts] section: the calculation should be based on invoices instead of sales.

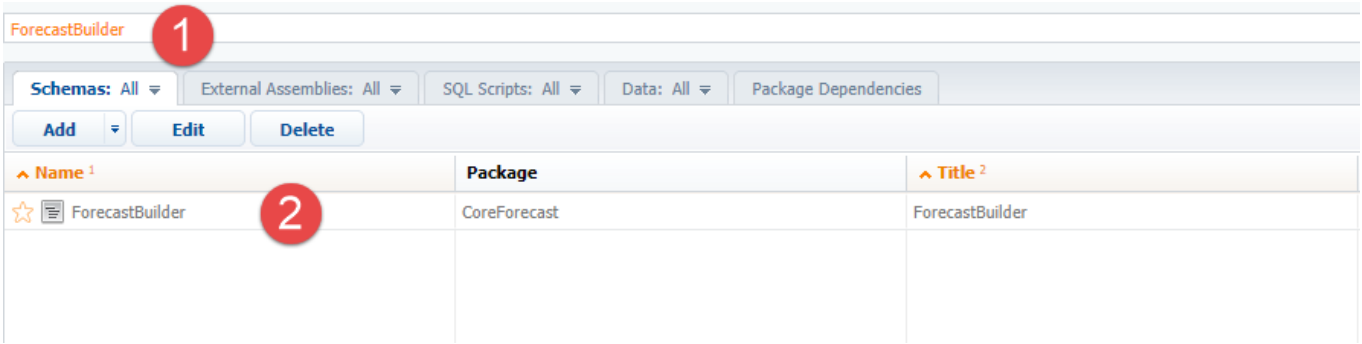
> NOTE
>
> The resulting source codes of the module and stored procedure are available by the link below.

Case implementation algorithm

1. Copy the source code of the forecast building module

To do this, search for the *ForecastBuilder* schema name in the [Configuration] section (Fig.1.1). Double click the found schema (Fig. 1.2) to open the module schema in the module designer (“**Module designer**”).

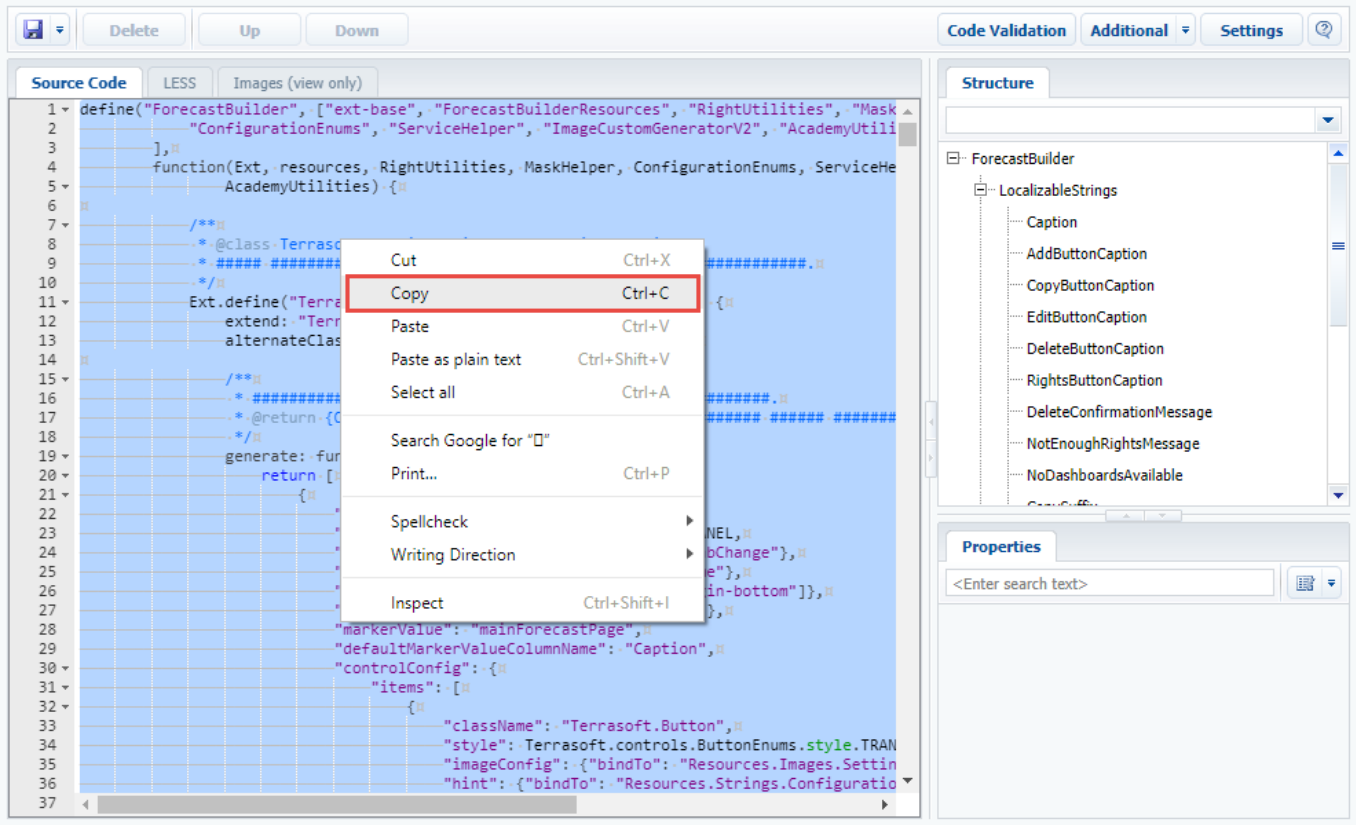
Fig. 1. Search field in the [Configuration] section



> NOTE
>
> The schema is located in the pre-installed package. You will get the massage that changes for this item could not be saved. The pre-installed packages are described in detail in the "**Package structure and contents**” article.

Copy all content from the [Source code] area (Fig. 2) to a text file. .

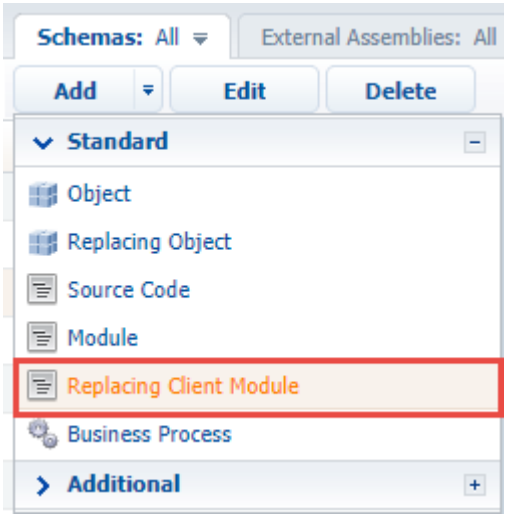
Fig. 2. Copying the source code of the schema



2. Create a replacing forecast building module.

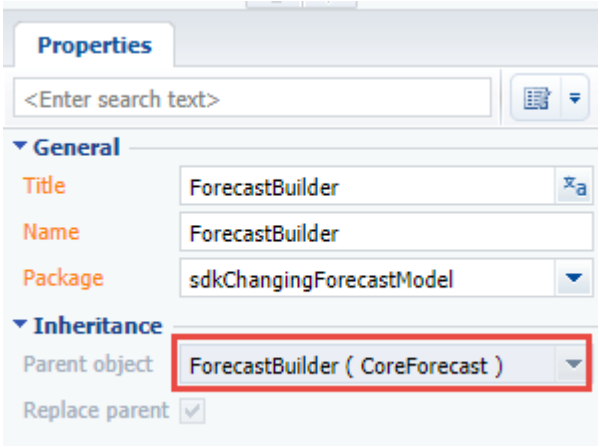
To do this, select a custom package in the [Configuration] section and execute the [Add] -> [Replacing Client Module] action on the [Schemas] tab (Fig. 3). The procedure for creating a replacing custom module is covered in the “**Creating a custom client module schema**”.

Fig. 3. Creating a replacing module



For the created module schema, set the “ForecastBuilder” as a [Parent object] (Fig.4). After that, all schema properties from the parent module will be applied automatically.

Fig. 4. Properties of the module schema



Add the source code of the parent schema (that was copied at the previous step) on the [Source Code] tab and save the schema.

3. Change the method of the forecast page opening

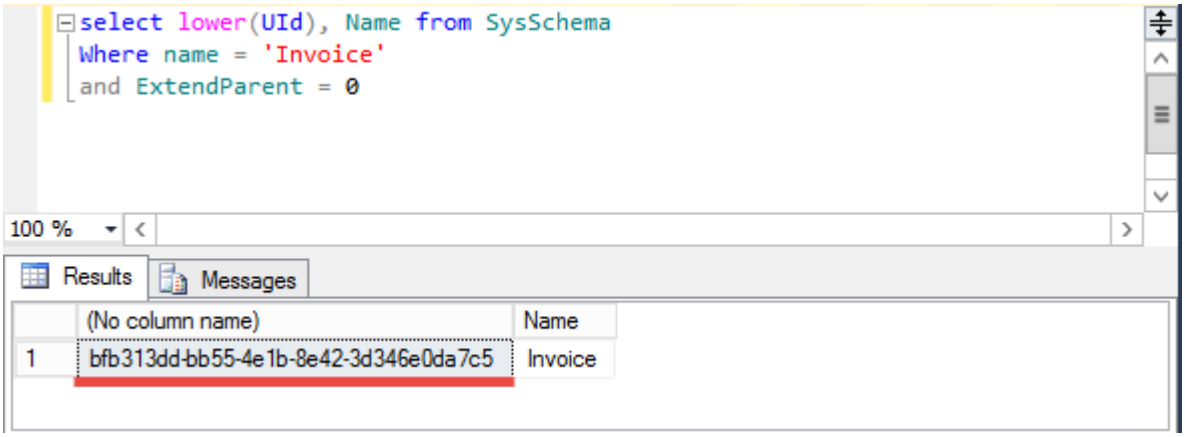
In the source code of the ForecastBuilder module schema, change the values of the *valuePairs* array in the *openForecastPage()* method of the *Terrasoft.configuration.BaseForecastsViewModel* to the values corresponding the schema of the [Invoice] object. The source code for the changes (previous values are commented out):

```
...
openForecastPage: function(moduleId, operation) {
    ...
    var valuePairs = [
        {
            name: "EntitySchemaUId",
            value: "bfb313dd-bb55-4e1b-8e42-3d346e0da7c5" //value: "ae46fb87-c02c-
4ae8-ad31-a923cdd994cf"
        },
        {
            name: "EntitySchemaName",
            value: "Invoice" //value: "Opportunity"
        }
    ];
    ...
},
...
```

You can get the value of the *EntitySchemaUId* Id by executing following SQL query to the bpm'online database (Fig. 5):

```
select lower(UId), Name from SysSchema
Where name = 'Invoice'
and ExtendParent = 0
```

Fig. 5. Request result



Save the schema to apply changes.

5. Modify the tsp_RecalculateForecastFact stored procedure

The *tsp_RecalculateForecastFact* stored procedure recalculates the values of the "Closed" column for selected time period. Executing and applying changes in the stored procedures is described in the "How to: Modify a Stored Procedure (SQL Server Management Studio)" MSDN article.

> ATTENTION
>
> Pay high attention when creating and executing the SQL query. Executing an incorrect SQL query can damage existing data and disrupt the system.

To use invoices for calculation, make following modification to the procedure (previous values are commented out).

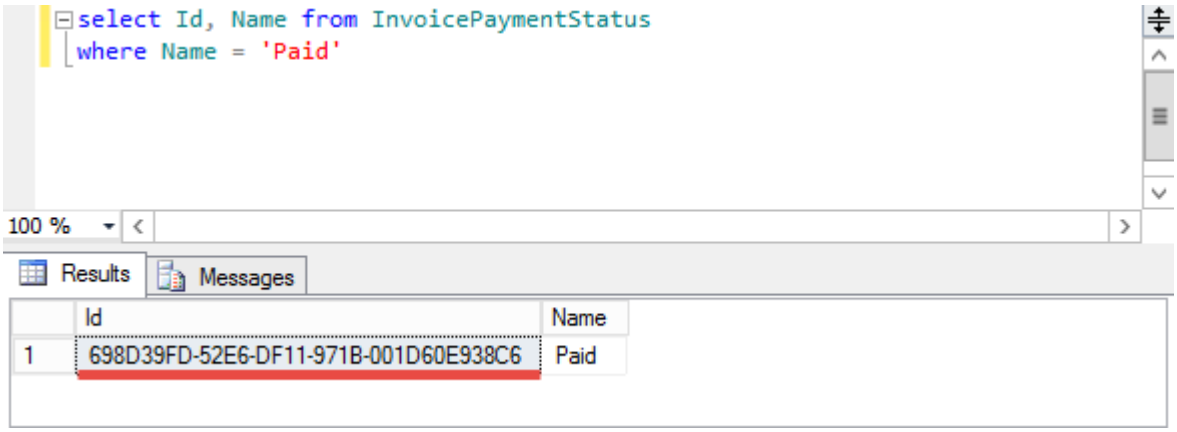
1. Change the value stored in the @CompletedId variable.

```
--SET @CompletedId = '{60D5310C-5BE6-DF11-971B-001D60E938C6}'
SET @CompletedId = '{698D39FD-52E6-DF11-971B-001D60E938C6}'
```

This variable stores the Id of the status of a paid invoice. You can get the value of the variable by executing following SQL query to the bpm'online database (Fig. 6):

```
select Id, Name from InvoicePaymentStatus
where Name = 'Paid'
```

Fig. 6. Request result



2. Change the query which result is stored in the @MaxDueDate variable.

```
--SET @MaxDueDate = (SELECT Convert(Date, MAX(DueDate), 104) FROM Opportunity o WHERE
o.StageId = @CompletedId)
SET @MaxDueDate = (SELECT Convert(Date, MAX(StartDate), 104) FROM Invoice o WHERE
o.PaymentStatusId = @CompletedId)
```

The query searches for the newest paid invoice.

3. Change the subquery expression stored in the @SQLText variable. In this subquery the logic of calculating the "Closed" and "Pipeline" columns is implemented.

```
--Initial value
/*SET @SQLText = N'
    SELECT
    (SELECT SUM(ISNULL(fiv.[Value], 0))
    FROM [ForecastItemValue] fiv
    WHERE fiv.[ForecastItemId] = @P5
    AND fiv.[PeriodId] = @P6
    AND fiv.[ForecastIndicatorId] = @P7
    ) PlanAmount,
    (SELECT SUM(ISNULL(o.[Amount], 0))
    FROM [Opportunity] o
    WHERE o.[StageId] = @P1
    AND o.[DueDate] >= @P2
    AND o.[DueDate] < @P3
    AND o.' + @ColumnName + N' = @P4
    ) FactAmount,
    (SELECT SUM(ISNULL(o.[Amount], 0) * ISNULL(o.[Probability], 0) / 100)
    FROM [Opportunity] o
    INNER JOIN [OpportunityInStage] ois ON ois.[OpportunityId] = o.[Id]
    INNER JOIN [OpportunityStage] os ON os.[Id] = ois.[StageId]
    WHERE os.[End] = 1
    AND ois.[DueDate] >= @P2
    AND ois.[DueDate] < @P3
    AND ois.[Historical] = 0
    AND o.' + @ColumnName + N' = @P4
    ) PotentialAmount'*/
--New value
SET @SQLText = N'
    SELECT
    (SELECT SUM(ISNULL(fiv.[Value], 0))
    FROM [ForecastItemValue] fiv
    WHERE fiv.[ForecastItemId] = @P5
    AND fiv.[PeriodId] = @P6
    AND fiv.[ForecastIndicatorId] = @P7
    ) PlanAmount,
    (SELECT SUM(ISNULL(o.[Amount], 0))
    FROM [Invoice] o
    WHERE o.[PaymentStatusId] = @P1
    AND o.[StartDate] >= @P2
    AND o.[StartDate] < @P3
    AND o.' + @ColumnName + N' = @P4
    ) FactAmount,
    (SELECT SUM(ISNULL(o.[Amount], 0))
    FROM [Invoice] o
    INNER JOIN [InvoicePaymentStatus] os ON os.[Id] = o.[PaymentStatusId]
    WHERE os.[FinalStatus] = 0
    AND o.[StartDate] >= @P2
    AND o.[StartDate] < @P3
    AND o.' + @ColumnName + N' = @P4
    ) PotentialAmount'
```

Run the SQL script (F5 key) to apply the changes.

As a result, the calculation of "Closed" and "Potential" columns will be based on invoices (Fig. 8) istead of sales (Fig. 7).

Fig. 7. Calculating the "Closed" columns by sales

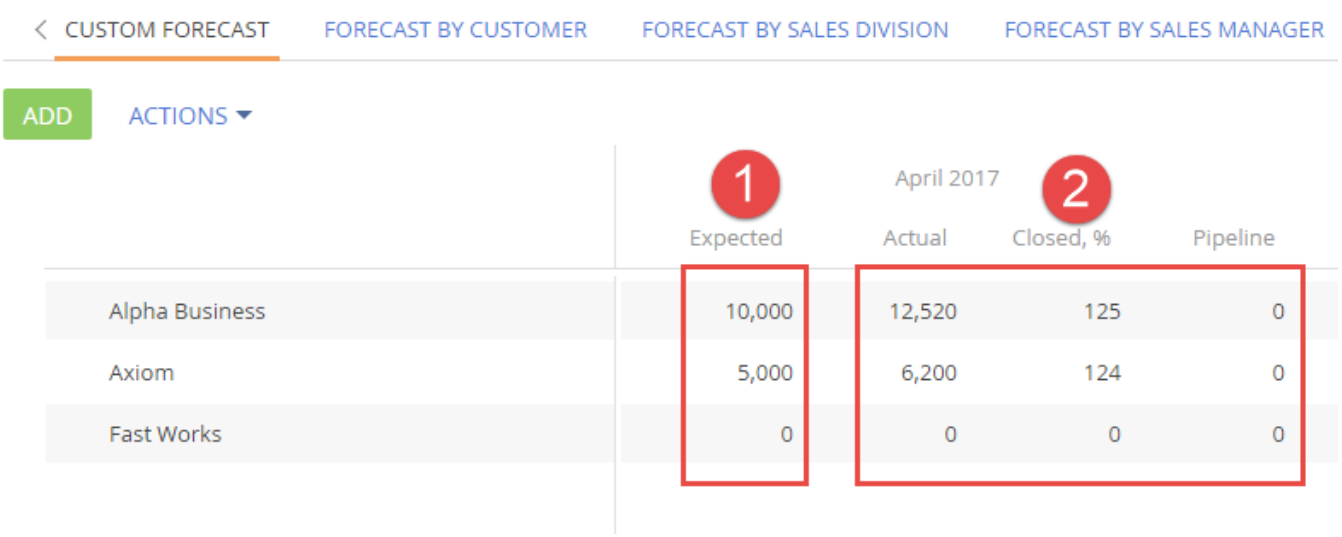
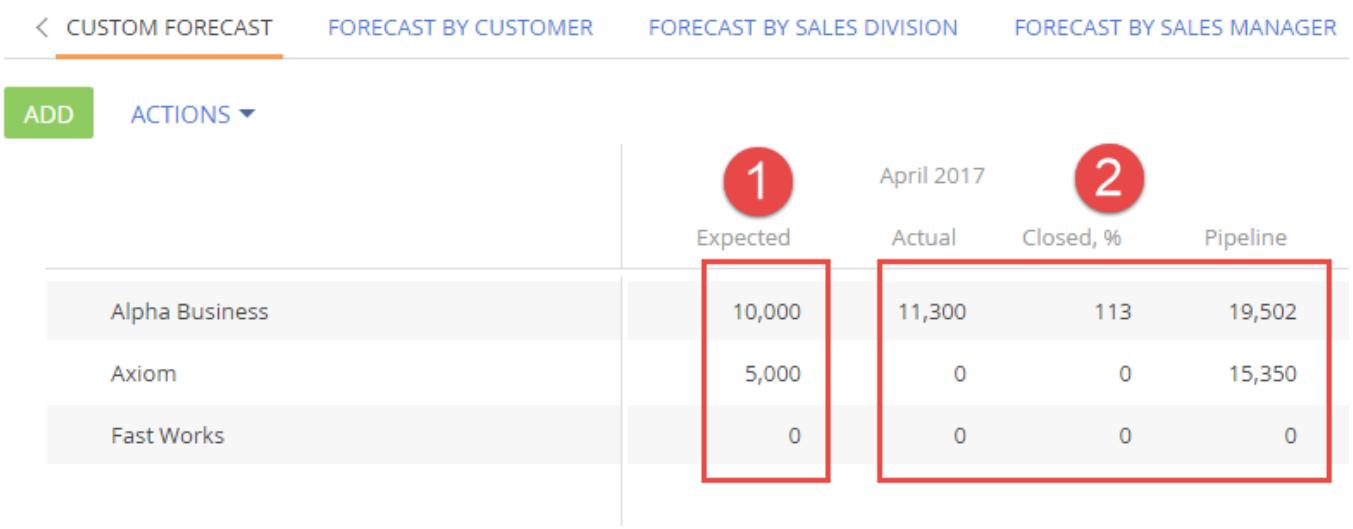
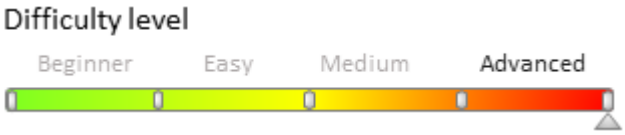


Fig. 8. Calculating the “Closed” columns by invoices



The resulting source codes of the module and stored procedure can be downloaded from the link.

Configuration of the editable columns on the product selection page



Introduction

In the bpm'online version 7.11.2 or higher the editable columns are available on the product selection page. By default the [Quantity], [Unit of measure] and [Price] columns are available for edit. You can also make other columns editable.

Case description

Make editable the [Discount, %] column on the product selection page in the [Orders] section. Add and make editable the [Custom price] column.

NOTE

The case can be also done for the product selection page in the [Invoices] section.

Case implementation algorithm

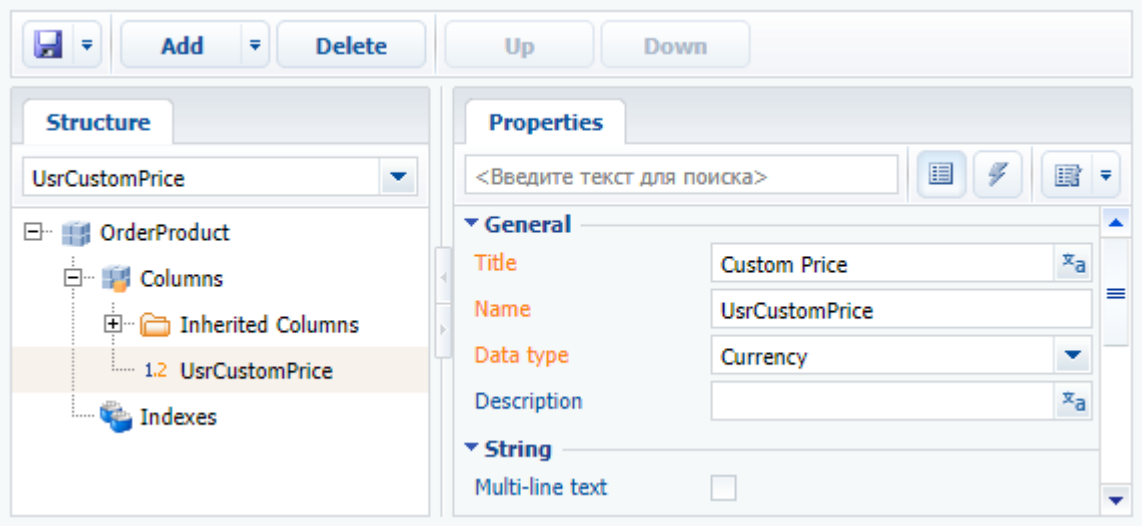
1. Add a custom column to the [Product in order] object

For this, create the [Product in order] replacing object and add a column to it. Creating replacing object and adding a custom column is described in the “**Creating the entity schema**”.

Set following properties for the added column (Fig. 1):

- [Title] – “Custom Price”
- [Name] – "UsrCustomPrice"
- [Data type] – “Currency.”

Fig. 1. Properties of the custom column



Publish the object schema to apply changes.

NOTE

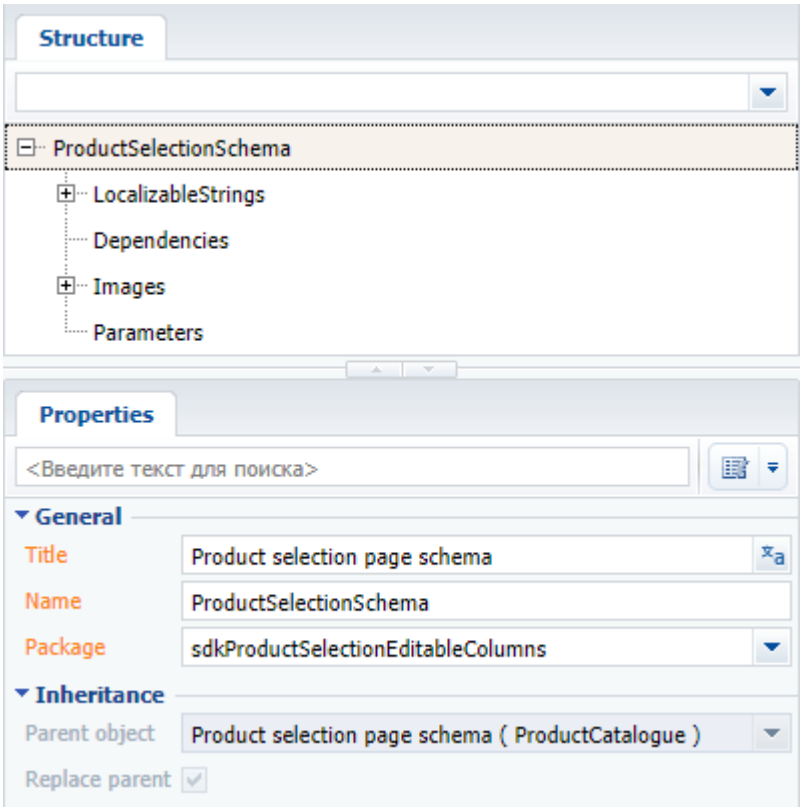
To implement case in the [Invoices] section, perform above steps for the [Product in invoice] object.

2. Create a replacing custom module for the product selection page schema

The procedure of creating a replacing custom module is covered in the “**Creating client schema**”. Set following properties for the for the created module (Fig. 2):

- [Title] – [Product selection page schema]
- [Name] – "ProductSelectionSchema”
- [Parent object] – [Product selection page schema].

Fig. 2. Properties of the replacing client module



Add the following source code on the [Source Code] tab of the schema designer:

```
define("ProductSelectionSchema", [],
    function() {
        return {
            methods: {
                getEditableColumns: function() {
                    // Getting an array of editable columns.
                    var columns = this.callParent(arguments);
                    // Adding the [Discount,%] column to the array of editable
columns.
                    columns.push("DiscountPercent");
                    // Adding a custom column.
                    columns.push("UsrCustomPrice");
                    return columns;
                },
                setColumnHandlers: function(item) {
                    this.callParent(arguments);
                    // Bind the event handler of the user column change event.
                    item.on("change:UsrCustomPrice", this.onCustomPriceChanged,
this);
                },
                // A handler method that will be called when the field value is
changed.
                onCustomPriceChanged: function(item, value) {
                    window.console.log("Changed: ", item, value);
                }
            }
        };
    });
```

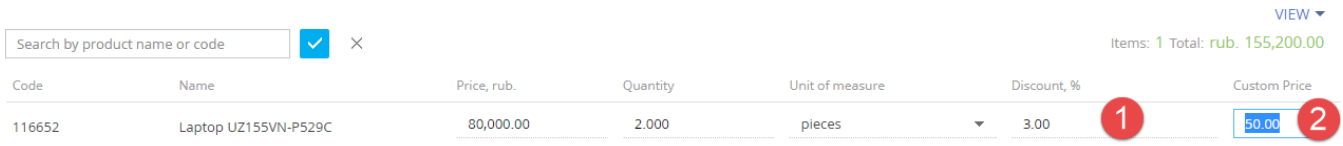
Save the schema to apply changes.

3. Configure columns display on the product selection page

Configure the columns to display them on the product selection page, (see ”Setting up columns”). In title view configuration of the list, add the [Discount, %] and [Custom price] columns.

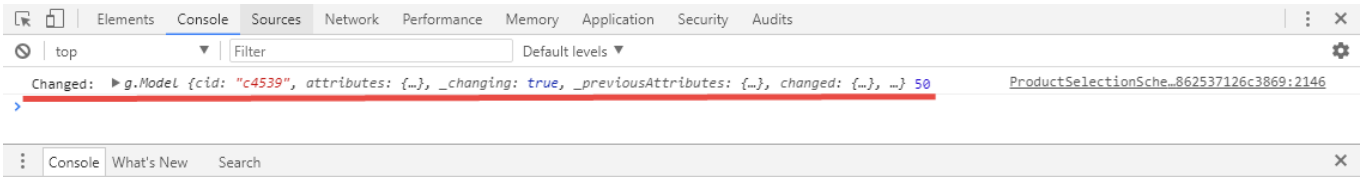
As a result, two editable columns will be displayed on the product selection page (Fig. 3).

Fig. 3. Case result



After modification of the value in the [Custom price] column, the corresponding message will be displayed in the browser console (Fig. 4).

Fig. 4. The result in the browser console

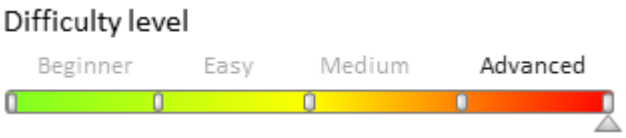


Service products customization

Contents

- **Adding a new rule for calculating case deadline**
- **Adding a macro handler in email templates**
- **Creating Web-to-Case landing pages**
- **How to hide feed area in the agent desktop**
- **Adding floating icons for internal case feed posts**

Adding a new rule for calculating case deadline



Introduction

Bpm’online enables implementing custom logic of receiving parameters for calculating case deadline. When calculating or recalculating a case deadline, a developer implemented strategy is used instead of one of the base calculation strategies.

You can select a specific calculation rule in the [Case deadline calculation rules] lookup. Follow these steps to add a new calculation rule:

1. Create an object schema and add columns necessary for storage of response and resolution deadlines, links to the calendar, service agreement and service.

2. Based on the created object schema, add a lookup and populate it with values needed to calculate the deadline parameters.

3. Add the source code schema and declare the class inherited from the *BaseTermStrategy* abstract class. Implement custom mechanism of receiving response and resolution deadline parameters in the class.

4. Add a new rule.

Case description

Add a custom rule for calculating case deadline parameters for the [Lost data recovery] service as per the [78 — Elite Systems] agreement. Set the following values for the new rule:

- response time – 2 working hours
- resolution time – 1 working day
- used calendar – [Default calendar]

Source code of the case:

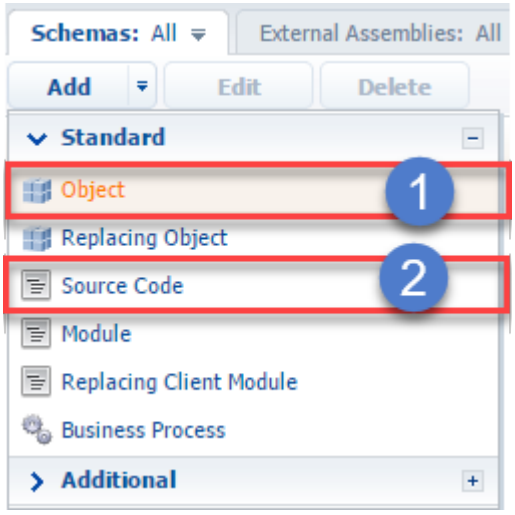
You can download the package with case implementation using the following link.

Case implementation algorithm

1. Creating an object schema containing the necessary columns for calculation

Perform the [Add] – [Object] action on the [Schemas] tab of the [Configuration] section.

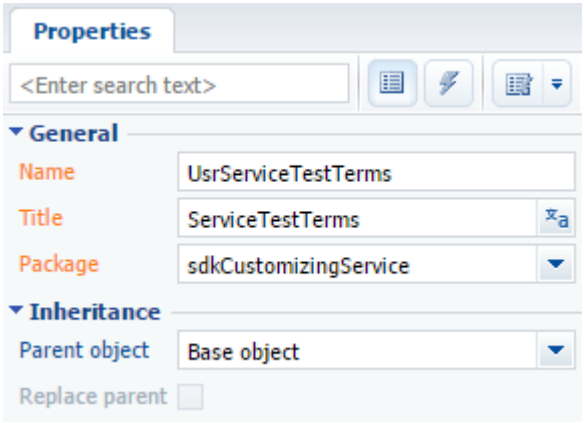
Fig. 1. Adding the schema



Set the following properties for the created object schema (Fig. 2):

- [Name] – "UsrServiceTestTerms"
- [Title] – "ServiceTestTerms"
- [Parent object] – the [Base object] schema

Fig. 2. Properties of the added object schema



In the created schema, create a number of columns, whose primary properties are listed in table 1.

Table 1. Properties of the added columns

Name	Title	Type	Description
UsrReactionTimeUnit	Response time unit	The [Time unit] lookup	Specifies the time unit (calendar days, hours, etc.) that will be used for calculating the [Response time] parameter.
UsrReactionTimeValue	Response time value	Integer	A column for storage the response time value.
UsrSolutionTimeUnit	Response time unit	The [Time unit] lookup	Specifies the time unit (calendar days, hours, etc.) that will be used for calculating the [Response time] parameter.
UsrSolutionTimeValue	Resolution time	Integer	A column for storage the response time value.
UsrCalendarId	Calendar that is used	The [Calendar] lookup	The calendar used for calculating the case deadline.
UsrServicePactId	Service agreement	The [Service agreement] lookup	Link to the [Service agreement] object. Added for enabling filtration.
UsrServiceItemId	Service	The [Service] lookup	Link to the [Service] object. Added for enabling filtration.

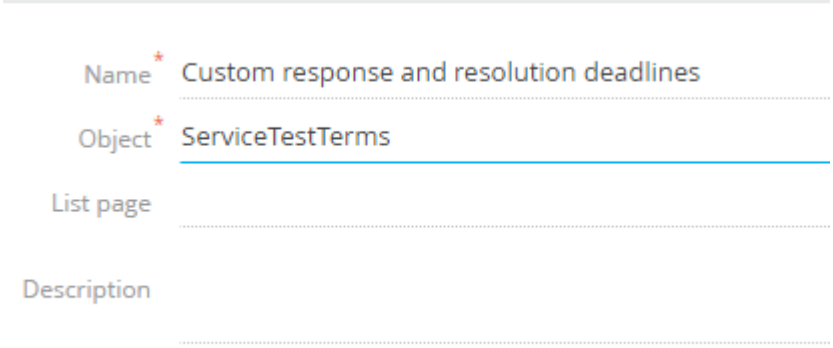
Publish the schema after adding the columns.

2. Adding a lookup and populating it with values needed to calculate the deadline parameters

Provide specific values to calculate the case response and resolution deadline. To do this, add a lookup with the following values based on the added schema (fig.3):

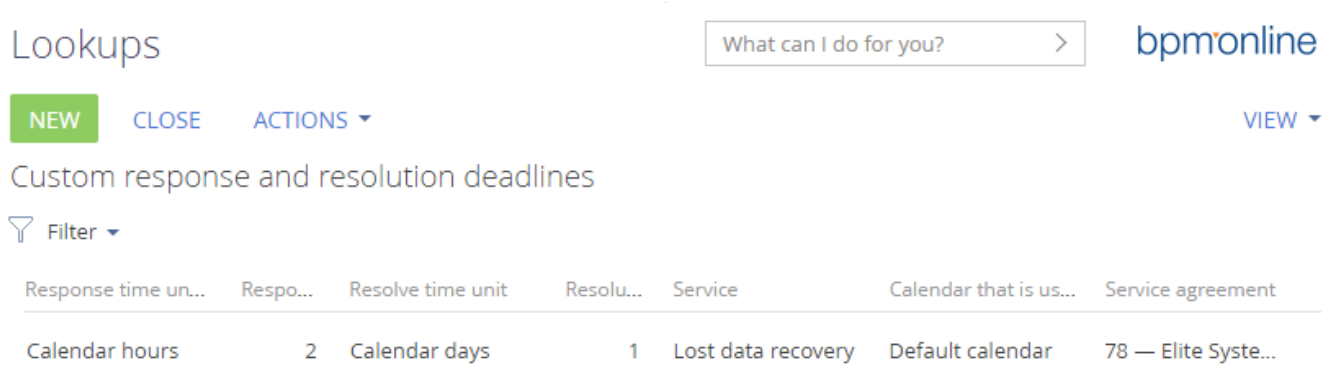
- [Name] – "Custom response and resolution deadlines"
- [Object] – ServiceTestTerms

Fig. 3. Properties of the added lookup



Add a record with the following data to the added lookup (as per the case conditions) (fig.4):

Fig. 4. A record in the created lookup that meets the case conditions



3. Implementing a class with the mechanism of receiving deadline parameters

Add the source code schema (fig.1, 2) Add the class inherited from the *BaseTermStrategy* abstract class (declared in the *Calendar* package) to the schema source code. Implement a parameterized constructor with the following parameters in the class:

- *UserConnection userConnection* – user current connection
- *Dictionary<string, object> args* – arguments that are the base of performing calculation

Implement the *GetTermInterval()* abstract method declared in the base class. This method accepts the mask of populated values as the incoming parameter, which is the base of taking a decision about populating the specific deadline parameters of the *TermInterval* returned class implementing the *ITermInterval<TMask>* interface.

The complete schema source code:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
    using Terrasoft.Common;
    using Terrasoft.Configuration.Calendars;
    using Terrasoft.Core;
    using Terrasoft.Core.Entities;
    using CalendarsTimeUnit = Calendars.TimeUnit;
    using SystemSettings = Terrasoft.Core.Configuration.SysSettings;
    public class ServiceTestTermsStrategy: BaseTermStrategy<CaseTermInterval,
CaseTermStates>
    {
        // Container class for storage of data received from the entrance point.
        protected class StrategyData
        {
            public Guid ServiceItemId {
                get;
                set;
            }
            public Guid ServicePactId {
                get;
                set;
            }
        }
        // The field for storage of data received from the entrance point.
        protected StrategyData _strategyData;
        // Parameterized constructor necessary for the correct
        // initialization by selector class.
        public ServiceTestTermsStrategy(UserConnection userConnection,
Dictionary<string, object> args)
            : base(userConnection) {
            _strategyData = args.ToObject<StrategyData>();
        }
```

```
        // Method that receives data and returns them in the CaseTermInterval class
instance.
        public override CaseTermInterval GetTermInterval(CaseTermStates mask) {
            var result = new CaseTermInterval();
            // Creating the EntitySchemaQuery query.
            var esq = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"UsrServiceTestTerms");
            // Adding columns to the query.
            string reactionTimeUnitColumnName =
esq.AddColumn("UsrReactionTimeUnit.Code").Name;
            string reactionTimeValueColumnName =
esq.AddColumn("UsrReactionTimeValue").Name;
            string solutionTimeUnitColumnName =
esq.AddColumn("UsrSolutionTimeUnit.Code").Name;
            string solutionTimeValueColumnName =
esq.AddColumn("UsrSolutionTimeValue").Name;
            string calendarColumnName = esq.AddColumn("UsrCalendarId.Id").Name;
            // Adding filters to the query.
            esq.CreateFilterWithParameters(FilterComparisonType.Equal,
"UsrServiceItemId", _strategyData.ServiceItemId);
            esq.CreateFilterWithParameters(FilterComparisonType.Equal,
"UsrServicePactId", _strategyData.ServicePactId);
            // Execution and processing of query results.
            EntityCollection entityCollection =
esq.GetEntityCollection(UserConnection);
            if (entityCollection.IsNotEmpty()) {
                // Adding response time to the nurtured value.
                if (!mask.HasFlag(CaseTermStates.ContainsResponse)) {
                    result.ResponseTerm = new TimeTerm {
                        Type =
entityCollection[0].GetTypedColumnValue<CalendarsTimeUnit>
(reactionTimeUnitColumnName),
                        Value = entityCollection[0].GetTypedColumnValue<int>
(reactionTimeValueColumnName),
                        CalendarId = entityCollection[0].GetTypedColumnValue<Guid>
(calendarColumnName)
                    };
                }
                // Adding resolution time to the nurtured value.
                if (!mask.HasFlag(CaseTermStates.ContainsResolve)) {
                    result.ResolveTerm = new TimeTerm {
                        Type =
entityCollection[0].GetTypedColumnValue<CalendarsTimeUnit>
(solutionTimeUnitColumnName),
                        Value = entityCollection[0].GetTypedColumnValue<int>
(solutionTimeValueColumnName),
                        CalendarId = entityCollection[0].GetTypedColumnValue<Guid>
(calendarColumnName)
                    };
                }
            }
            return result;
        }
    }
}
```

Publish the schema after adding the source code.

4. Adding the new rule

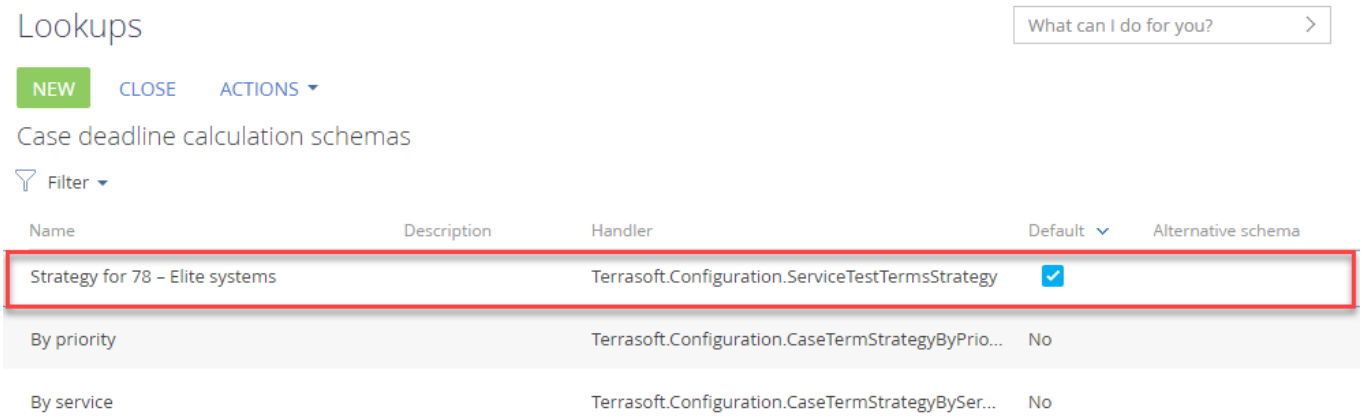
Add a value to the [Case deadline calculation schemas] lookup. In the [Handler] column, specify the full qualified name of the created class (specifying the namespaces).

In the [Alternative schema] column you may specify the rule for calculating the deadline in case calculation by current rule is not possible. Take into considerations that if any of deadline parameters is not calculated by strategy class, a class instance of an alternative strategy will be created. In case the alternative strategy cannot calculate the deadline either, another alternative strategy will be created, thus forming a rule queue.

Select the [Default] checkbox for the added record.

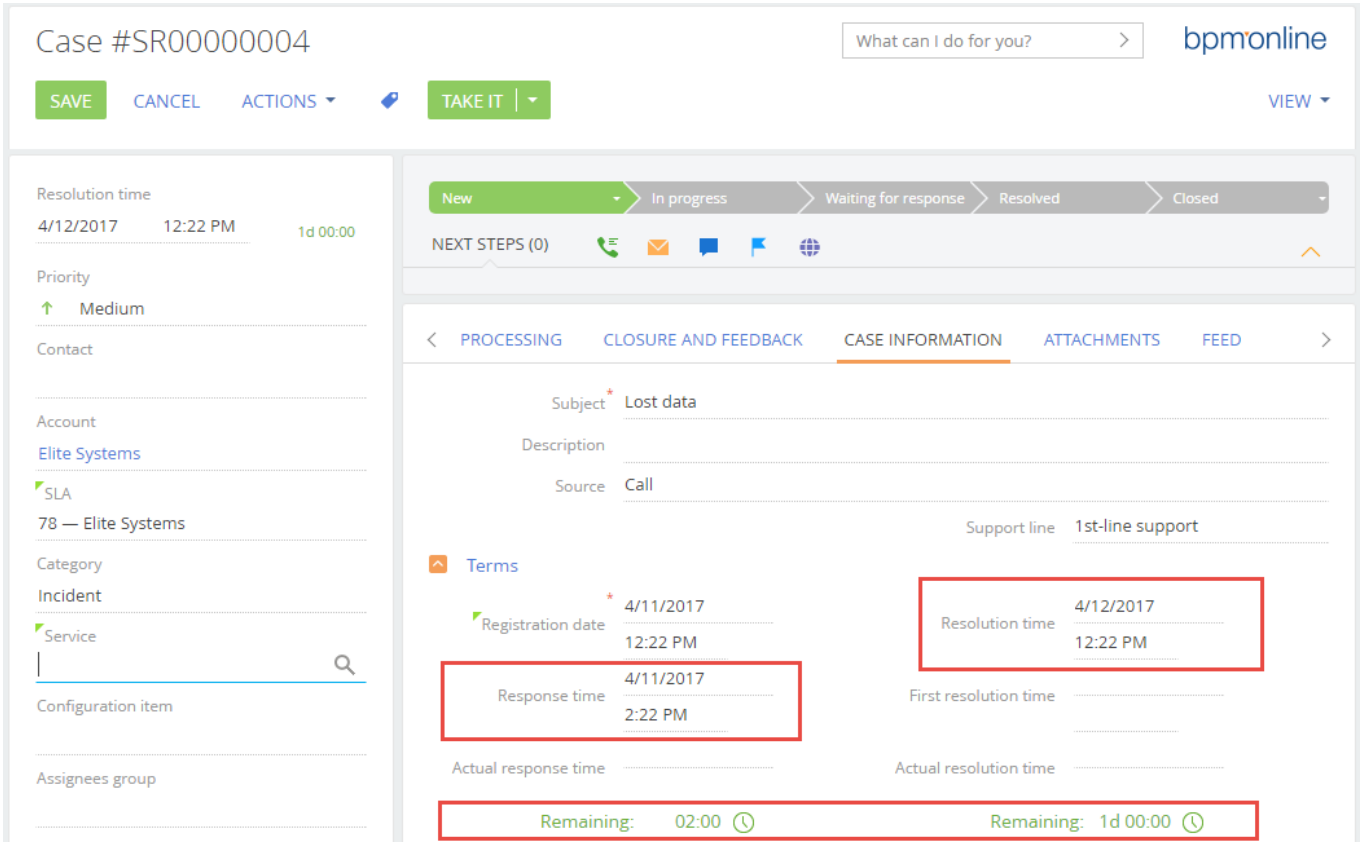
See an example of an added record to the [Case deadline calculation schemas] lookup in fig.5.

Fig. 5. A record of a custom deadline calculation rule



As a result, new response and resolution deadline calculation rules will be applied for cases per the [78 — Elite Systems] agreement for the [Lost data recovery] service.

Fig. 6. Case result



Adding a macro handler in email templates

Difficulty level

Introduction

Email templates are pre-formatted and/or pre-written email messages. For example, customer service specialists often use email templates to automate routine communications with the customers. Bpm’online stores email templates in the [Email templates] lookup (e.g. the “Case resolution notification” email template is used to notify customers that their case has been resolved). Learn more about email templates in the Automatic emailing setup article.

Pre-configured macros are used in email message templates to fill them with certain object column values (e.g. to personalize emails with titles or phone numbers of contacts).

Bpm'online enables you to implement a custom logic of populating values, which returns a macro handler. In this case, the system executes the algorithm implemented by the developer instead of the base logic during the processing of macros.

The *@Invoke* tag points to a specialized class handler. Specify the class name of the *IMacrosInvokable* interface that includes the *GetMacrosValue()* method, separated by dots. This method must return a string that will substitute the macro string.

To implement a custom macro handler:

1. Create a class that implements the *IMacrosInvokable* interface.
2. Register a macro in the *EmailTemplateMacros* table, specifying its *ParentId* (the base template with the @Invoke tag) and *ColumnPath* (the class name).
3. Add a macro to the email template.

Case description

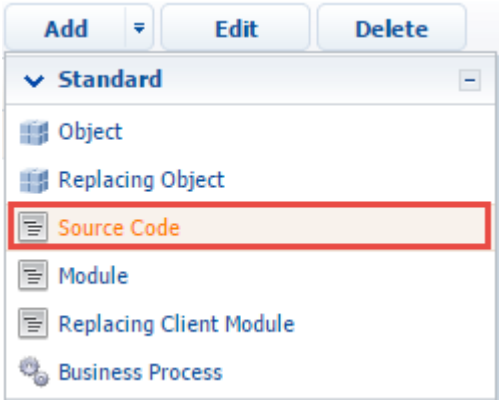
Add an email template macro handler that will return the “Test” string.

Case implementation algorithm

1. Creating a class that implements the IMacrosInvokable interface

To create a new class that implements the *IMacrosInvokable* interface, add the [Source Code] schema to the package used for development. To do this, go to the **[Configuration] section** of the system designer, select a custom package and on the [Schemas] tab, execute the menu command [Add] -> [User Task] (Fig.1). As a result, the **source code designer** window will open for further schema configuration.

Fig. 1. Adding a new [Source code] schema



Populate the following required values for the created object schema:

- [Title] – “Text string generator”.

- [Name] – “UsrVwContactAdress”.

Add the following source code on the [Source Code] tab of the schema designer:

```
namespace Terrasoft.Configuration
{
    using System;
    using Terrasoft.Core;
    // The class of the email template macro handler.
    public class UsrTestStringGenerator : IMacrosInvokable
    {
        // User connection.
        public UserConnection UserConnection {
            get;
            set;
        }
        // The method that returns the substitution value.
        public string GetMacrosValue(object arguments) {
            return "Test";
        }
    }
}
```

Publish the created schema.

2. Registering a macro in the EmailTemplateMacros table

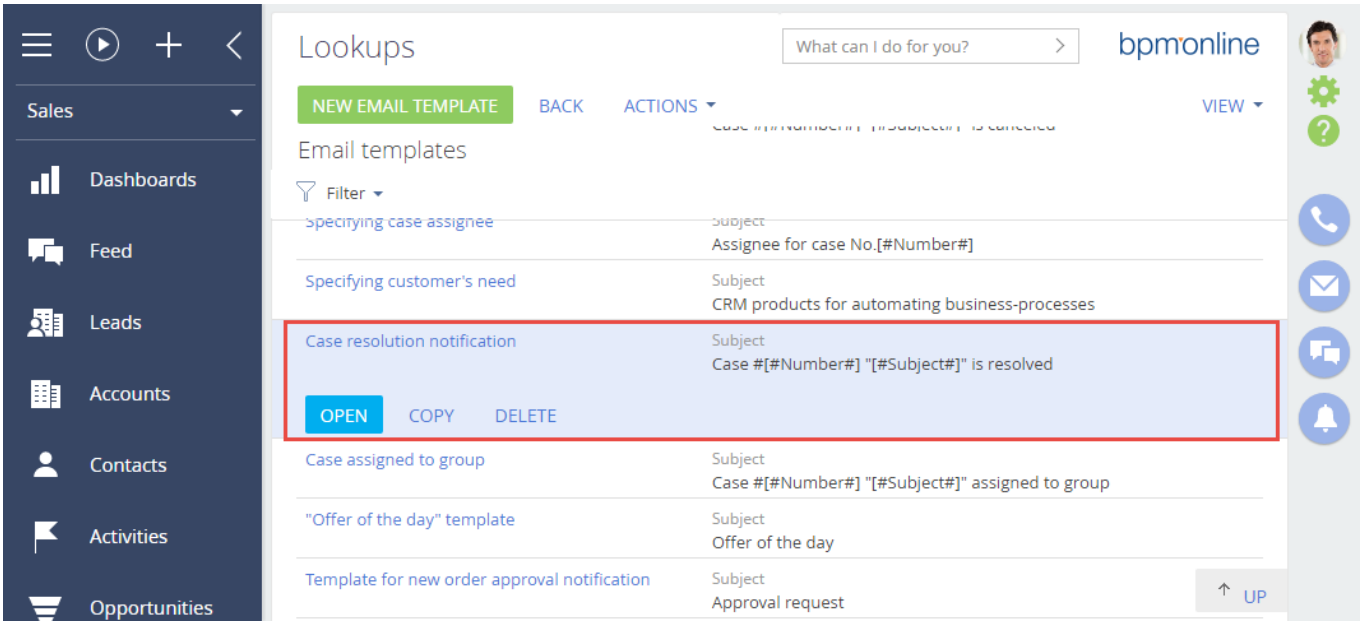
To register a macro in the *EmailTemplateMacros* table, execute the following SQL query:

```
INSERT INTO EmailTemplateMacros(Name, Parentid, ColumnPath)
VALUES (
    'UsrTestStringGenerator',
    (SELECT TOP 1 Id
    FROM EmailTemplateMacros
    WHERE Name = '@Invoke'),
    'Terrasoft.Configuration.UsrTestStringGenerator'
)
```

3. Adding a macro handler in the email template

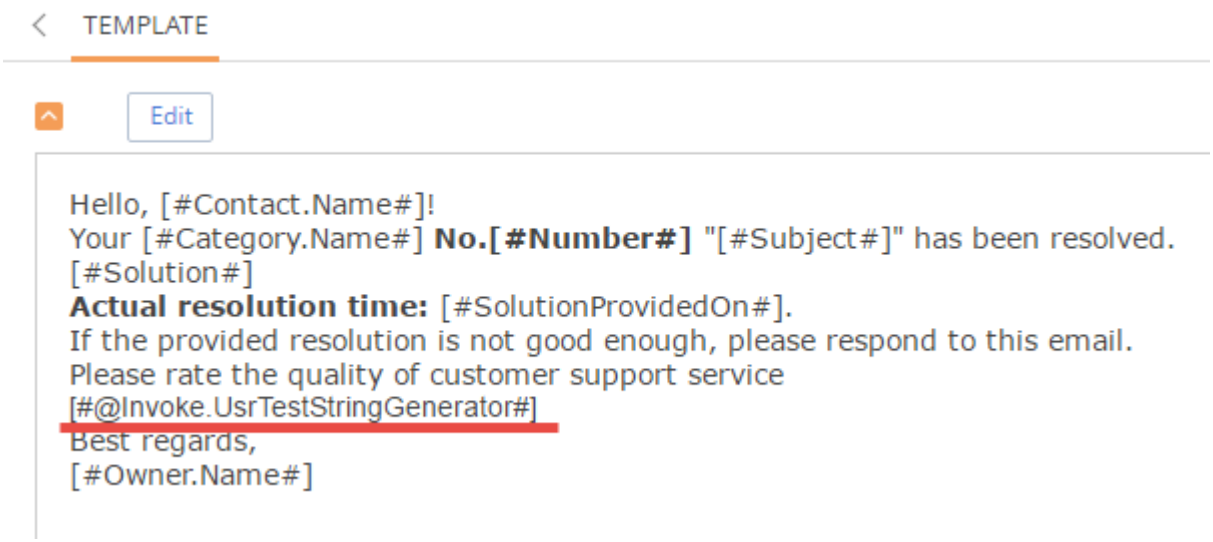
After registering the macro, you can start using it in email templates. To do this, you must change one or more records in the [Email templates] lookup (Fig. 2).

Fig. 2. The [Email templates] lookup

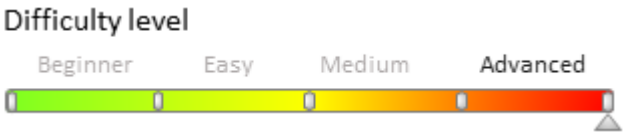


For example, modify the [Case resolution notification] record to edit the email used to notify customers that their case has been resolved. If you add the [#@Invoke.UsrTestStringGenerator#] macro to the template (Fig. 3), the "Test” value will substitute the macro when the email is sent to the client.

Fig. 3. A macro in the email template



Creating Web-to-Case landing pages



Introduction

Web-to-Case functionality implements the ability to create cases in the bpm'online by filling out a web form fields embedded in a landing page on a third-party website.

Web-to-Case landing record can be configured in the system interface in the [Landing pages and web forms] section. To add the JavaScript code (generated by bpm’online for each landing record) to a third-party site, you need the basic Web development skills.

More information about landings can be found in the [Landing pages and web forms] section articles of the corresponding products (such as bpm'online marketing). More information about the Web-to-Case functionality can be found in the "**Web-to-Case**” article.

To create a Web-to-Case landing page:

1. Create new landing record in the bpm'online.

2. Create new landing page that will contain the code that binds landing form (on the website) and the landing record (in bpm’online).

3. Add the landing page to the website.

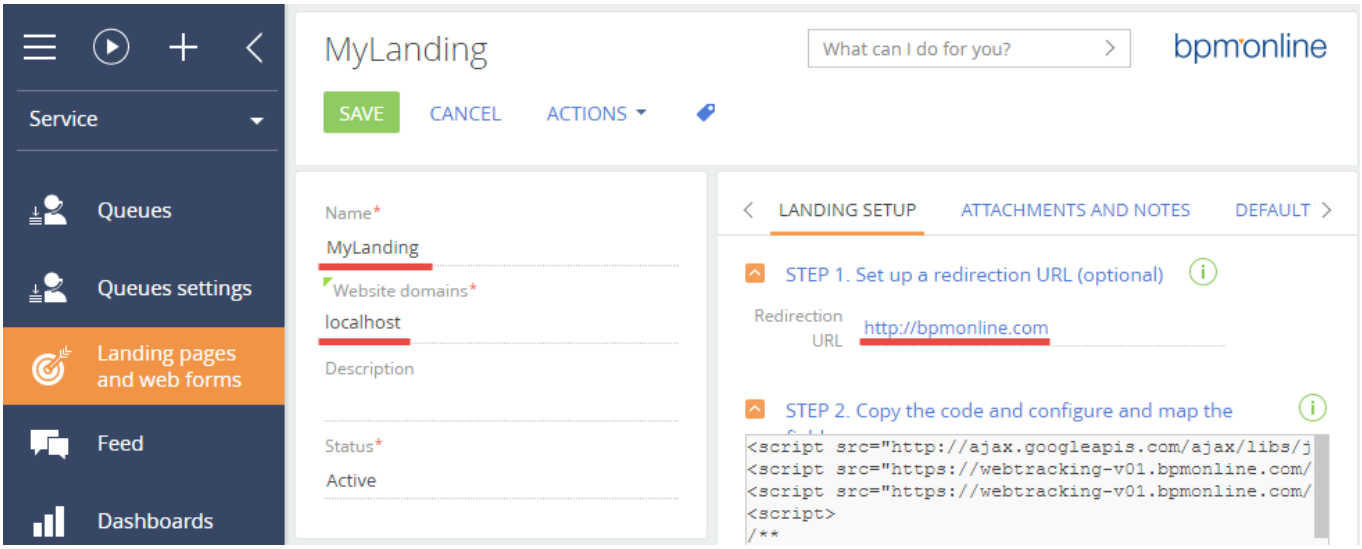
Steps to create Web-to-Case landing

1. Create new landing record in the bpm'online

To create a new landing record, execute the [Add] action in the [Landing pages and web forms] section. Fill in the following fields on the opened page (Fig. 1):

- [Name] – landing page name in bpm'online.
- [Website domains] – your landing page URL.
- [Status] – landing status.
- [Redirection URL] – the URL that is opened after the landing page form is completed.

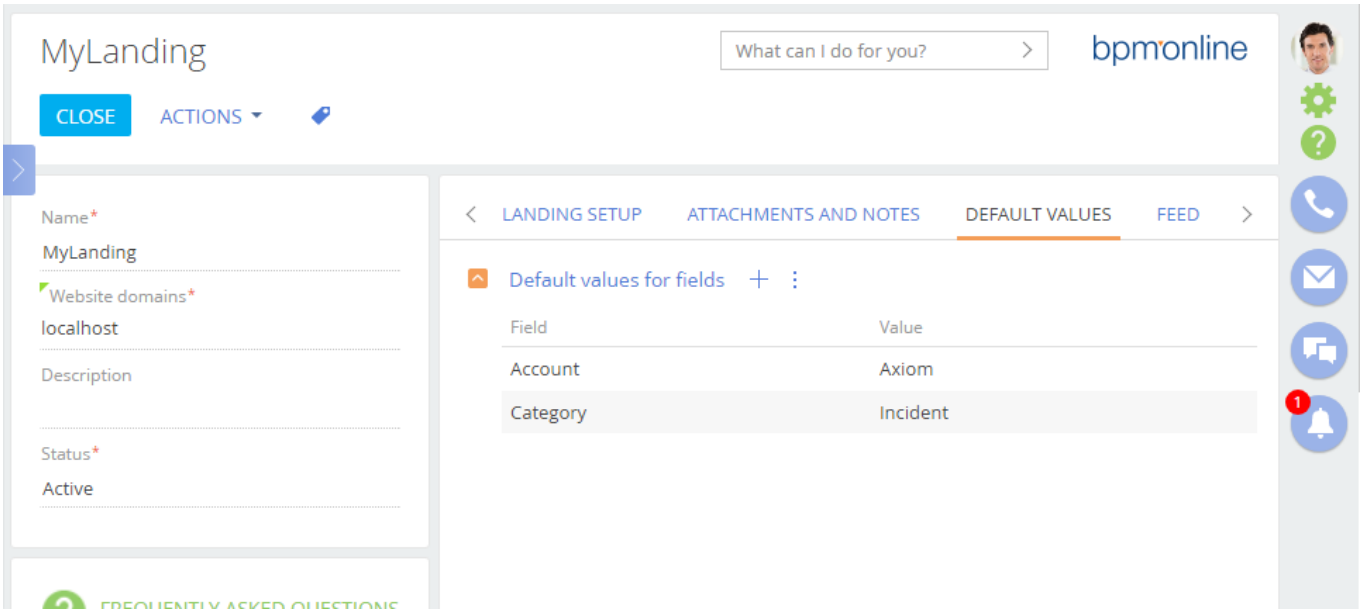
Fig. 1 Landing edit page



Attention!

When creating a case, you can receive only four fields ("Subject, "Email", "Name" and "Phone") from the landing page. Therefore, you must set the default values for the new landing record(Fig. 2).

Fig. 2 Values by default



Save the page to apply the changes.

2. Create a landing page

To create landing page, you need to create a standard HTML page containing a Web form in any text editor using HTML markup.

To register the data sent via the web-form, add four fields to the form (using <input> element) that define the case:

- Case subject
- Contact email
- Contact name
- Contact phone

Specify the *name* and *id* attributes for each field.

To send a form data to bpm'online when creating a new [Case] object, you need to add a JavaScript script to the HTML page. Copy the script source code from the [STEP 2. Copy the code and configure and map the fields] field of the landing edit page (Fig. 1).

> NOTE
>
> The script must be copied from the already saved landing.

The script contains the *config* configuration object that has following properties:

- *fields* – contains the object with "Subject, "Email", "Name" and "Phone” values that must match the *id* attribute selectors of the corresponding web form fields.
- *landingId* – contains the landing Id in the database.
- *serviceUrl* – contains URL of the service to which the form data will be sent.
- *redirectUrl* – contains redirection URL specified in the [Redirection URL] field of the landing.
- *onSuccess* – contains a function that handles the successful creation of a case. Optional property.
- *onError* – contains a function that handles the error of the case creation. Optional property.

The *config* configuration object is passed as an argument of the *createObject()* function that must be executed when the form is submitted.

To call the *createObject()* function when sending a form, add the *onSubmit = "createObject(); return false"* attribute to the *<form>* tag of the HTML page of the Landing page (see STEP 3, Fig. 1).

An example of the complete landing page source code for the case registration:

```
<!DOCTYPE html>
<html>
```

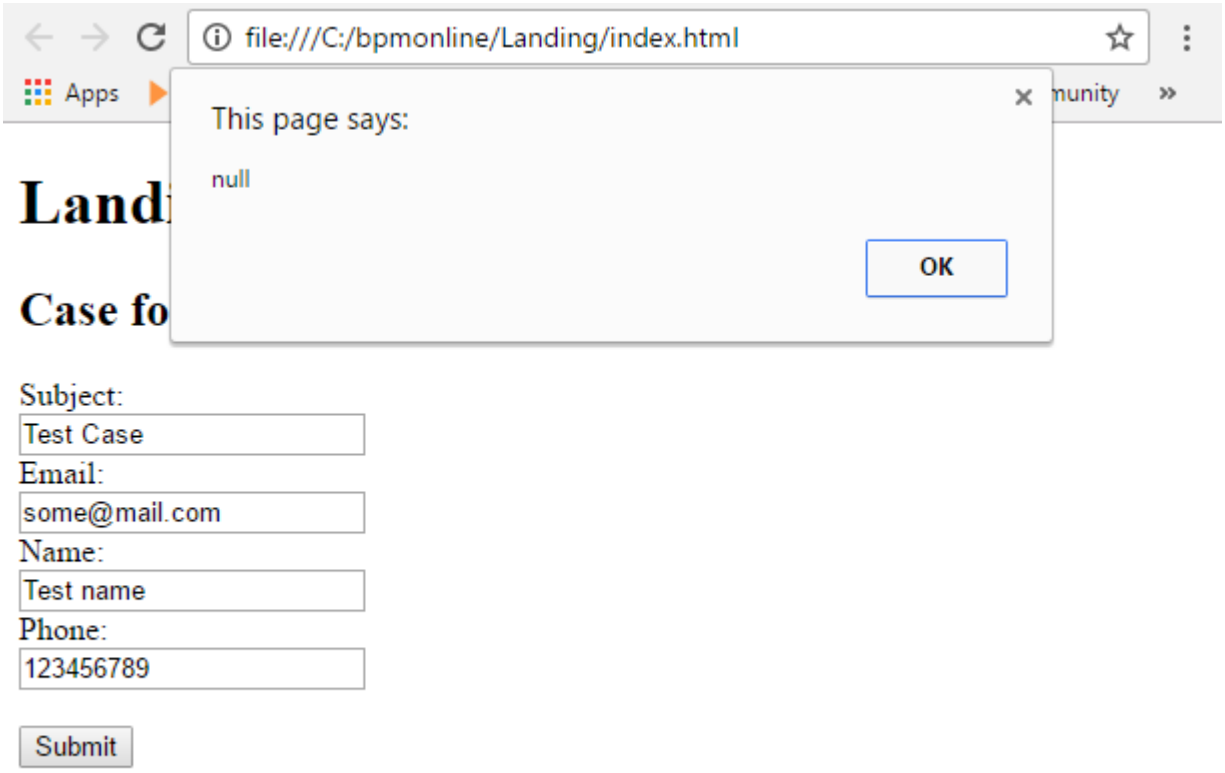
```
<head>
    <meta charset="UTF-8">
    <!--STEP 2-->
    <!--This part needs to be copied from the STEP 2 field of the lending edit page--
>
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js">
</script>
    <script src="https://webtracking-v01.bpmonline.com/JS/track-cookies.js"></script>
    <script src="https://webtracking-v01.bpmonline.com/JS/create-object.js"></script>
    <script>
        /**
         * Replace the "css-selector" placeholders in the code below with the element
selectors on your landing page.
         * You can use #id or any other CSS selector that will define the input field
explicitly.
         * Example: "Email": "#MyEmailField".
         * If you don't have a field from the list below placed on your landing,
leave the placeholder or remove the line.
         */
        var config = {
            fields: {
                "Subject": "#subject-field", // Case subject
                "Email": "#email-field", // Visitor's email
                "Name": "#name-field", // Visitor's name code
                "Phone": "#phone-field", // Visitor's phone number
            },
            landingId: "8ab71187-0428-4372-b81c-fd05b141a2e7",
            serviceUrl:
"http://localhost/bpmonlineservice710/0/ServiceModel/GeneratedObjectWebFormService.sv
c/SaveWebFormObjectData",
            redirectUrl: "http://bpmonline.com",
            onSuccess: function(response) {
                window.alert(response.resultMessage);
            },
            onError: function(response) {
                window.alert(response.resultMessage);
            }

        };
        /**
         * The function below creates a object from the submitted data.
         * Bind this function call to the "onSubmit" event of the form or any other
elements events.
         * Example: <form class="mainForm" name="landingForm"
onSubmit="createObject(); return false">
         */
        function createObject() {
            landing.createObjectFromLanding(config)
        }
    </script>
    <!--STEP 2-->
</head>
<body>
<h1>Landing web-page</h1>
<div>
    <h2>Case form</h2>
    <form class="mainForm" name="landingForm" onSubmit="createObject(); return
false">
        Subject:<br>
        <input type="text" name="subject" id="subject-field"><br>
        Email:<br>
        <input type="text" name="Email" id="email-field"><br>
```

```
        Name:<br>
        <input type="text" name="Name" id="name-field"><br>
        Phone:<br>
        <input type="text" name="Phone" id="phone-field"><br><br>
        <input type="submit" value="Submit">
        </font>
    </form>
</div>
</body>
</html>
```

3. Add the page to the website.

A case from the landing page will be added to the bpm'online only if the page is hosted on the site whose name is listed in the [Website domains] field of the landing page record in bpm’online. If you open the page in the browser locally, then an empty message will be displayed when the case is created.

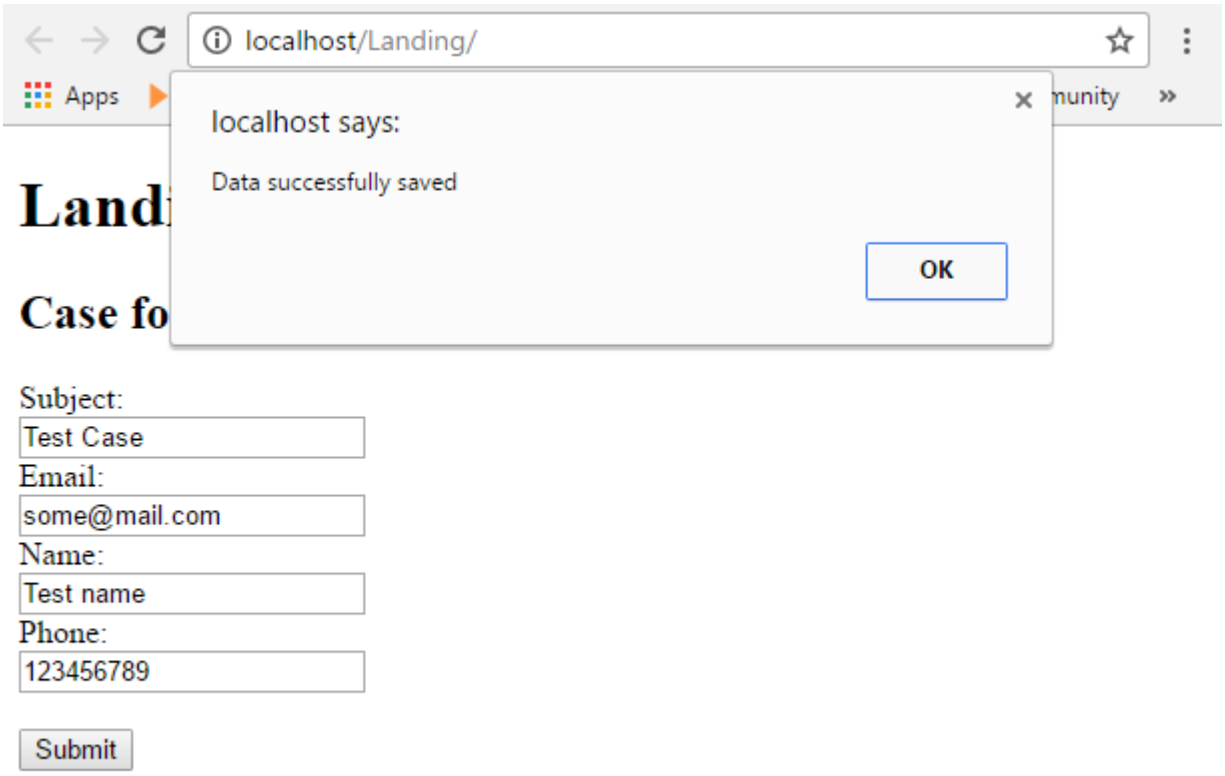
Fig. 3 Empty message



> NOTE
>
> The output of an empty message is configured in the *onError()* method of the configuration object.

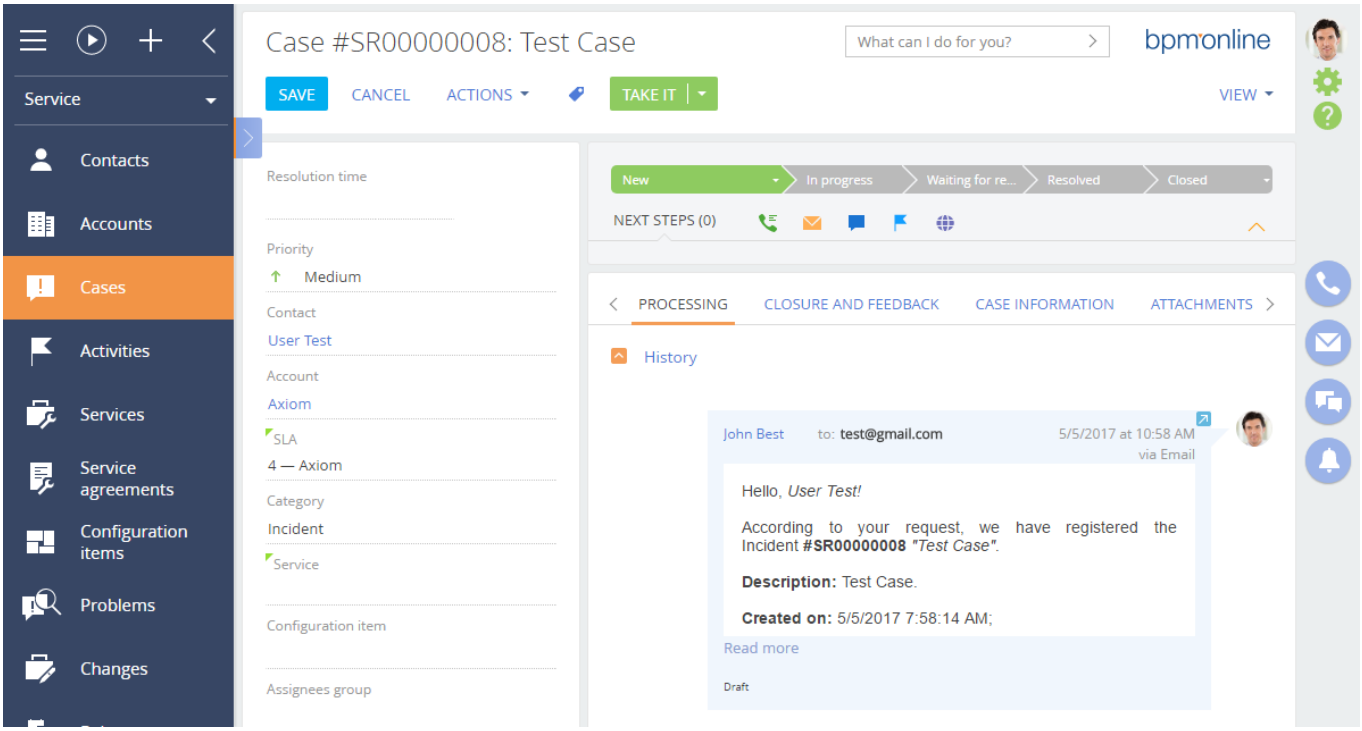
If you place the page on the local server of the computer that serves as the reserved domain name localhost (as specified in the landing setting , Fig. 1), then the script that adds the address from the web page of the landing will work correctly (Fig. 4)

Fig. 4 The correct adding of data

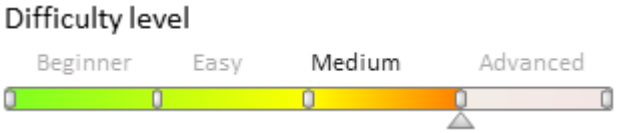


As a result, a case with specified parameters will be automatically created.

Fig. 5 Automatically created case

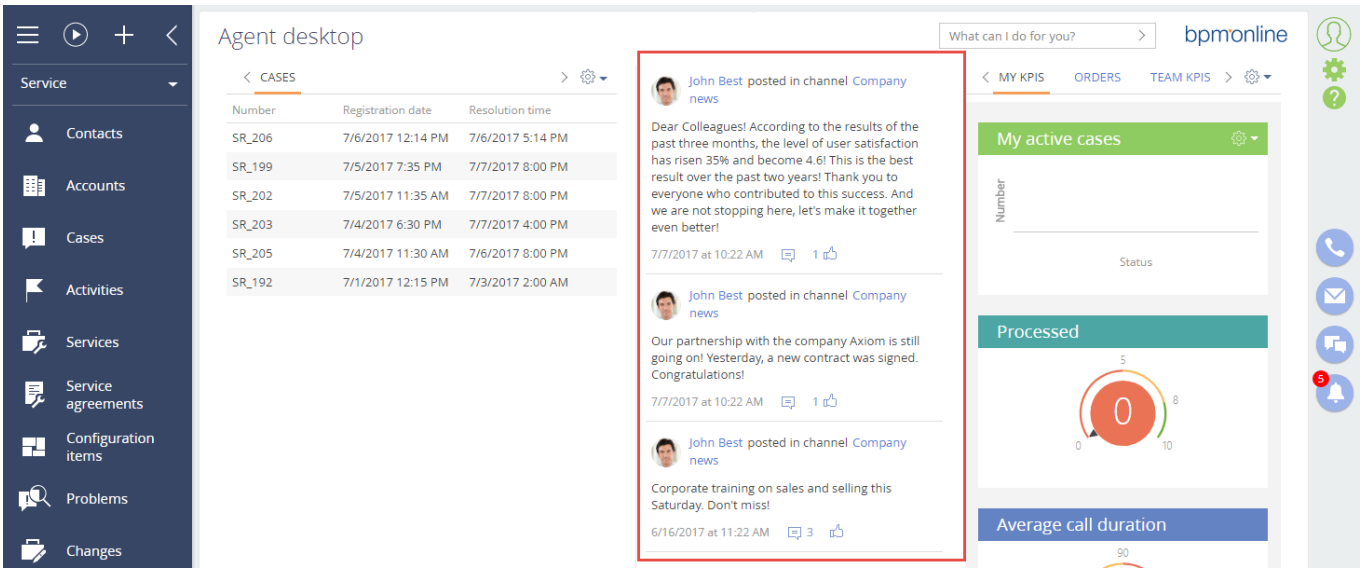


How to hide feed area in the agent desktop



The feed area of the [Agent desktop] section used to notify helpdesk or contact center agents about noteworthy events of the company (Fig. 1).

Fig. 1. Agent desktop feed area



To hide feed area:

1. Create the [Agent desktop page] replacing schema in the custom package. The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**" article.

2. Add the following source code to the schema:

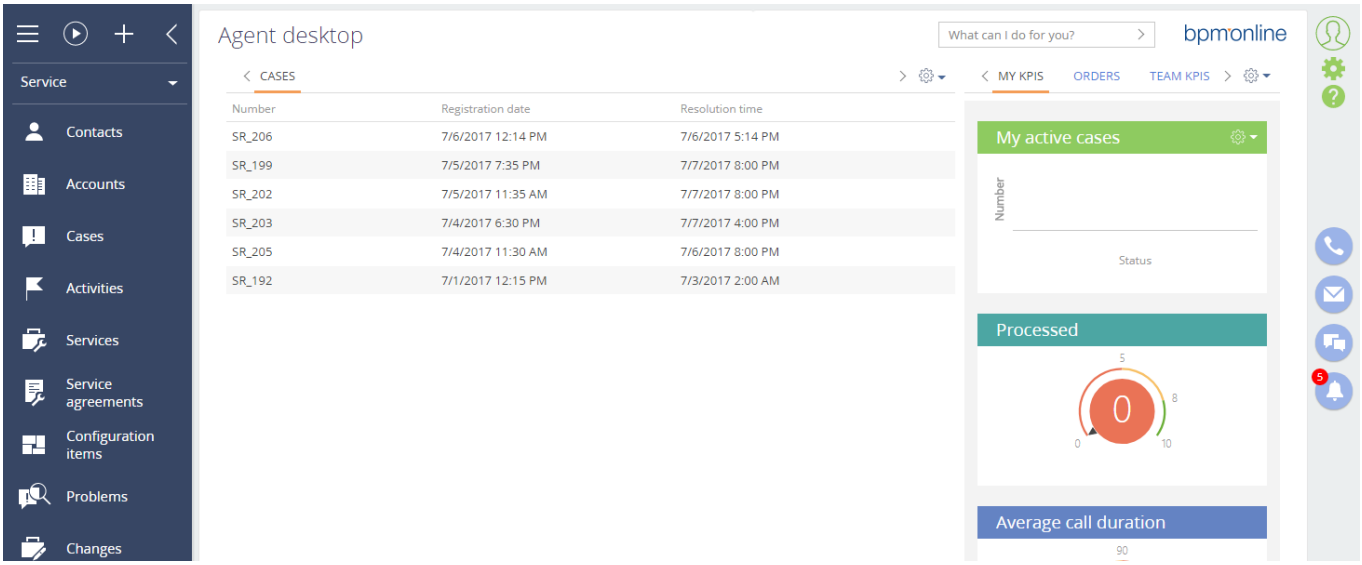
```
define("OperatorSingleWindowPage", [],
    function() {
        return {
            methods: {
                // Replacing the base method to exclude the ENSFeedModule feed module
from the loaded modules.
                loadContent: function() {
                    // the ESNFeedModule module does not need to be loaded because
The centerContainer container is removed
                    //this.loadModule("ESNFeedModule", "centerContainer");
                    this.loadModule("SectionDashboardsModule", "rightContainer");
                    this.loadModule("OperatorQueuesModule", "leftContainer");
                }
            },
            diff: /**SCHEMA_DIFF*/[
                {
                    "operation": "remove",
                    "name": "centerContainer"
                }
            ]/**SCHEMA_DIFF*/
        };
    }
);
```

3. Save the changes.

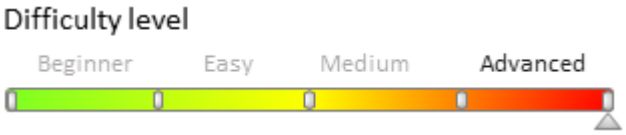
4. Refresh the browser page.

As a result the feed area in the agent desktop will be hidden (Fig. 2).

Fig. 2. [Agent desktop] section without feed area



Adding floating icons for internal case feed posts



Introduction

In bpm’online 7.12.0 you can quickly add a new case based on existing case communication email thread. Select a text in an email from the case message history and click the  button. The values of the source message fields whose [Make copy] checkbox is selected in the section wizard will be copied to the new case. Bpm’online will automatically reply to the email with a standard case registration notification. Adding cases based on an email text works both for emails and portal posts.

This function was implemented via the *SelectionHandlerMultiLineLabel* control element located in the *Message* package.

To process the [Processing] tab posts added to the internal feed:

1. Create the SocialMessageHistoryItemPage replacing schema.

2. Implement the selected text processing logic via the *selectedTextChanged* and *selectedTextHandlerButtonClick* events of the *SelectionHandlerMultiLineLabel* control element.

3. Add a configuration object with the *SelectionHandlerMultiLineLabel* element settings to the diff array.

Case description

Add the floating icon function when selecting a text from the internal case feed posts on the [Processing] tab of the [Cases] section. A new case with automatically populated fields should be added upon clicking the floating icon.

Source code

You can download the package with case implementation using the following link.

> ⚠ ATTENTION
>
> You can set up the package only for the *Service* line products or for product lines containing the *Message* and

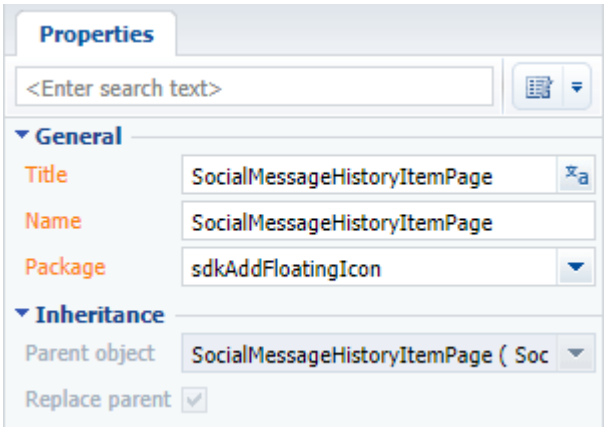
SocialMessage packages.

Case implementation algorithm

1. Create the SocialMessageHistoryItemPage replacing schema.

Create a replacing client module and specify the *SocialMessageHistoryItemPage* as parent object (Fig. 1). Creating a replacing page is covered in the"**Creating a custom client module schema**" article.

Fig. 1. Properties of the replacing module



2. Implement the selected text processing logic

Add the following methods to the created schema method collection (the source code is provided below):

- *getMessageFromHistory()* – an overridden base method. Receives the selected post subject.
- *onSelectedTextChanged()* – sets the selected text value to the *HighlightedHistoryMessage* attribute. Triggered upon text selection.
- *onSelectedTextButtonClick()* – adds a case whose subject is received from the previously installed *HighlightedHistoryMessage* attribute. The logic of adding a case is defined in the BaseMessageHistory parent schema. Triggered upon clicking the floating icon.

3. Set up the SelectionHandlerMultiLineLabel element.

Add the configuration object with the *SelectionHandlerMultiLineLabel* element settings to the *diff* array of the created schema. The replacing schema source code is as follows:

```
define("SocialMessageHistoryItemPage", ["SocialMessageConstants",
"css!SocialMessageHistoryItemStyle"],
    function(socialMessageConstants) {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "BaseMessageHistory",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            // Edit page view model methods.
            methods: {
                // Overridden base method. Receives subject for the selected post.
                getMessageFromHistory: function() {
                    var message = this.get("HighlightedHistoryMessage");
                    if (this.isHistoryMessageEmpty(message)) {
                        message = this.get("[Activity:Id:RecordId].Body");
                    }
                    return message;
                },
                // Text selection event handler.
                onSelectedTextChanged: function(text) {
                    this.set("HighlightedHistoryMessage", text);
```

```
                },
                // Floating icon clicking handler.
                onSelectedTextButtonClick: function() {
                    // Preparing case data from histroy.
                    this.prepareCaseDataFromHistory();
                }
            },
            diff: /**SCHEMA_DIFF*/[
                {
                    // Change the existing "MessageText" component.
                    "operation": "merge",
                    // Component name.
                    "name": "MessageText",
                    // Object properties.
                    "values": {
                        // View generator properties.
                        "generator": function() {
                            return {
                                // HTML-tag id value.
                                "id": "MessageText",
                                // Marker value.
                                "markerValue": "MessageText",
                                // Component class name.
                                "className":
"Terrasoft.SelectionHandlerMultilineLabel",
                                // CSS-style setup.
                                "classes": {
                                    "multilineLabelClass": ["messageText"]
                                },
                                // Caption.
                                "caption": {
                                    "bindTo": "Message"
                                },
                                "showLinks": true,
                                // Binding of the selected text modification event to
the handler method.
                                "selectedTextChanged": {"bindTo":
"onSelectedTextChanged"},
                                // Binding of the selected text floating icon
clicking event to the handler method.
                                "selectedTextHandlerButtonClick": {"bindTo":
"onSelectedTextButtonClick"},
                                // Floating icon display checkbox.
                                "showFloatButton": true
                            };
                        }
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
    }
);
```

A floating icon will be displayed on the [Processing] tab of the case page when you select a text from the internal case feed post upon your saving the schema and updating the page (Fig.2). A new case with automatically populated fields will be added upon clicking the floating icon (Fig.3).

Fig. 2. Floating icon upon selecting a text

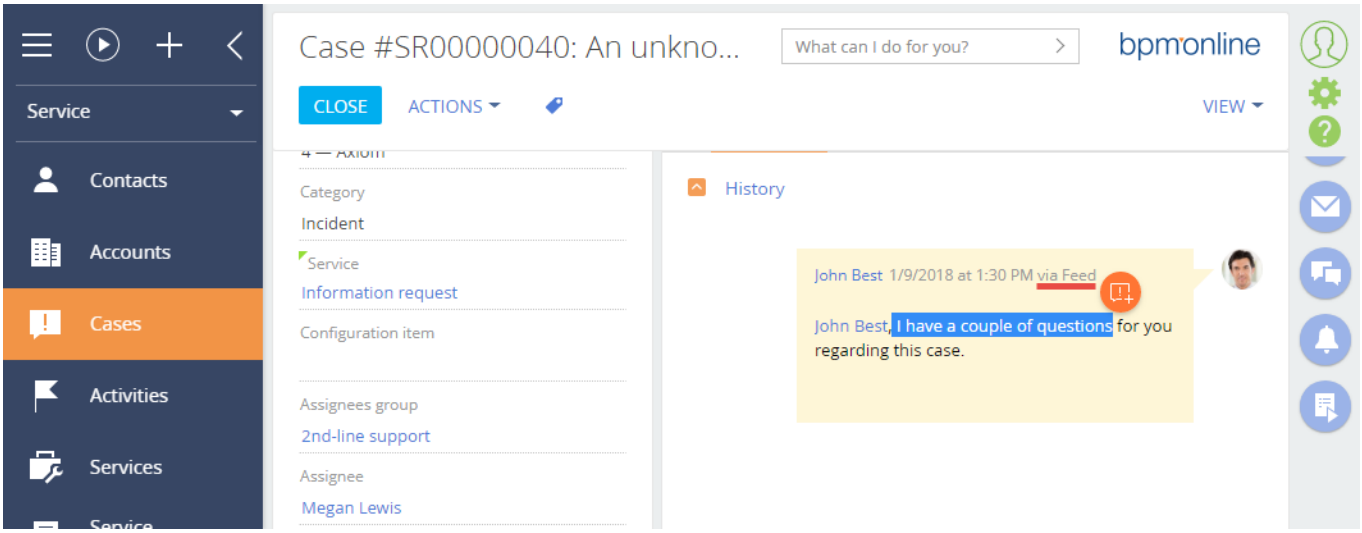
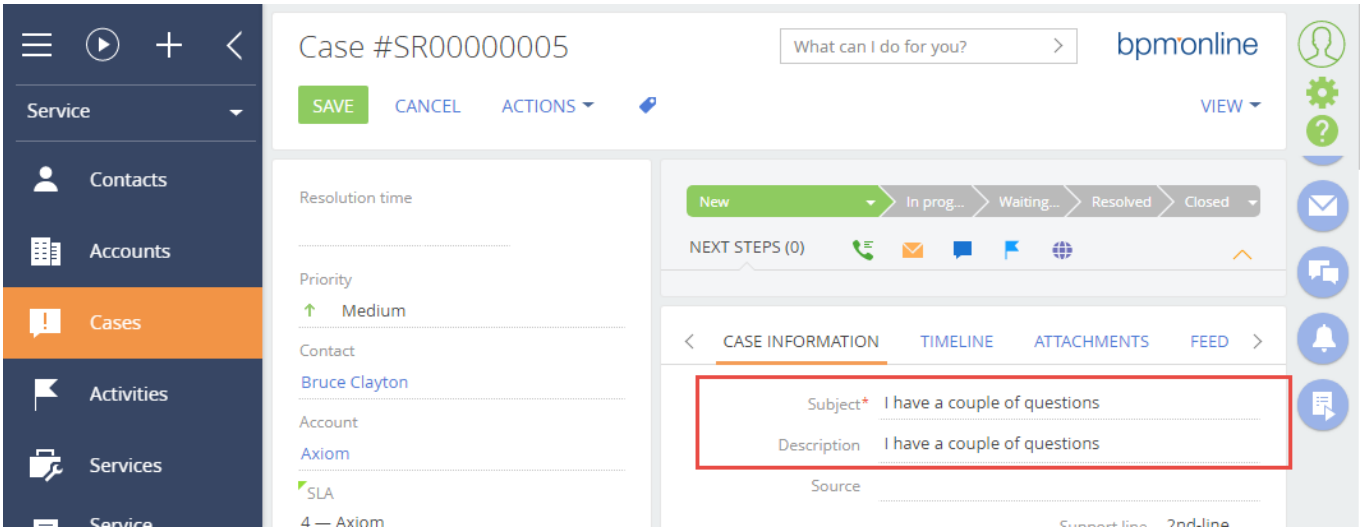


Fig. 3. New case

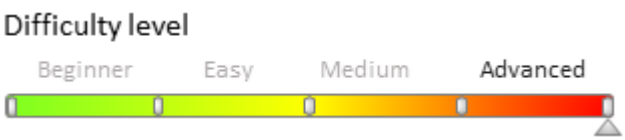


Lending product customization

Contents

- **How to create custom verification action page**
- **Using the EntityMapper schema**

How to create custom verification action page



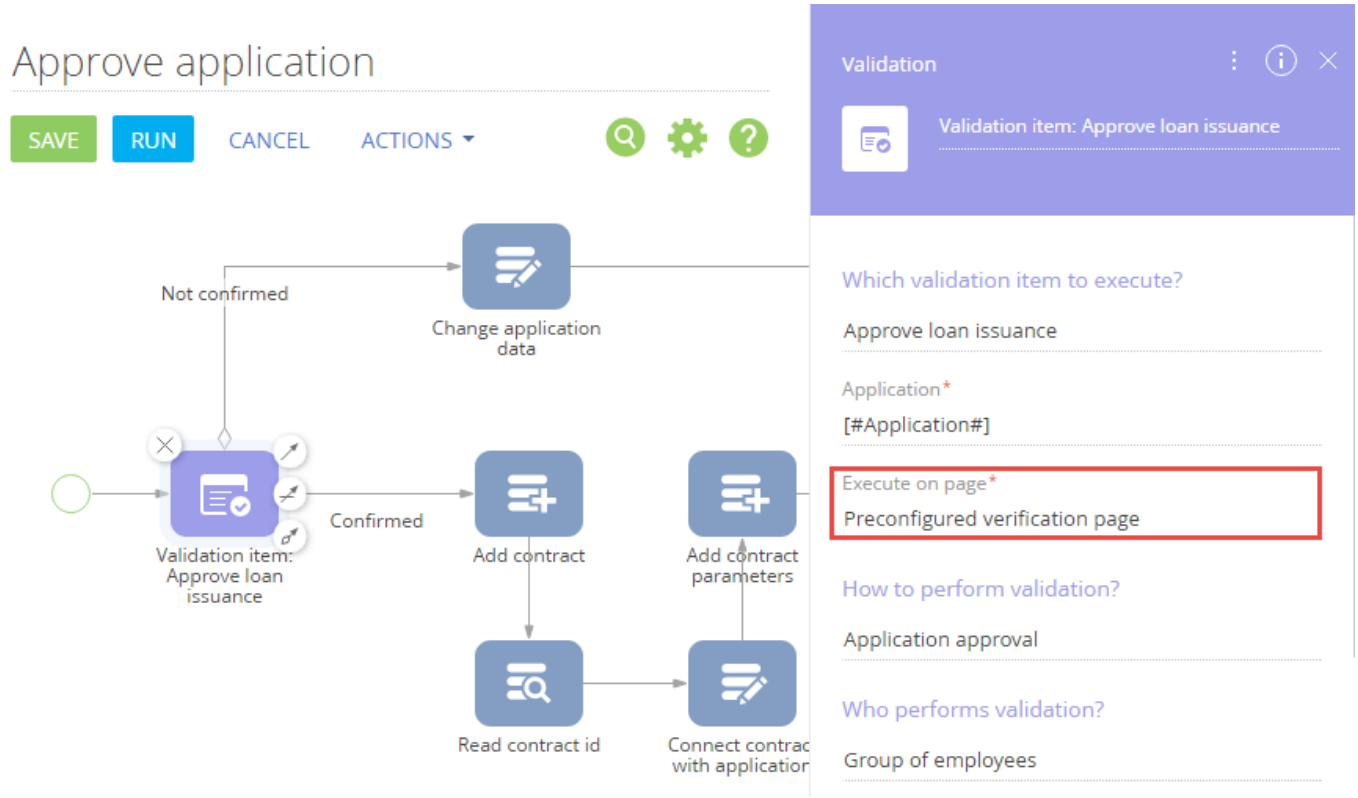
Introduction

Verification action is a confirmation that the data in the application form corresponds to the requirements of the application. The verification action is performed to check the data provided by the clients when they fill out their application forms.

When creating a custom verification action page, for example, in the *Approve application* business process, you can

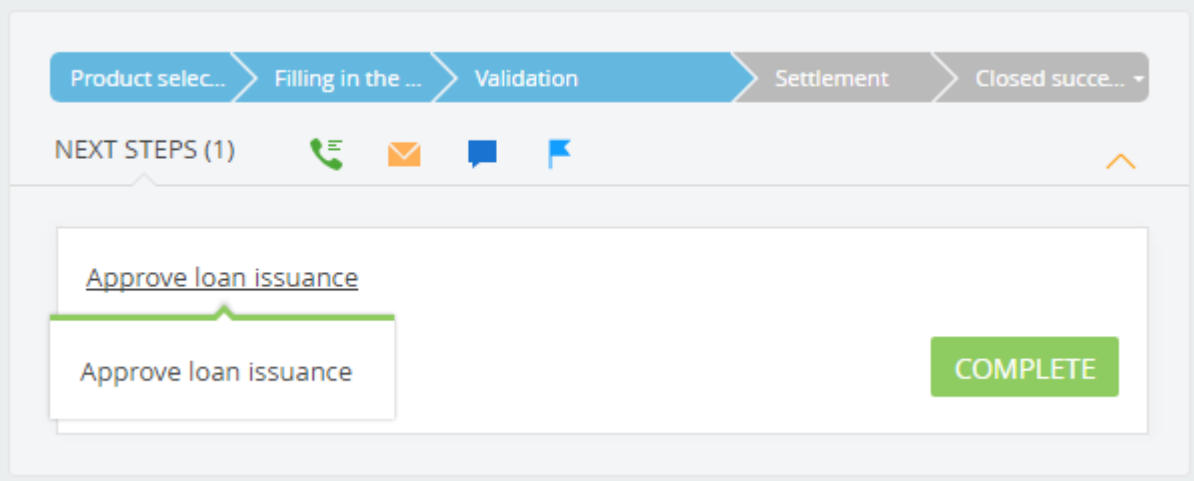
select the [Preconfigured verification page].

Fig. 1 Selecting the verification page



The page is displayed after clicking the [Complete] button of the [Approve loan issuance] activity that is created when the application is moved to the [Validation] stage (fig. 2).

Fig. 2 Activity on the [Validation] stage



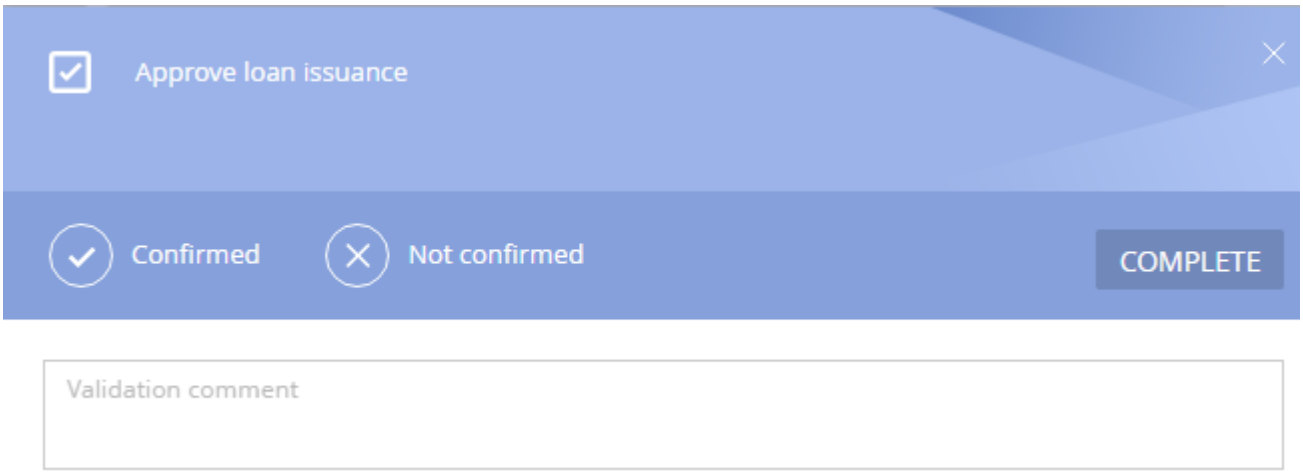
Preconfigured verification page contains (Fig. 3):

1. Buttons for selecting the result of the verification action.
2. The [Comment] field – comments to the verification action.
3. The [Conversation script] detail – contains a hints for the verifiers who call customers in the process of verification. Read only.
4. The [Attachments] detail – contains files and links attached to the validation stage. Read only.
5. The [Checklist] detail – contains control questions and answers to them.

Attention!

If a detail has no attached data, it will not be displayed to save page space.

Fig. 3 Verification page



You can create custom verification pages, inheriting them from the preconfigured page. To create a custom page:

1. Create a custom schema of the verification action page.
2. Use the schema created in the business process.

Case description

Create a verification action page where the [Comment] field is hidden.

Case implementation algorithm

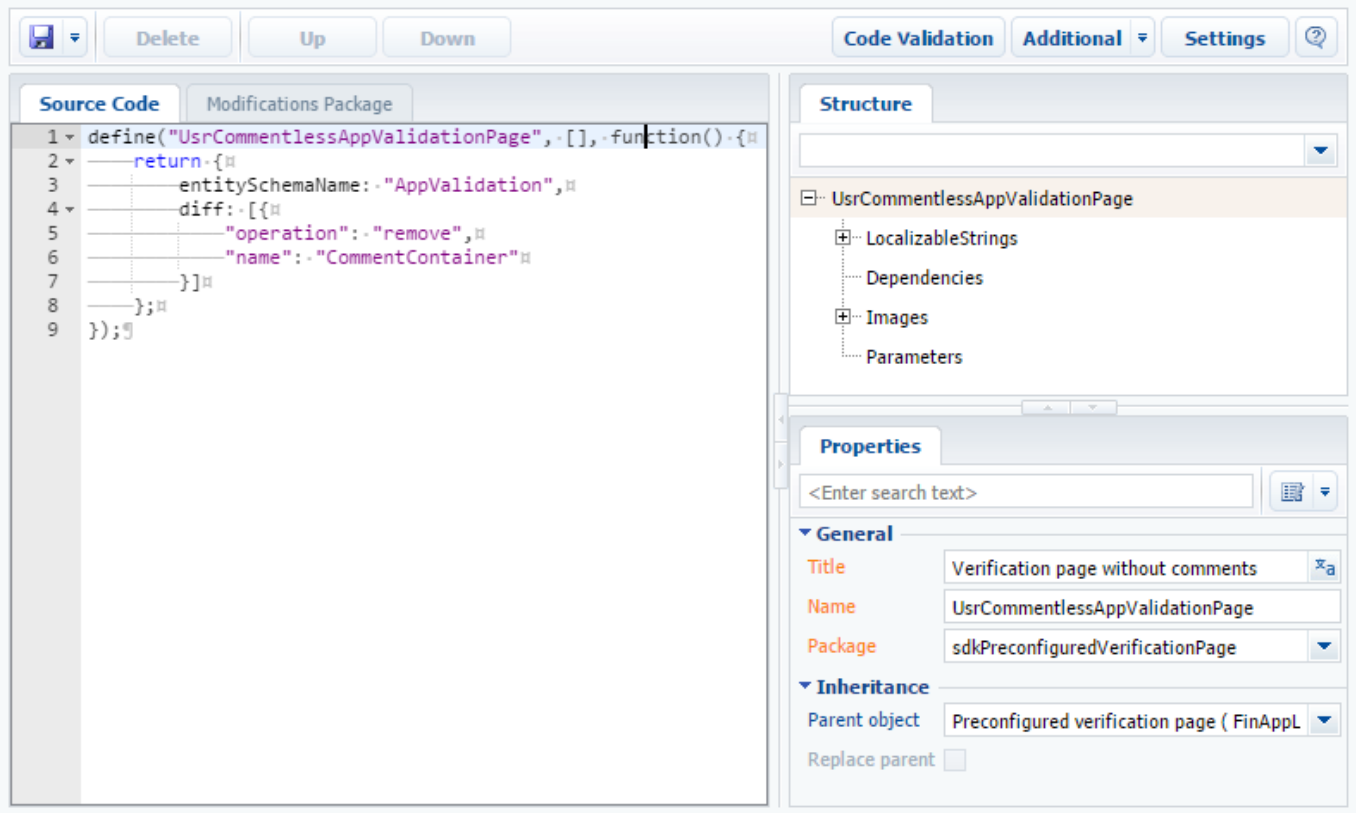
1. Create a schema of the verification action page

To do this, go to the [Configuration] section and select a custom package. Then execute the [Add] > [Schema of the Edit Page View Model] command. The process of creating custom schema of the view model is covered in the "**Creating a custom client module schema**” article.

You need to assign the following properties for the created schema (Fig. 4):

- [Title] – Verification page without comments.
- [Name] – UsrCommentlessAppValidationPage.
- [Package] – Custom (or another custom package).
- [Parent object] – Preconfigured verification page of the *FinAppLending* package.

Fig. 4 Schema properties of the view model page



Add the following source code to the [Source code] tab:

```
define("UsrCommentlessAppValidationPage", [], function() {
    return {
        entitySchemaName: "AppValidation",
        diff: [{
            "operation": "remove",
            "name": "CommentContainer"
        }]
    };
});
```

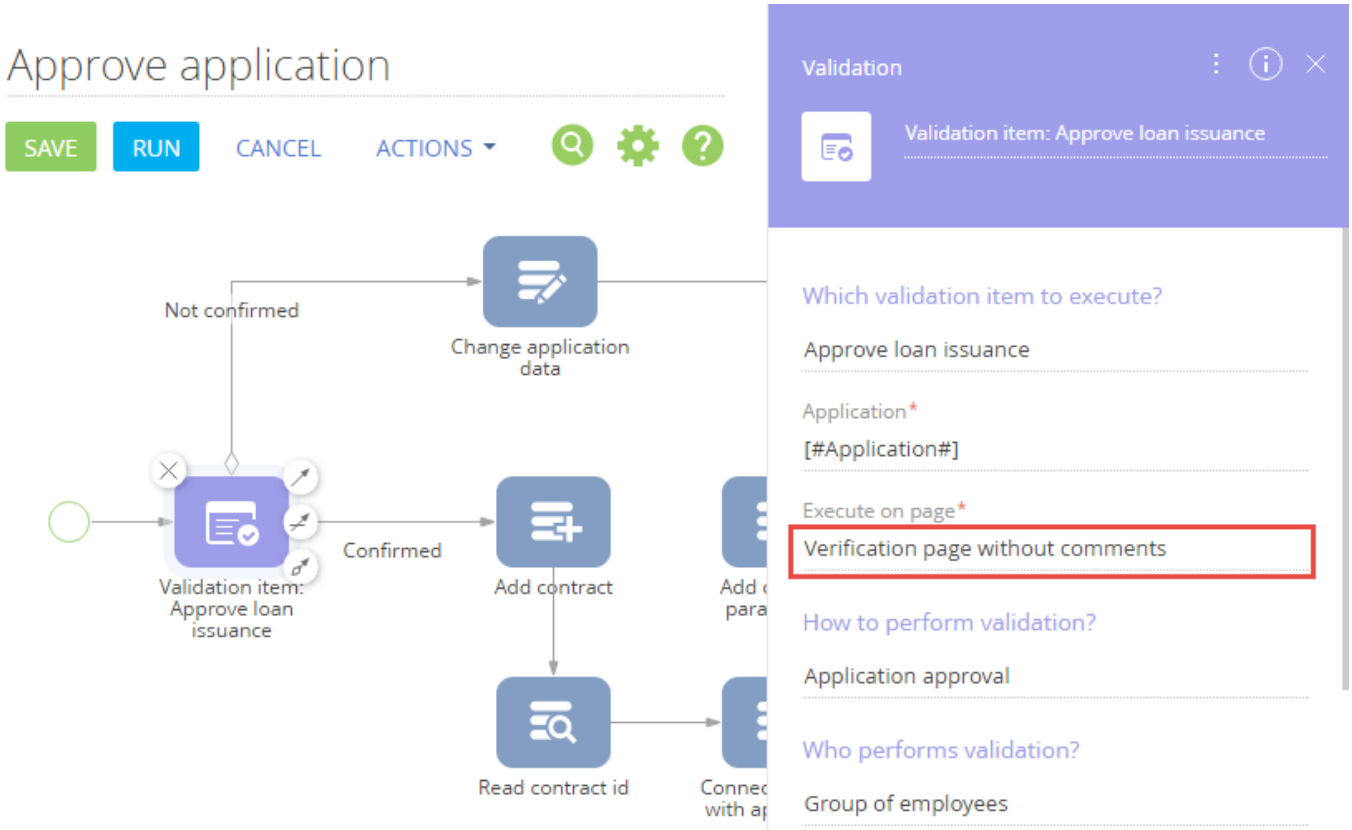
The [Comment] field is removed from the parent element in the **diff array**.

Save the schema to apply the changes.

2. Use the schema created in the business process.

To use the created schema, specify it in the [Execute on page] field of the [Validation item] item of the business process. This schema can be used in both new and existing business processes, such as *Approve application* (Fig. 5).

Fig. 5 Specifying the custom verification page

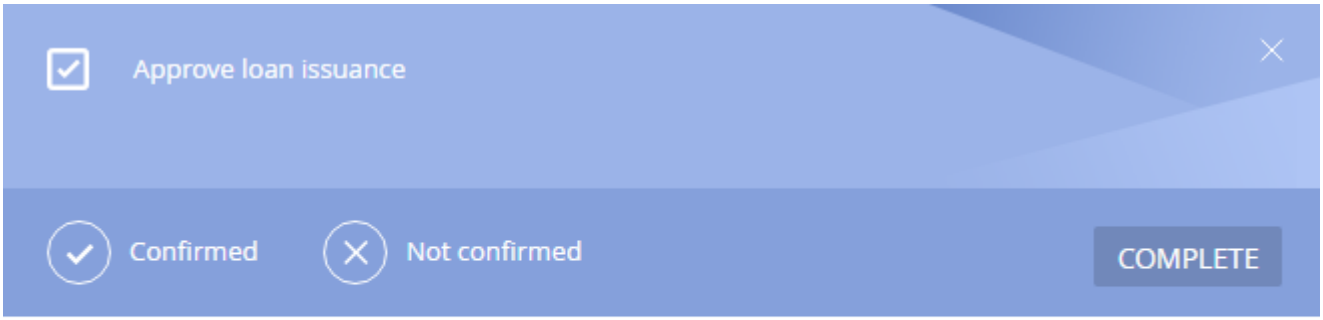


Save the business process to apply the changes.

> Attention!
>
> Restart the application in IIS for the changes to take effect.

After the changes are applied, the previous verification page (Fig. 3) will be replaced with a custom page that does not contain the [Comment] field (Fig. 6).

Fig. 6 Verification page without the [Comment] field.



Using the EntityMapper schema

Difficulty level

Introduction

Terrasoft.Configuration.EntityMapper is the utility configuration class, implemented in the *EntittyMapper* schema of the [FinAppLending] package in bpm’online lending. *EntittyMapper* enables you to match the data of one *entity* with another according to the rules defined in the configuration file. This approach prevents the creation of monotonous code.

Bpm’online lending features two objects with identical columns – [Contact] and [AppForm]. There are several details related to the [Contact] object and having similar details pertaining to [AppForm]. When the application is filled, there should be a possibility to get a list of all columns and values by the [Id] column of the [Contact] object, as well as a list of necessary details with their columns and values, and match this data with the application form data. After that, you can automatically fill out the fields of the application form with mapped data. This enables you to reduce manual data input.

Case description

Create a custom *UsrEntityMapperConfigsContainer* class to check the data matching mechanism through *Terrasoft.Configuration.EntityMapper*. Implement the data matching logic for the [Contact] and [AppForm] objects in this class. Implement a custom configuration service for data matching on the client side of the application. Add a button that will launch the custom configuration service on the edit page of the application form. The result has to be displayed in the browser’s console.

Case implementation algorithm

1. Create a custom UsrEntityMapperConfigsContainer class for data matching.

Learn more about the process of creating the [Source code] schema in the “**Creating the [Source code] schema**” article.

Property values for the created schema:

- [Title] — "UsrEntityMapperConfigsContainer".
- [Name] – "UsrEntityMapperConfigsContainer".
- [Package] — "Custom" (or a different custom package).

Add the following source code on the [Source Code] tab of the schema designer:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
// This class contains mapping settings.
public class UsrEntityMapperConfigsContainer
{
    // Settings for contact and application form mapping.
    public MapConfig ContactToAppFormConfig { get; protected set; }

    public UsrEntityMapperConfigsContainer() {
        this.InitContactToAppFormConfig();
    }
    // Configures the mapping of contact and application form objects.
    protected virtual void InitContactToAppFormConfig() {
        var columns = new Dictionary<string, string>();
        // In this case, the column names of the contact and the application form
coincided.
        columns.Add("Surname", "Surname");
        columns.Add("GivenName", "GivenName");
```

```
            columns.Add("MiddleName", "MiddleName");
            columns.Add("INN", "INN");
            columns.Add("SpouseSurname", "SpouseSurname");
            columns.Add("SpouseGivenName", "SpouseGivenName");
            columns.Add("SpouseMiddleName", "SpouseMiddleName");
            columns.Add("Spouse", "Spouse");
            var config = new MapConfig {
                SourceEntityName = "Contact",
                Columns = columns,
                RelationEntities = new List<RelationEntityMapConfig>() {
                        new RelationEntityMapConfig() {
                            SourceEntityName = "Contact",
                            ParentColumnName = "Spouse",
                            Columns = new Dictionary<string, string>() {
                                { "Surname", "SpouseSurname" },
                                { "BirthDate", "SpouseBirthDate" }
                            }
                        }
                    },
                    DetailsConfig = new List<DetailMapConfig>() {
                        new DetailMapConfig() {
                            SourceEntityName = "ContactAddress",
                            DetailName = "RegistrationAddressFieldsDetail",
                            Columns = new Dictionary<string, string>() {
                                { "AddressType", "AddressType" },
                                { "Country", "Country" },
                                { "Region", "Region" }
                            },
                            Filters = new List<EntityFilterMap>() {
                                new EntityFilterMap(){
                                    ColumnName = "AddressType",
                                    Value = BaseFinanceConst.RegistrationAddressTypeId
                                }
                            }
                        }
                    },
                    CleanDetails = new List<string>() {
                        "AppFormIncomeDetail"
                    }
                };
                this.ContactToAppFormConfig = config;
            }
        }
}
```

Publish the schema to apply changes.

2. Create a custom configuration service for data matching

The process of creating a custom configuration service is described in the “**How to create custom configuration service**” article.

Create the [Source Code] schema in a custom package. Property values for the created schema:

- [Title]— "UsrEntityMappingService".
- [Name] — "UsrEntityMappingService".
- [Package] — "Custom" (or a different custom package).

Add the following source code on the [Source Code] tab of the schema designer:

```
namespace Terrasoft.Configuration
{
using System;
```

```
using System.Linq;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Activation;
using System.ServiceModel.Web;
using Terrasoft.Core;
using Terrasoft.Core.Factories;
using Terrasoft.Core.Entities;
using Terrasoft.Common;
using System.Web;
using Terrasoft.Web.Common;
using Terrasoft.Nui.ServiceModel.DataContract;
using Terrasoft.Common.Json;
using Terrasoft.Core.Configuration;

    /// Service class for mapping entities and their details.
    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class UsrEntityMappingService: BaseService
    {
        private EntityMapper _entityMapper;
        // Returns an EntityMapper instance.
        protected virtual EntityMapper EntityMapper {
            get {
                return _entityMapper ?? (_entityMapper =
ClassFactory.Get<EntityMapper>(
                    new ConstructorArgument("userConnection", this.UserConnection)));
            }
        }
        // Returns mapping settings.
        protected virtual MapConfig GetConfig() {
            UsrEntityMapperConfigsContainer mapperConfigsContainer = new
UsrEntityMapperConfigsContainer();
            return mapperConfigsContainer.ContactToAppFormConfig;
        }
        // Performs the mapping and returns the result. The main method of service.
        [OperationContract]
        [return: MessageParameter(Name = "result")]
        [WebInvoke(Method = "POST", BodyStyle = WebMessageBodyStyle.Wrapped,
            RequestFormat = WebMessageFormat.Json, ResponseFormat =
WebMessageFormat.Json)]
        public EntityMappingResult GetMappedEntity(string id)
        {
            EntityMappingResult result = new EntityMappingResult();
            try {
                Guid recordId;
                MapConfig config = this.GetConfig();
                EntityResult entityResult = new EntityResult(config);
                result.columns = entityResult.Columns;
                result.details = entityResult.Details;
                if (!Guid.TryParse(id, out recordId)) {
                    return result;
                }
                entityResult = EntityMapper.GetMappedEntity(recordId, config);
                result.columns = entityResult.Columns;
                result.details = entityResult.Details;
                result.Success = true;
            } catch (Exception e){
                result.Success = false;
                result.Exception = e;
```

```
            }
            return result;
        }
    }

}
```

Publish the schema to apply changes.

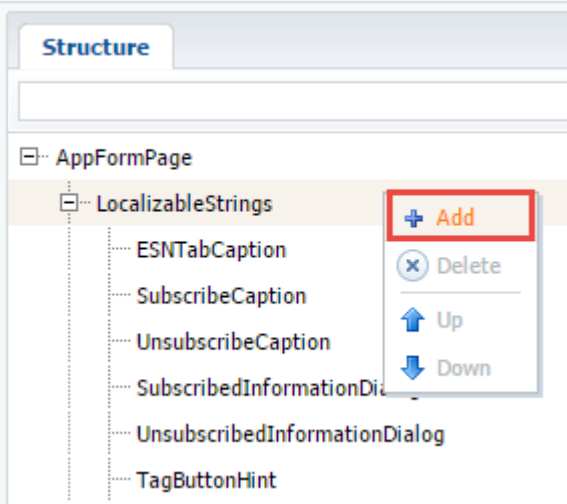
3. Adding a data mapping button to the application form edit page

To add a data mapping button, replace the existing [Application Form Edit Page] schema. The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**".

Add a localizable string to a replacing schema (Fig. 1) with the following properties:

- [Title] — "Call service".
- [Name] — "EntityMappingButtonCaption".

Fig. 1. Adding a localizable string



Add the following source code on the [Source Code] tab of the schema designer:

```
define("AppFormPage", [], function() {
    return {
        entitySchemaName: "AppForm",
        methods: {
            // User service request function.
            requestContactData2: function() {
                var data = {
                    id: this.get("Contact").value
                };
                // A configuration object for passing parameters to the service.
                var config = {
                    serviceName: "UsrEntityMappingService",
                    methodName: "GetMappedEntity",
                    data: data
                };
                // Calling a service.
                this.callService(config, this.parseMappedEntityResponse2, this);
            },
                //Callback-function for outputting the service response to the
console.
            parseMappedEntityResponse2: function(response) {
                window.console.log("Ответ от UsrEntityMappingService", response);
            }
        },
```

```
        diff: /**SCHEMA_DIFF*/[
            {
                "operation": "insert",
                "parentName": "ProfileContainer",
                "propertyName": "items",
                "name": "EntityMappingButton",
                "values": {
                    itemType: Terrasoft.ViewItemType.BUTTON,
                    "style": Terrasoft.controls.ButtonEnums.style.GREEN,
                    // Bind the button's title to the localized string of the schema.
                    caption: { bindTo: "Resources.Strings.EntityMappingButtonCaption"
},
                    // Bind the button click method-handler.
                    click: { bindTo: "requestContactData2" },
                    "layout": {
                        "column": 0,
                        "row": 2,
                        "colSpan": 24
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

Add the configuration object to the **diff array**. The object will be used to add the data matching button to the application form edit page. The *requestContactData2()* method is called when the button is pressed, and the *UsrEntityMappingService* configuration service is called in the method with all necessary parameters. The *parseMappedEntityResponse2()* callback-function will display the service response in the browser console.

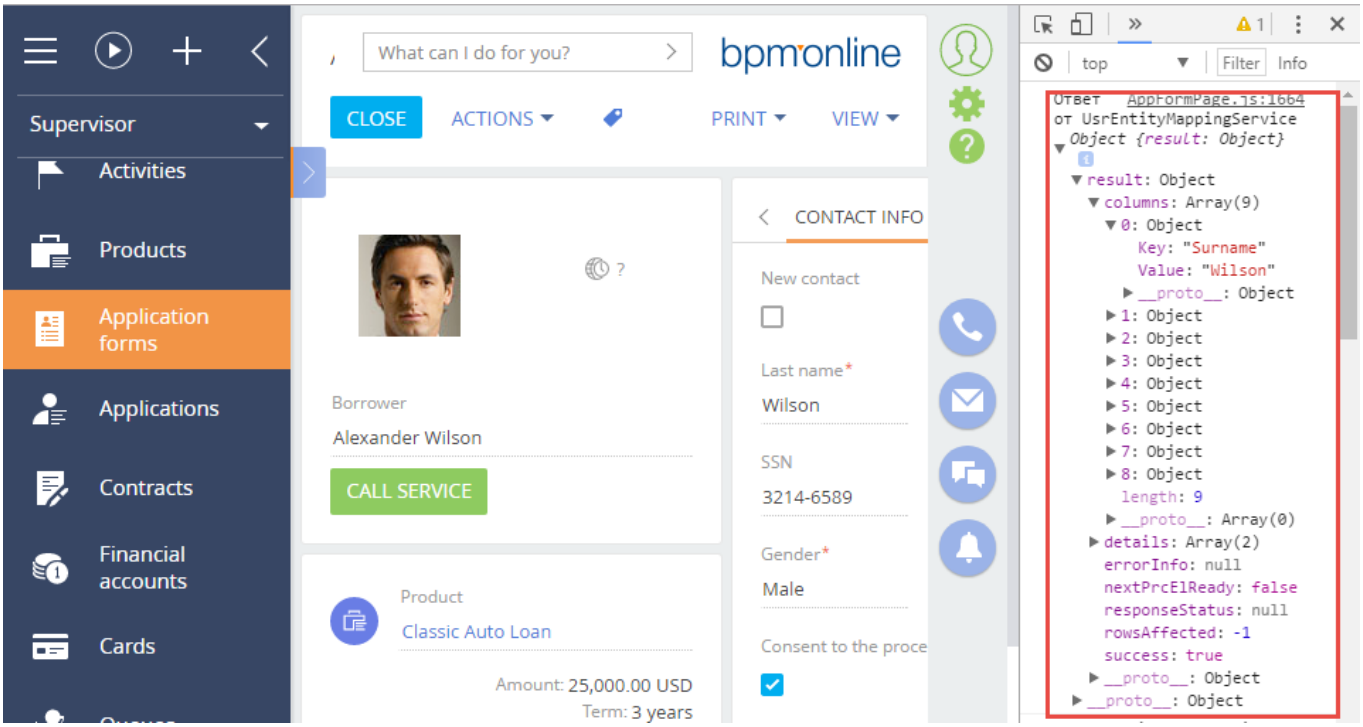
ATTENTION

Instead of browser console output, you can implement the mechanism of auto completing fields of the application form edit page with the matched values. However, this functionality is already implemented in the *requestContactData()* and *parseMappedEntityResponse()* methods of the *AppFormPage* parent schema.

Save the schema to apply changes.

As a result, the button will appear on the application form edit page. When you press the button, the object with mapped data will be displayed in the browser console.

Fig. 2. Case result



ATTENTION

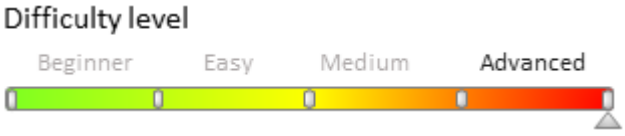
If the profile edit page is open in the new record creation mode, you must first select or create a contact connected to the created application form. If a contact is not selected, an exception will occur, because *This.get ("Contact")* returns *null*.

Marketing product customization

Contents

- **Adding a custom campaign element**

Adding a custom campaign element



Introduction

Use the Campaign designer to set up marketing campaigns. Using this designer, you can create a campaign diagram that consists of interconnected elements. In addition to default campaign elements you can create custom ones.

The general procedure for adding a custom campaign element is as follows:

1. Create a new element for the Campaign designer.
2. Create the element’s edit page.
3. Expand the Campaign designer menu with a new element.
4. Create the element’s server part.
5. Create executable element for the new campaign element.

6. Add custom logic for processing campaign events.

Case description

Create a new campaign element for sending text messages (SMS) for users.

Case implementation algorithm

1. Creating a new element for the Campaign designer

To display the element in the Campaign designer UI, add a new module schema for the campaign element. The procedure for creating a module schema is covered in the “**Creating a custom client module schema**” article. Set the following properties for the created schema:

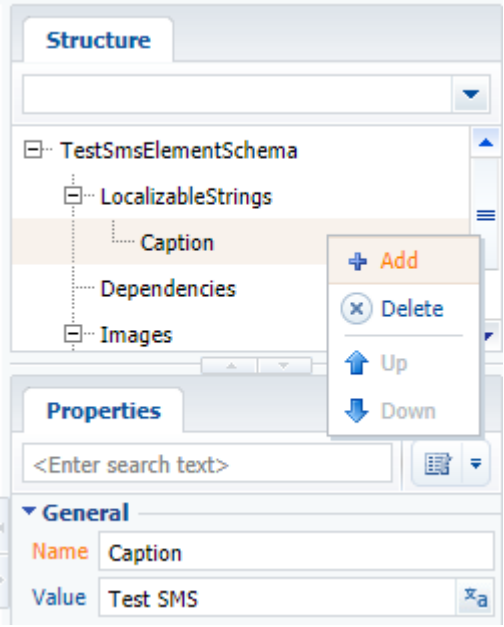
- [Title] – "Test SMS Element Schema".
- [Name] – "TestSmsElementSchema".

> ATTENTION
>
> The schema names in the case below do not contain the *Usr* prefix. You can change the default prefix in the [Prefix for object name] (*SchemaNamePrefix*) system setting.

Add a localized string (Fig. 1) to the schema:

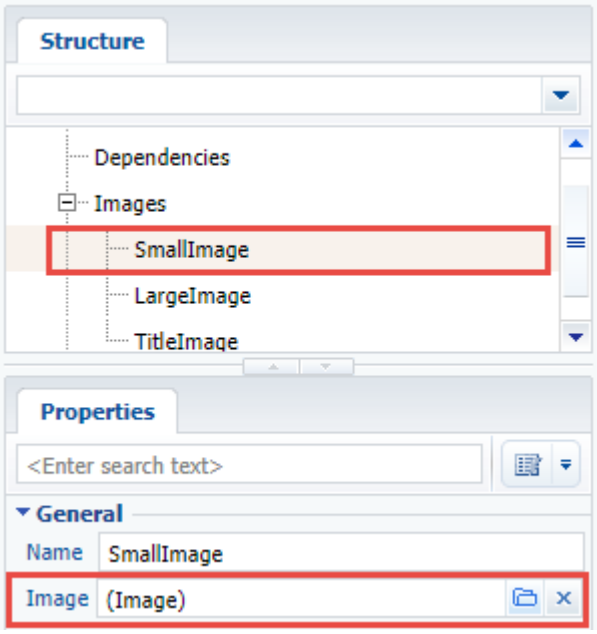
- [Name] – "Caption".
- [Value] – "Test SMS".

Fig. 1. Adding localized string to the schema



Add images that will represent the campaign element in the Campaign designer. Use the *SmallImage*, *LargeImage* and *TitleImage* (Fig. 2) properties to add the images.

Fig. 2. Adding a campaign element image



In this example we used a scalable vector graphics (SVG) image available here.

Add following source code on the [Source Code] section of the schema":

```
define("TestSmsElementSchema", ["TestSmsElementSchemaResources",
"CampaignBaseCommunicationSchema"],
    function(resources) {
        Ext.define("Terrasoft.manager.TestSmsElementSchema", {
            // Parent schema.
            extend: "Terrasoft.CampaignBaseCommunicationSchema",
            alternateClassName: "Terrasoft.TestSmsElementSchema",
            // Manager Id. Must be unique.
            managerItemUId: "a1226f93-f3e3-4baa-89a6-11f2a9ab2d71",
            // Plugged mixins.
            mixins: {
                campaignElementMixin: "Terrasoft.CampaignElementMixin"
            },
            // Element name.
            name: "TestSms",
            // Resource binding.
            caption: resources.localizableStrings.Caption,
            titleImage: resources.localizableImages.TitleImage,
            largeImage: resources.localizableImages.LargeImage,
            smallImage: resources.localizableImages.SmallImage,
            // Schema name of the edit page.
            editPageSchemaName: "TestSmsElementPropertiesPage",
            // Element type.
            elementType: "TestSms",
            // Full name of the class that corresponds to the current schema.
            typeName: "Terrasoft.Configuration.TestSmsElement,
Terrasoft.Configuration",
            // Overriding the properties of visual styles.
            color: "rgba(249, 160, 27, 1)",
            width: 69,
            height: 55,
            // Setting up element-specific properties.
            smsText: null,
            phoneNumber: null,
            // Determining the types of the elemen's outbound connections.
            getConnectionUserHandles: function() {
                return ["CampaignSequenceFlow", "CampaignConditionalSequenceFlow"];
```

```
            },
            // Expnding the properties for serialization.
            getSerializableProperties: function() {
                var baseSerializableProperties = this.callParent(arguments);
                return Ext.Array.push(baseSerializableProperties, ["smsText",
"phoneNumber"]);
            },
            // Setting up the icons that are displayed on the campaign diagram.
            getSmallImage: function() {
                return this.mixins.campaignElementMixin.getImage(this.smallImage);
            },
            getLargeImage: function() {
                return this.mixins.campaignElementMixin.getImage(this.largeImage);
            },
            getTitleImage: function() {
                return this.mixins.campaignElementMixin.getImage(this.titleImage);
            }
        });
        return Terrasoft.TestSmsElementSchema;
    });
```

Specifics:

- The *managerItemUId* property value must be unique for the new element and not repeat the value of the other elements.
- The *typeName* property contains the name of the C# class that corresponds to the campaign element name. This class will be saving and reading the element's properties from the schema metadata.

Save the schema to apply changes.

Adding a group of elements

If a new group of elements, such as [Scripts] must be created for the campaign element, the schema source code must be supplemented with the following code:

```
// Name of the new group.
group: "Scripts",

constructor: function() {
      if (!Terrasoft.CampaignElementGroups.Items.contains("Scripts")) {
           Terrasoft.CampaignElementGroups.Items.add("Scripts", {
                name: "Scripts",
                caption: resources.localizableStrings.ScriptsElementGroupCaption
           });
      }
      this.callParent(arguments);
}
```

Also, add a localized string with the following properties:

- [Name] – "ScriptsElementGroupCaption".
- [Name] – "Scripts".

Save the schema to apply changes.

2. Creating the element's edit page

Create the campaign element's edit page in the custom package to enable the users to view and edit the element's properties. To do this, create a schema that expands *BaseCampaignSchemaElementPage* (*CampaignDesigner* package). The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**" article.

Set the following properties for the created schema:

- [Title] – "TestSmsElementPropertiesPage".

- [Name] – "TestSmsElementPropertiesPage".
- [Parent object] – "BaseCampaignSchemaElementPage".

Add localized strings to the created schema (Fig. 1) with properties given in the table 1.

Table 1. Localized string properties

Name	Value
PhoneNumberCaption	Sender phone number
SmsTextCaption	Message
TestSmsText	Which text message should be sent? (Which SMS text to send?)

Add following source code on the [Source Code] section of the schema":

```
define("TestSmsElementPropertiesPage", [],
    function() {
        return {
            attributes: {
                // Attributes that correspond to specific properties of element
schema.
                "PhoneNumber": {
                    "dataValueType": this.Terrasoft.DataValueType.TEXT,
                    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN
                },
                "SmsText": {
                    "dataValueType": this.Terrasoft.DataValueType.TEXT,
                    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN
                }
            },
            methods: {
                init: function() {
                    this.callParent(arguments);
                    this.initAcademyUrl(this.onAcademyUrlInitialized, this);
                },
                // Element code for generating a contextual help link.
                getContextHelpCode: function() {
                    return "CampaignTestSmsElement";
                },
                // Initialization of attributes with the current schema property
values.
                initParameters: function(element) {
                    this.callParent(arguments);
                    this.set("SmsText", element.smsText);
                    this.set("PhoneNumber", element.phoneNumber);
                },
                // Saving schema properties.
                saveValues: function() {
                    this.callParent(arguments);
                    var element = this.get("ProcessElement");
                    element.smsText = this.getSmsText();
                    element.phoneNumber = this.getPhoneNumber();

                },
                // Reading current attribute values.
                getPhoneNumber: function() {
                    var number = this.get("PhoneNumber");
                    return number ? number : "";
                },
                getSmsText: function() {
                    var smsText = this.get("SmsText");
```

```
            return smsText ? smsText : "";
        }
    },
    diff: [
        // UI container.
        {
            "operation": "insert",
            "name": "ContentContainer",
            "propertyName": "items",
            "parentName": "EditorsContainer",
            "className": "Terrasoft.GridLayoutEdit",
            "values": {
                "itemType": Terrasoft.ViewItemType.GRID_LAYOUT,
                "items": []
            }
        },
        // Element primary signature.
        {
            "operation": "insert",
            "name": "TestSmsLabel",
            "parentName": "ContentContainer",
            "propertyName": "items",
            "values": {
                "layout": {
                    "column": 0,
                    "row": 0,
                    "colSpan": 24
                },
                "itemType": this.Terrasoft.ViewItemType.LABEL,
                "caption": {
                    "bindTo": "Resources.Strings.TestSmsText"
                },
                "classes": {
                    "labelClass": ["t-title-label-proc"]
                }
            }
        },
        // Caption for the text field where sender name is entered.
        {
            "operation": "insert",
            "name": "PhoneNumberLabel",
            "parentName": "ContentContainer",
            "propertyName": "items",
            "values": {
                "layout": {
                    "column": 0,
                    "row": 1,
                    "colSpan": 24
                },
                "itemType": this.Terrasoft.ViewItemType.LABEL,
                "caption": {
                    "bindTo": "Resources.Strings.PhoneNumberCaption"
                },
                "classes": {
                    "labelClass": ["label-small"]
                }
            }
        },
        // Text field for entering phone number.
        {
            "operation": "insert",
            "name": "PhoneNumber",
```

```
            "parentName": "ContentContainer",
            "propertyName": "items",
            "values": {
                "labelConfig": {
                    "visible": false
                },
                "layout": {
                    "column": 0,
                    "row": 2,
                    "colSpan": 24
                },
                "itemType": this.Terrasoft.ViewItemType.TEXT,
                "classes": {
                    "labelClass": ["feature-item-label"]
                },
                "controlConfig": { "tag": "PhoneNumber" }
            }
        },
        // Caption for text field for entering message text.
        {
            "operation": "insert",
            "name": "SmsTextLabel",
            "parentName": "ContentContainer",
            "propertyName": "items",
            "values": {
                "layout": {
                    "column": 0,
                    "row": 3,
                    "colSpan": 24
                },
                "classes": {
                    "labelClass": ["label-small"]
                },
                "itemType": this.Terrasoft.ViewItemType.LABEL,
                "caption": {
                    "bindTo": "Resources.Strings.SmsTextCaption"
                }
            }
        },
        // Text field for entering message text.
        {
            "operation": "insert",
            "name": "SmsText",
            "parentName": "ContentContainer",
            "propertyName": "items",
            "values": {
                "labelConfig": {
                    "visible": false
                },
                "layout": {
                    "column": 0,
                    "row": 4,
                    "colSpan": 24
                },
                "itemType": this.Terrasoft.ViewItemType.TEXT,
                "classes": {
                    "labelClass": ["feature-item-label"]
                },
                "controlConfig": { "tag": "SmsText" }
            }
        }
    ]
```

```
            };
        }
);
```

Save the schema to apply changes.

3. Expanding the Campaign designer menu with a new element

To display the new element in the Campaign designer menu, expand the campaign element base schema manager. Add a schema that expands *CampaignElementSchemaManagerEx* (the *CampaignDesigner* package) to the custom package. The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**" article.

Set the following properties for the created schema:

- [Title] – "TestSmsCampaignElementSchemaManagerEx".
- [Name] – "CampaignElementSchemaManagerEx".
- [Parent object] – "CampaignElementSchemaManagerEx".

Add following source code on the [Source Code] section of the schema":

```
require(["CampaignElementSchemaManager", "TestSmsElementSchema"],
    function() {
        // Adding a new schema to the list of available element schemas in the
Campaign designer.
        var coreElementClassNames =
Terrasoft.CampaignElementSchemaManager.coreElementClassNames;
        coreElementClassNames.push({
            itemType: "Terrasoft.TestSmsElementSchema"
        });
    });
```

Save the schema to apply changes.

4. Creating server part of the custom campaign element

To implement saving the campaign element properties, create a class that interacts with the application server part. The class must inherit *CampaignSchemaElement* and override the *ApplyMetaDataValue()* and *WriteMetaData()* methods.

Create a source code schema with the following properties:

- [Title] – "TestSmsElement".
- [Name] – "TestSmsElement".

For more information on creating source code schemas, please see the **Creating the [Source code] schema** article.

Add the following source code on the [Source Code] section of the schema":

```
namespace Terrasoft.Configuration
{
    using System;
    using Terrasoft.Common;
    using Terrasoft.Core;
    using Terrasoft.Core.Campaign;
    using Terrasoft.Core.Process;

    [DesignModeProperty(Name = "PhoneNumber",
        UsageType = DesignModeUsageType.NotVisible, MetaPropertyName =
PhoneNumberPropertyName)]
    [DesignModeProperty(Name = "SmsText",
        UsageType = DesignModeUsageType.NotVisible, MetaPropertyName =
SmsTextPropertyName)]
    public class TestSmsElement : CampaignSchemaElement
```

```
    {
        private const string PhoneNumberPropertyName = "PhoneNumber";
        private const string SmsTextPropertyName = "SmsText";
        // Default constructor.
        public TestSmsElement() {
            ElementType = CampaignSchemaElementType.AsyncTask;
        }
        // Constructor with parameter.
        public TestSmsElement(TestSmsElement source)
                : base(source) {
            ElementType = CampaignSchemaElementType.AsyncTask;
            PhoneNumber = source.PhoneNumber;
            SmsText = source.SmsText;
        }

        // Instance action Id.
        protected override Guid Action {
            get {
                return CampaignConsts.CampaignLogTypeMailing;
            }
        }

        // Phone number.
        [MetaTypeProperty("{A67950E7-FFD7-483D-9E67-3C9A30A733C0}")]
        public string PhoneNumber {
            get;
            set;
        }
        // Text message.
        [MetaTypeProperty("{05F86DF2-B9FB-4487-B7BE-F3955703527C}")]
        public string SmsText {
            get;
            set;
        }
        // Applies metadata values.
        protected override void ApplyMetaDataValue(DataReader reader) {
            base.ApplyMetaDataValue(reader);
            switch (reader.CurrentName) {
                case PhoneNumberPropertyName:
                    PhoneNumber = reader.GetValue<string>();
                    break;
                case SmsTextPropertyName:
                    SmsText = reader.GetValue<string>();
                    break;
            }
        }

        // Records metadata values.
        public override void WriteMetaData(DataWriter writer) {
            base.WriteMetaData(writer);
            writer.WriteValue(PhoneNumberPropertyName, PhoneNumber, string.Empty);
            writer.WriteValue(SmsTextPropertyName, SmsText, string.Empty);
        }
        // Copies element.
        public override object Clone() {
            return new TestSmsElement(this);
        }

        // Creates a specific ProcessFlowElement instance.
        public override ProcessFlowElement CreateProcessFlowElement(UserConnection
userConnection) {
            var executableElement = new TestSmsCampaignProcessElement {
```

```
                UserConnection = userConnection,
                SmsText = SmsText,
                PhoneNumber = PhoneNumber
            };
            InitializeCampaignProcessFlowElement(executableElement);
            return executableElement;
        }
    }
}
```

Publish the source code schema.

5. Creating executable element for the new campaign element

For the custom campaign element to execute the needed logic, add an executable element. It is a class that inherits the CampaignProcessFlowElement class, where the *SafeExecute()* method is implemented.

To create an executable element, add a source code schema element with the following properties in the custom package:

- [Title] – "TestSmsCampaignProcessElement".
- [Name] – "TestSmsCampaignProcessElement".

Add following source code on the [Source Code] section of the schema":

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
    using Terrasoft.Core.Campaign;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Process;

    public class TestSmsCampaignProcessElement : CampaignProcessFlowElement
    {
        public const string ContactTableName = "Contact";

        public TestSmsCampaignProcessElement(ICampaignAudience campaignAudience) {
            CampaignAudience = campaignAudience;
        }

        public TestSmsCampaignProcessElement() {
        }

        // Audiences for whom to send texts on the current step.
        private ICampaignAudience _campaignAudience;
        private ICampaignAudience CampaignAudience {
            get {
                return _campaignAudience ??
                    (_campaignAudience = new CampaignAudience(UserConnection,
CampaignId));
            }
            set {
                _campaignAudience = value;
            }
        }

        // SMS-specific properties. Passed from an instance of the TestSmsElement
class.
        public string PhoneNumber {
            get;
            set;
        }
```

```
        public string SmsText {
            get;
            set;
        }

        // Implementation of the element execution method
        protected override int SafeExecute(ProcessExecutingContext context) {
            // TODO: Implement sending SMS messages.
            //
            // Current step for audiences is set as completed.
            return CampaignAudience.SetItemCompleted(SchemaElementUId);
        }
    }
}
```

Publish the source code schema.

6. Adding custom logic for processing campaign events

Use the event handler mechanism to implement custom logic on saving, copying, deleting, running and stopping campaigns. Create a public *sealed* handler class that inherits *CampaignEventHandlerBase*. Implement interfaces that describe specific event handler signatures. This class must not be generic. It must have a constructor available by default.

The following interfaces are supported in the current version:

- *IOnCampaignBeforeSave* – contains method that will be called before saving the campaign.
- *IOnCampaignAfterSave* – contains method that will be called after saving the campaign.
- *IOnCampaignDelete* – contains method that will be called before deleting the campaign.
- *IOnCampaignStart* – contains method that will be called before running the campaign.
- *IOnCampaignStop* – contains method that will be called before stopping the campaign.
- *IOnCampaignValidate* – contains method that will be called on validating the campaign.
- *IOnCampaignCopy* – contains method that will be called after copying the campaign.

If an exception occurs during the event processing, the call chain is stopped, and campaign status is reverted to the previous one in DB.

> **NOTE**
>
> When implementing the *IOnCampaignValidate* interface, save errors in the campaign schema using the *AddValidationInfo(string)* method.

Additional case conditions

In order for the new custom campaign element to work, SMS gateway connection is required. The connection, account status and other parameters must be checked during campaign validation. The messages must be sent when campaign starts.

To implement these conditions, add a source code schema element with the following properties in the custom package:

- [Title] – "TestSmsEventHandler".
- [Name] – "TestSmsEventHandler".

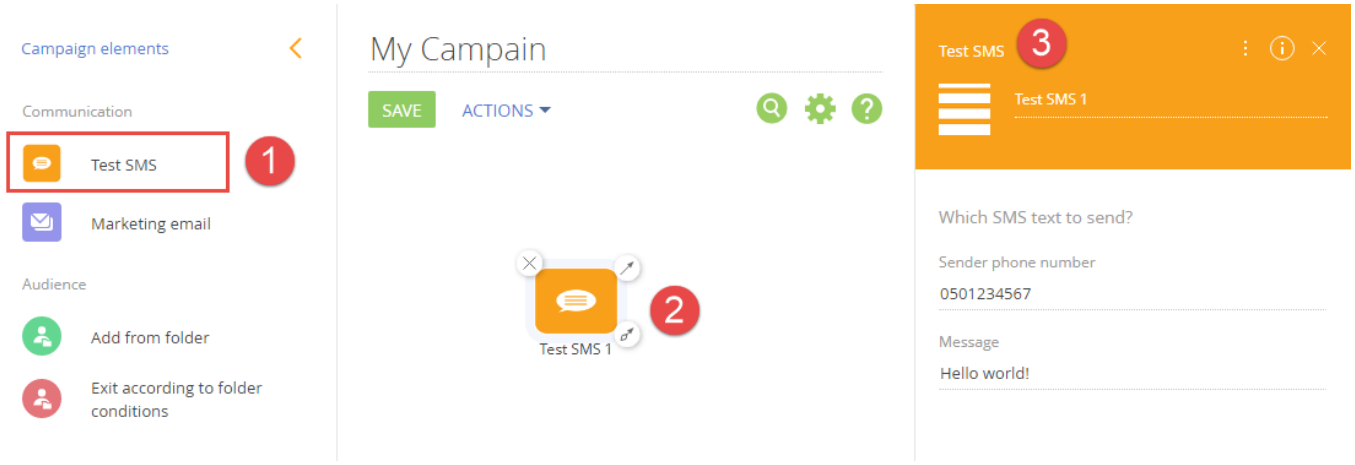
Add following source code on the [Source Code] section of the schema":

```
namespace Terrasoft.Configuration
{
    using System;
    using Terrasoft.Core.Campaign.EventHandler;

    public sealed class TestSmsEventHandler : CampaignEventHandlerBase,
IOnCampaignValidate, IOnCampaignStart
```

```
    {
        // Implementing handler for the campaign start event.
        public void OnStart() {
            // TODO: Text SMS message sending logic...
            //
        }
        // Implementing event handler for campaign validation.
        public void OnValidate() {
            try {
                // TODO: SMS gateway connection validation logic...
                //
            } catch (Exception ex) {
                // If errors are found, add information to the campaign schema.
                CampaignSchema.AddValidationInfo(ex.Message);
            }
        }
    }
}
```

After making the changes, publish the schema. Compile the application and clear the cache.

As a result, a new [TestSMS] element will be added in the campaign element menu (Fig. 3, 1) that the users can add to the campaign diagram (Fig. 3, 2). When an added element is selected, its edit page will be displayed (Fig. 3, 3).



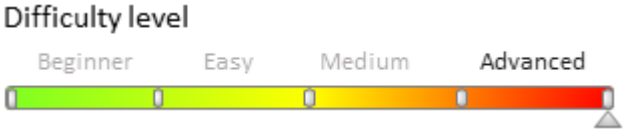
> **ATTENTION**
>
> When saving the campaign, the "Parameter 'type' cannot be null" may occur. The error indicates that the configuration library was not updated after the compilation and therefor does not contain the new types.
>
> Recompile the project and clear all possible storages with cached data. You may also need to clear the application pool and restart the website in IIS on the application server.

Prediction

Contents

- **How to implement custom prediction model**

How to implement custom prediction model



Introduction

Prediction service of the lookup field uses methods of statistic analysis for learning on the base of historical data and prediction of values for new records.

For more information about thess functions please refer to the “**Machine learning service** ” article.

Case description

Implement automatic prediction for the [AccountCategory] column by the values of the [Country], [EmployeesNumber] and [Industry] field while saving the account record. The following conditions should be met:

- Model learning should be created on the base of account records for last 90 days.
- Moodel Retraining should be performed every 30 days.
- Permissible value of prediction accuracy for the model – 0,6.

ATTENTION

To complete this case you need to check the correctness of the value of the [Bpmonline cloud services API key] (*CloudServicesAPIKey* code) system setting and the URL of the predictive service in the [Service endpoint Url] field of the [ML problem types] lookup.

Case implementation algorithm

1. Model learning

To learn the model:

1. Add a record to the [*ML Model*] lookup. Values of the record fields are given in the Table 1.

Table 1. Values of the record fields of the MLModel lookup

Field	Value
Name	Predict account category
ML problem type	Lookup prediction
Target schema for prediction	Account
Quality metric low limit	0,6
Model retrain frequency (days)	30
Training set metadata	{ "inputs": [{ "name": "CountryId", "type": "Lookup", "isRequired": true }, { "name": "EmployeesNumberId", "type": "Lookup", "isRequired": true }, {

```
                "name": "IndustryId",
                "type": "Lookup",
                "isRequired": true
            }
        ],
        "output": {
            "name": "AccountCategoryId",
            "type": "Lookup",
            "displayName": "AccountCategory"
        }
    }
```

Training set query	`new Select(userConnection) .Column("a", "Id").As("Id") .Column("a", "CountryId") .Column("a", "EmployeesNumberId") .Column("a", "IndustryId") .Column("a", "AccountCategoryId") .Column("c", "Name").As("AccountCategory") .From("Account").As("a") .InnerJoin("AccountCategory").As("c").On("c", "Id").IsEqual("a", "AccountCategoryId") .Where("a", "CreatedOn").IsGreater(Column.Parameter(DateTime.Now.AddDays(-90)))`
Predictions enabled (checkbox)	Enable

2. Perform the [Execute model training job] action on the [*ML Model*] lookup field.

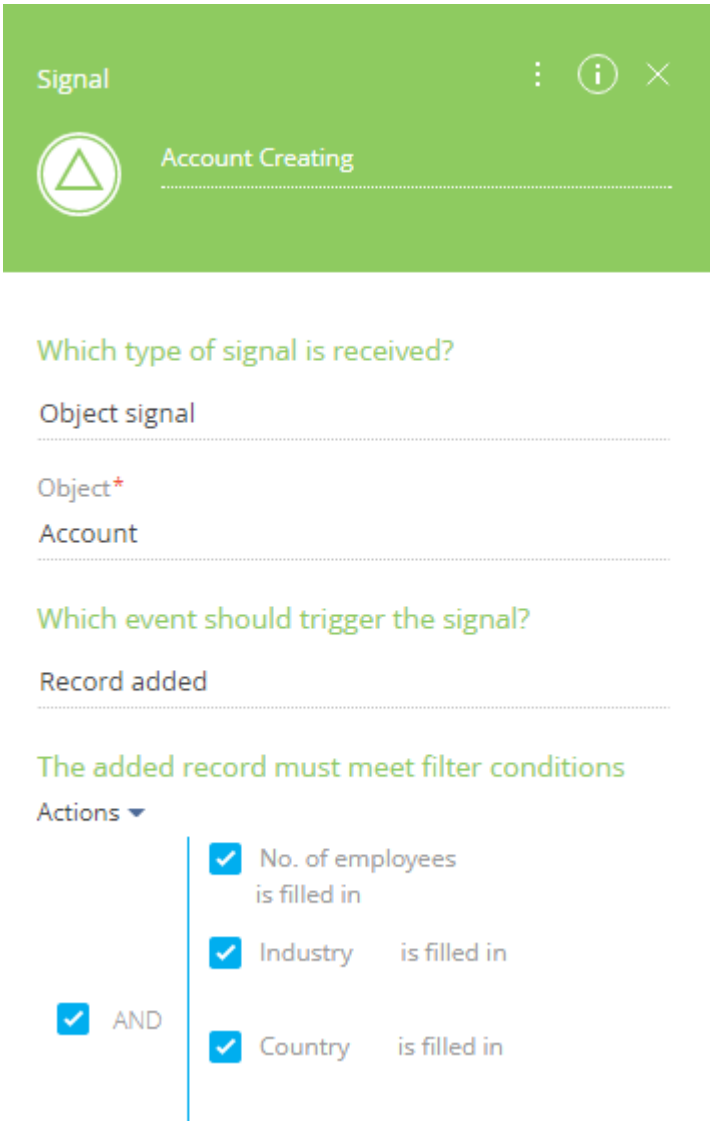
Wait until the values of the [Model processing status] field will be changed in following sequence: *DataTransfer*, *QueuedToTrain*, *Training*, *Done*. The process may take several hours to finish (it depends on the amount of passed data and general workload of the predictive service.

2. Performing the prediction

To start the predictions:

1. Create a business process in the user package. Select the saving of the [Contact] object as a start signal for the process. Check if the required fields are field (Fig. 1).

Fig. 1. Start signal properties.



2. Add the *MLModelId* lookup parameter that refers to the [ML Model] entity. Select the record with the [Predict account category] model as a value.

3. Add the *RecordId* lookup parameter that refers to the [Account] entity. Select the a reference on the*RecordId* parameter of the [Signal] element as a value.

4. Add a [Script task] element on the business process diagram and add the following code there:

```
var userConnection = Get<UserConnection>("UserConnection");
// Getting the Id of the Account record.
Guid entityId = Get<Guid>("RecordId");
// Geeting the id of the model.
var modelId = Get<Guid>("MLModelId");
var connectionArg = new ConstructorArgument("userConnection", userConnection);
// Object for calling prediction.
var predictor = ClassFactory.Get<LookupMLPredictor>(connectionArg);
// Model load.
if (predictor.TryLoadModelDataForPrediction(modelId)) {
    // Predictable entity schema name.
    var schemaName = "Account";
    // Mapping Entity Fields to Model Fields.
    var inputColumnPathMap = new Dictionary<string, string> {
        { "Country", "CountryId" },
        { "Industry", "IndustryId" },
        { "EmployeesCount.ZZZ.ZZZ", "EmployeesCountId" }
```

```
    };
    // Call of the forecasting service. The Data is saved in MLPrediction and in case
of high probability of forecasting the data is saved in the required field of the
Account.
    predictor.PredictAndSaveResults(schemaName, entityId, inputColumnPathMap,
"AccountCategory");
}
return true;
```

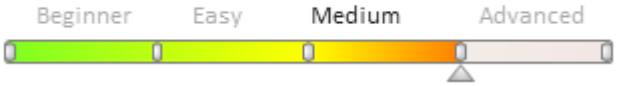
After saving and compiling the process, the prediction will be performed for new accounts. The prediction will be displayed on the account edit page.

ATTENTION

This implementation of the prediction slows down the saving an account record because call of the prediction service is executed in 2 seconds. This can reduce the performance of the mass operations with data saving, like import from Excel.

Integration with bpm'online and public API

Difficulty level



Bpm’online has a wide range of integrations with custom third-party applications.

First, accessing bpm’online by a third-party application requires authentication. For more information on accessing bpm’online, see “**Authenticating external requests to bpm'online services**”. For more information on bpm’online primary authentication service, see “**The AuthService.svc authentication service**”.

Starting with version 7.10, authentication is protected from CSRF attacks. For more information, see “**Protection from CSRF attacks during integration with bpm'online**”.

Brief description and comparison of bpm’online basic integration methods is available in the “**Choosing the method of integration with bpm'online**” article.

Wide range of integration capabilities is available through API provided by the DataService web service. Fore more information on create, read, update and delete operations (the CRUD operations), as well as the API, see the “**DataService web service**” article.

If the third-party system uses the OData protocol, it cal also be used for integration with bpm’online. For more information, see the “**OData**” article.

Using the iframe HTML element for integration is covered in the “**Integration of third-party sites via iframe**” article.

For more information on the “Web-to-Object” integrations, see the “**Web-To-Object. Using landings and web-forms**”.

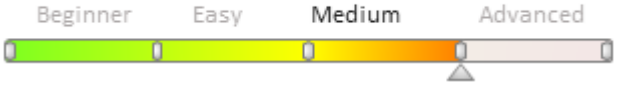
Use methods of the ProcessEngineService.svc web service to trigger bpm'online processes via third-party applications. Description of methods and public web service API are available in the following article: "The ProcessEngineService.svc web service".

Contents

- **Choosing the method of integration with bpm'online**
- **Authenticating external requests to bpm'online services**
- **The AuthService.svc authentication service**
- **Protection from CSRF attacks during integration with bpm'online**
- **DataService web service**
- **OData**
- **Integration of third-party sites via iframe**
- **Web-To-Object. Using landings and web-forms**
- **The ProcessEngineService.svc web service**

Choosing the method of integration with bpm'online

Difficulty level



Introduction

Bpm’online enables range of methods for integration with third-party software products. Choosing the method of integration depends on the needs of the client, the type and architecture of third-party software products and the developer's skills. A comparison of the main characteristics of the supported methods of integration with bpm'online is given in Table 1.

Table 1. Comparison of main methods of integration with bpm’online

DataService	OData	Configuration service	Web-to-Object	iframe
Supported formats of the data exchange				
XML, JSON, JSV, CSV	XML, JSON	XML, JSON	JSON	No
Tasks are being solved				
CRUD operations with bpm'online objects, data filtering and use of built-in bpm'online macros	CRUD-operations with objects, adding and removing links, obtaining metadata, collections, object fields, sorting, etc.	All user tasks that can be solved within the open bpm'online API	Only adding objects	Only interaction with user interface of the integrated third-party web application (web page).
Complexity				
High	Medium	Medium	Medium	Low
Ways of authentication				
Forms	Basic, Forms	Anonymous, Forms — depends on service implementation.	Forms	Forms (with bpm'online)
Availability of auxiliary custom libraries				
Bpm’online .dll libraries can be used only for .NET applications	Enabled http://www.odata.org/libraries	No need	No need	No need
Developer				
bpm'online	Microsoft	bpm'online	bpm'online	bpm'online

A brief description and the main advantages and disadvantages of each of the methods are given below.

Integration with DataService

The DataService web service is the main link between the bpm'online client and server parts. It helps to transfer the data that were entered by user via user interface to server side of the application for further processing and saving to the database. More information about the DataService web service can be found in the “**DataService web service**” article.

Key benefits and integration options

- Data exchange with XML, JSON, JSV, CSV.
- Available operations of **creating**, **reading**, **updating** and **deleting** the bpm’online objects (CRUD operations). You can use **built-in macros** and **data filtering**. **Batch processing** is available for

complex queries.
- User authorization is required for access.

Disadvantages

- High complexity of query building.
- Required in-depth knowledge for development.
- Auxiliary libraries for popular application and mobile platforms are disabled.

Example

An example of the source code of a simple application for sending adding data request to the DataService web service is given below. A request is made to create a new contact, for which the main columns are filled. Detail description of this example can be found in the “**DataService. Adding records**

```
using System;
using System.Text;
using System.IO;
using System.Net;
using System.Collections.Generic;
using Terrasoft.Nui.ServiceModel.DataContract;
using Terrasoft.Core.Entities;
using System.Web.Script.Serialization;

namespace DataServiceInsertExample
{
    class Program
    {
        private const string baseUri = @"http://userapp.bpmonline.com";
        private const string authServiceUri = baseUri +
@"/ServiceModel/AuthService.svc/Login";
        private const string insertQueryUri = baseUri +
@"/0/DataService/json/reply/InsertQuery";
        private static CookieContainer AuthCookie = new CookieContainer();

        private static bool TryLogin(string userName, string userPassword)
        {
            bool result = false;
            // TODO: Implementation of authentication.
            return result;
        }

        static void Main(string[] args)
        {
            if (!TryLogin("User1", "User1"))
            {
                return;
            }
            // Generate a JSON object for requesting the addition of data.
            // Use the InsertQuery data contract.
            var insertQuery = new InsertQuery()
            {
                RootSchemaName = "Contact",
                ColumnValues = new ColumnValues()
            };
            var columnExpressionName = new ColumnExpression
            {
                ExpressionType = EntitySchemaQueryExpressionType.Parameter,
                Parameter = new Parameter
                {
                    Value = "John Smith",
```

```
                    DataValueType = DataValueType.Text
                }
            };
            var columnExpressionPhone = new ColumnExpression
            {
                ExpressionType = EntitySchemaQueryExpressionType.Parameter,
                Parameter = new Parameter
                {
                    Value = "+12 345 678 00 00",
                    DataValueType = DataValueType.Text
                }
            };
            var columnExpressionJob = new ColumnExpression
            {
                ExpressionType = EntitySchemaQueryExpressionType.Parameter,
                Parameter = new Parameter
                {
                    Value = "11D68189-CED6-DF11-9B2A-001D60E938C6",
                    DataValueType = DataValueType.Guid
                }
            };
            insertQuery.ColumnValues.Items = new Dictionary<string, ColumnExpression>
();
            insertQuery.ColumnValues.Items.Add("Name", columnExpressionName);
            insertQuery.ColumnValues.Items.Add("Phone", columnExpressionPhone);
            insertQuery.ColumnValues.Items.Add("Job", columnExpressionJob);
            var json = new JavaScriptSerializer().Serialize(insertQuery);
            byte[] jsonArray = Encoding.UTF8.GetBytes(json);

            // Sending an Http-request.
            var insertRequest = HttpWebRequest.Create(insertQueryUri) as
HttpWebRequest;
            insertRequest.Method = "POST";
            insertRequest.ContentType = "application/json";
            insertRequest.CookieContainer = AuthCookie;
            insertRequest.ContentLength = jsonArray.Length;
            using (var requestStream = insertRequest.GetRequestStream())
            {
                requestStream.Write(jsonArray, 0, jsonArray.Length);
            }
            using (var response = (HttpWebResponse)insertRequest.GetResponse())
            {
                using (StreamReader reader = new
StreamReader(response.GetResponseStream()))
                {
                    Console.WriteLine(reader.ReadToEnd());
                }
            }
        }
    }
}
```

Integration with OData

Open Data (OData) protocol is an open web protocol for data request and update based on the REST architectural approach using Atom/XML and JSON standards. Any third-party application that supports HTTP messaging and can process XML or JSON data can have the access to the bpm’online data and objects. Data is available as resources addressed through a URI. Access to the data and modification are performed by standard HTTP commands (GET, PUT, MERGE, POST and DELETE). More information about OData protocol can be found in the “**Possibilities for the bpm'online integration over the OData protocol**” article.

Key benefits and integration options

- Data exchange with XML, JSON.
- A lot of operations with bpm'online objects, including CRUD operations.
- Convenient functions for working with strings, dates and time.
- A large number of custom libraries for working with OData for popular application and mobile platforms.
- User authorization is required for access.

More information about OData integration can be found in the “**Possibilities for the bpm'online integration over the OData protocol**” article.

Disadvantages

- Complexity of query building.
- Requires advanced skills for development.

Case example

Example of method source code for adding a new contact record using OData is given below. Detail description of this example can be found in the “**Working with bpm'online objects over the OData protocol using Http request**” article.

```
// POST <bpm'online URL>/0/ServiceModel/EntityDataService.svc/ContactCollection/

public static void CreateBpmEntityByOdataHttpExample()
{
    // Create an xml message that contains information about the object being
created.
    var content = new XElement(dsmd + "properties",
                  new XElement(ds + "Name", "John Smith"),
                  new XElement(ds + "Dear", "John"));
    var entry = new XElement(atom + "entry",
                new XElement(atom + "content",
                new XAttribute("type", "application/xml"), content));
    Console.WriteLine(entry.ToString());
    // Create a service request that will add a new object to the collection of
contacts.
    var request = (HttpWebRequest)HttpWebRequest.Create(serverUri +
"ContactCollection/");
    request.Credentials = new NetworkCredential("BPMUserName", "BPMUserPassword");
    request.Method = "POST";
    request.Accept = "application/atom+xml";
    request.ContentType = "application/atom+xml;type=entry";
    // Write an xml message to the request thread.
    using (var writer = XmlWriter.Create(request.GetRequestStream()))
    {
        entry.WriteTo(writer);
    }
    // Receiving a response from the service about the result of the operation.
    using (WebResponse response = request.GetResponse())
    {
        if (((HttpWebResponse)response).StatusCode == HttpStatusCode.Created)
        {
            // Processing the result of the operation.
        }
    }
}
```

Integration with custom configuration web service

Bpm’online enables to create custom web services in the configuration that can implement specific integration tasks.

Configuration web service is a RESTful service implemented on the basis on WCF technology. More information about creation of custom configuration web service can be found in the “**How to create custom configuration service**” article.

Key benefits and integration options

- Data exchange is implemented by developer in any convenient way.
- Developer can implement any operation with bpm'online objects, including CRUD operations.
- User authorization is not required for access.

Disadvantages

- The entire functionality of the service need to be developed.
- Required in-depth knowledge for development.

Case example

The complete source code of the configuration service is available below: Service adds the "changed!” word to the incoming parameter and sends a new value in the HTTP answer. Detail description of this example can be found in the “**How to create custom configuration service**” article.

```
namespace Terrasoft.Configuration.CustomConfigurationService
{
    using System;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;

    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class CustomConfigurationService
    {
        [OperationContract]
        [WebInvoke(Method = "POST", RequestFormat = WebMessageFormat.Json, BodyStyle
= WebMessageBodyStyle.Wrapped,
        ResponseFormat = WebMessageFormat.Json)]
        public string GetTransformValue(string inputParam)
        {
            // Change the value of the incoming parameter.
            var result = inputParam + " changed!";
            return result;
        }
    }
}
```

Integration with Web-To-Object mechanism.

Web-to-Object is a mechanism of implementation of simple one-way integrations with bpm'online. It enables you to create records of the bpm’online sections (leads, cases, orders, etc.) by sending required data to the Web-to-Object service.

More common cases of using Web-to-Object service:

- Bpm’online integration with custom landings and web forms. The service call is executed from a specifically configured user web page (lending) after the visitor sends the data of the filled form.
- Integration with external systems that are involved in creation of bpm’online objects.

More information about the Web-To-Object can be found in the “**Web-To-Object. Using landings and web-forms**

Key benefits and integration options

- Data transfer with JSON.
- Easy creation of bpm'online objects.
- To access you need only URL of the service and Id.
- Required only basic knowledge for development.

Disadvantages

- Data transfer only in the bpm’online.
- Limited number of objects to use. Service needs to be modified to use custom objects.

Case example

To use service, send the POST query by the address:

```
[Path to bpm'online
application]/0/ServiceModel/GeneratedObjectWebFormService.svc/SaveWebFormObjectData
```

Query content type – application/json. Insert required cookies and a JSON object that contains data of the web form, to the query content. Example of the JSON object that contains data for creation of a new contact:

```
{
    "formData":{
        "formId":"d66ebbf6-f588-4130-9f0b-0ba3414dafb8",
        "formFieldsData":[
            {"name":"Name","value":"John Smith"},
            {"name":"Email","value":"j.smith@bpmoline.com"},
            {"name":"Zip","value":"00000"},
            {"name":"MobilePhone","value":"0123456789"},
            {"name":"Company","value":"bpmonline"},
            {"name":"Industry","value":""},
            {"name":"FullJobTitle","value":"Sales Manager"},
            {"name":"UseEmail","value":""},
            {"name":"City","value":"Boston"},
            {"name":"Country","value":"USA"},
            {"name":"Commentary","value":""},
            {"name":"BpmHref","value":"http://localhost/Landing/"},
            {"name":"BpmSessionId","value":"0ca32d6d-5d60-9444-ec34-5591514b27a3"}
        ]
    }
}
```

Integration of third-party sites via iframe

The most simple way to integrate external solutions to bpm'online. The third-party web application can be implemented to bpm’online with the iframe HTML element. This enables to view third-party web resources (web pages, video, etc.) from bpm’online. Examples of integration via iframe can be found in the “**Integration of third-party sites via iframe**” and “Developing an advanced marketplace application” (Marketplace development documentation).

Key benefits and integration options

- Convenience of viewing the third-party web resources directly from the bpm’online.
- Requires only basic skills for development.

Disadvantages

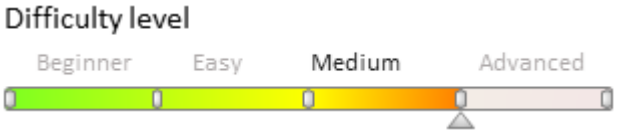
- Needs to be modified for data transfer (or use another integration method).
- Some sites prohibit uploading of their pages into the iframe element.

Case example

An example of the source code of the view model schema of bpm’online edit page with the implemented iframe element is given below. On the page, the iframe element displays a web site which URL is specified in the object associated with the page. Detail description of this example can be found in the “Developing an advanced marketplace application” article. Another approach is described in the “**Integration of third-party sites via iframe**” article.

```
define("tsaWebData1Page", [], function() {
    return {
        entitySchemaName: "tsaWebData",
        diff: /**SCHEMA_DIFF*/[
            // ...
            // A container with an embedded HTML iframe element.
            {
                "operation": "insert",
                "name": "IFrameStat",
                "parentName": "TabData",
                "propertyName": "items",
                "values": {
                    "id": "testiframe",
                    "itemType": Terrasoft.ViewItemType.CONTAINER,
                    "selectors": {"wrapEl": "#stat-iframe"},
                    "layout": { "colSpan": 24, "rowSpan": 1, "column": 0, "row": 0 },
                    "html": "<iframe id='stat-iframe' class='stat-iframe'
width='100%' height='550px'" +
                        "style = 'border: 1px solid silver;'></iframe>",
                    "afterrerender": {
                        "bindTo": "addUrl"
                    }
                }
            }
        ]/**SCHEMA_DIFF*/,
        methods: {
            // The event handler for the full data load.
            onEntityInitialized: function() {
                this.callParent(arguments);
                this.addUrl();
            },
            // The method of adding a URL to an HTML iframe element.
            addUrl: function() {
                var iframe = Ext.get("stat-iframe");
                if (!iframe) {
                    window.console.error("The tab with iframe element was not
found");
                    return;
                }
                var siteName = this.get("tsaName");
                if (!siteName) {
                    window.console.error("The website name was not provided");
                    return;
                }
                var url = "https://www.similarweb.com/website/" + siteName;
                this.set("tsaURL", url);
                iframe.dom.src = url;
            }
        }
    };
});
```

Authenticating external requests to bpm'online services



Introduction

A critical part of most web applications is identifying the users and managing their access to the application’s resources. Authentication is the process of confirming the user’s identity. To pass the authentication, the user must prove that the login attempt is made by this particular user. Usually, the identity proof consists of the user’s credentials: login and password.

All external requests to web services must be authenticated. Bpm’online supports the following authentication methods:

- Anonymous authentication.
- Basic authentication.
- Authentication via Cookies (the “Form-based authentication”).

Advantages and disadvantages of different methods of authentication are available in table 1.

Table 1. Authentication type comparison

Authentication type	Advantages	Disadvantages	Usage
Anonymous	Best performance. Does not require user account management.	Does not identify individual users.	Access to bpm’online public functions, such as the login page, logo, certain web services.
Basic	Widely used. Works with proxy servers. Identifies individual users.	Not secure without SSL/TLS.	Only access to EntityDataService.svc (OData).
Forms (Cookies)	Additional attributes for user credentials. Identifies individual users.	Not secure without SSL/TLS.	Most of bpm’online resources and web services.

Details on each method are available below.

Anonymous authentication

Anonymous authentication provides user access to the publicly available functions of the web application without the need to enter login credentials. From the technical perspective, the authentication is not performed, since the user does not have to provide username and password. Instead, IIS provides Windows previously saved authentication data for a special user.

Anonymous authentication is performed on the IIS level and is enabled by default. When the anonymous authentication is used, IIS does not require any other authentication schemes, provided the corresponding NTFS access permissions were granted for the resource.

Advantages

- Provides the best performance.
- Does not require user account management.

Disadvantages

- Does not identify users individually.

Usage

In bpm'online, anonymous authentication is used for accessing resources that are provided to all users, without authentication. Such resources include the login page, website logo, several web services (for example, AuthService.svc, UserService.svc, etc.).

Basic authentication

Basic authentication is a part of HTTP specification. This is a standard authentication method via HTTP headings. User credentials (username and password) in Base64 are added to the heading of the request to the service. Basic authentication is also performed on the IIS level.

ATTENTION

Basic authentication is not a secure authentication method, since the data are transfered openly. Use this authentication method only when interacting with the system via SSL (HTTPS) protocol.

To ensure security during the data transfer, authenticate external requests to bpm'online via **AuthService.svc**.

ATTENTION

To use Basic authentication in an application integrated with bpm'online, disable **protection from CSRF attacks**.

Advantages

- This authentication method is part of HTTP 1.0 specification and is widely used.
- Can perform authentication through proxy servers.
- Identifies users individually.

Disadvantages

- Not secure without SSL/TLS.

Usage

ATTENTION

Using Basic authentication, you can authenticate users only in the EntityDataService.svc, which can integrate with bpm'online via the **OData** protocol. It is recommended to **use AuthService.svc and corresponding Cookies** for other external requests.

Form-based authentication (Cookies)

ASP.NET has additional authentication methods that are performed only after IIS authentication (usually, anonymous).

One of the additional authentication methods implemented in ASP.NET is Form-based authentication (also known as Cookie-based authentication).

The Form-based authentication provider enables receiving user account data sent via POST request (for example, using an HTML or AJAX form). The user provides username and password for authentication directly to the web application. After successful authentication, the application provides the user special cookies that the user must add to the subsequent requests. If the request to a protected resource does not contain cookies, the application redirects the user to the login page. For more on the Form-based authentication, please see a separate article.

Bpm'online's Form-based authentication uses the **AuthService.svc** web service.

Advantages

- In addition to login and password, Form-based authentication enables using other attributes of user accounts, such as email address.
- Identifies users individually.

Disadvantages

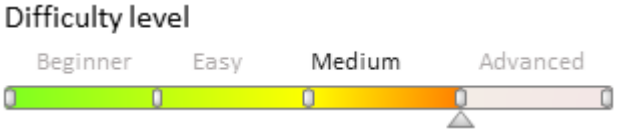
- Can be subject to attacks using cookie lifespan, unless SSL/TLS is used.

Usage

This authentication method is used for accessing most of bpm’online resources and pages.

> ⚠ ATTENTION
>
> Starting with version 7.10, bpm’online has a mechanism for protection from CSRF attacks. To enable the protection, make additional changes to the integration processes that use **DataService** or **OData** (see "**Protection from CSRF attacks during integration with bpm'online**").

The AuthService.svc authentication service



To pass the authentication, call the *Login()* method of the AuthService.svc service. The service request string is as follows:

```
http(s)://[bpm'online application address]/ServiceModel/AuthService.svc/Login
```

Example:

```
https://mycompany.bpmonline.com/ServiceModel/AuthService.svc/Login
```

Service request parameters:

- Method: POST
- ContentType: application/json

The request must pass bpm’online user credentials. The credentials are passed in the form of a JSON object with the following properties.

- UserName: Bpm’online user name
- UserPassword: Bpm’online user password

The titles of the reply to the POST request contain authentication cookies that must be saved on the client side or a client PC and used for future requests to bpm’online web services.

The reply also contains a JSON object of the authentication status. The primary properties of the returned JSON object are available in table 1.

Table 1. General properties of the authentication status JSON object

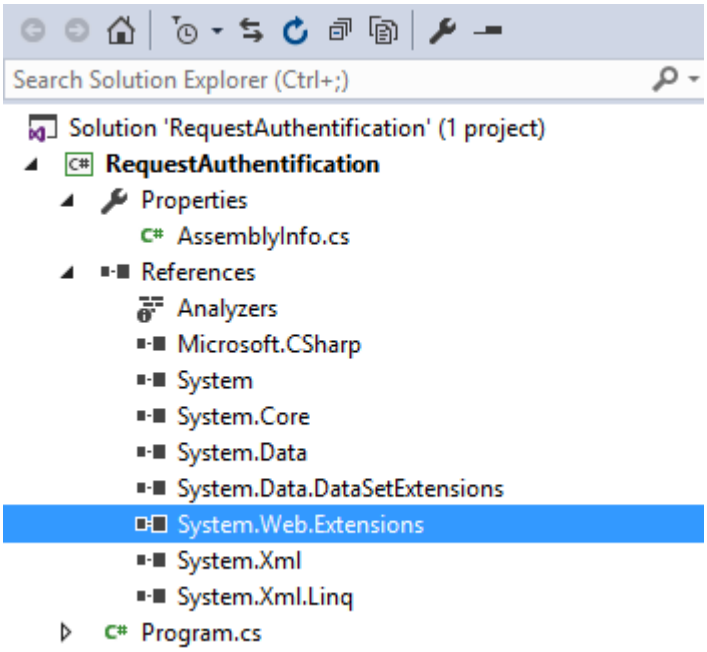
Property	Description
Code	Authentication status code. The authentication is successful if the value is “0”. Otherwise the authentication has failed.
Message	The message that contains the reason for failing the authentication.
Exception	The object that contains a detailed description of an exception that caused the authentication to fail.

An example of calling the AuthService.svc

This example contains an implementation of a C# console application that creates a request to the AuthService.svc for user authentication. User credentials are passed to the *TryLogin()* method as incoming parameters “userName” and “userPassword”. The method returns *true* upon successful authentication and *false* if the authentication has failed. A message with the reason for failed authentication will be sent to console.

To implement this example, create a simple console C# application in the Visual Studio: “RequestAuthentification” Add the *System.Web.Extensions.dll* system library (Fig. 1) to the dependencies (References) of the Visual Studio project. This library is needed for conversion of the authentication status JSON object from a string to a C# object (de-serialization).

Fig. 1. Visual Studio project dependencies



Add the following program code to the Program.cs file of the created application:

```
using System;
using System.IO;
using System.Net;

namespace RequestAuthentification
{
    // Auxiliary class for de-serialization of the JSON object from the HTTP reply.
    class ResponseStatus
    {
        public int Code { get; set; }
        public string Message { get; set; }
        public object Exception { get; set; }
        public object PasswordChangeUrl { get; set; }
        public object RedirectUrl { get; set; }
    }

    // Primary class of the application.
    class Program
    {
        // HTTP address of the application.
        private const string baseUri = "http://mybpmonlineapp.com";
        // Container for Cookie authentication in bpm'online. Must be used in
subsequent requests.
        // This is the most important resulting object.
        // The rest of the functions in this example are developed for implementation
of its properties.
```

```
        public static CookieContainer AuthCookie = new CookieContainer();
        // A request string to the "Login" method of the "AuthService.svc" service.
        private const string authServiceUri = baseUri +
@"/ServiceModel/AuthService.svc/Login";

        // Performs user authentication request.
        public static bool TryLogin(string userName, string userPassword)
        {
            // Creating an instance of the authentication service request.
            var authRequest = HttpWebRequest.Create(authServiceUri) as
HttpWebRequest;
            // Defining the request's method.
            authRequest.Method = "POST";
            // Defining the request's content type.
            authRequest.ContentType = "application/json";
            // Enabling the use of cookie in the request.
            authRequest.CookieContainer = AuthCookie;

            // Placing user credentials to the request.
            using (var requestStream = authRequest.GetRequestStream())
            {
                using (var writer = new StreamWriter(requestStream))
                {
                    writer.Write(@"{
                    ""UserName"":""" + userName + @""",
                    ""UserPassword"":""" + userPassword + @"""
                    }");
                }
            }

            // Auxiliary object where the HTTP reply data will be de-serialized.
            ResponseStatus status = null;
            // Getting a reply from the server. If the authentication is successful,
cookie will be placed to the AuthCookie property.
            // These cookies can be used for subsequent requests.
            using (var response = (HttpWebResponse)authRequest.GetResponse())
            {
                using (var reader = new StreamReader(response.GetResponseStream()))
                {
                    // De-serialization of the HTTP reply to an auxiliary object.
                    string responseText = reader.ReadToEnd();
                    status = new
System.Web.Script.Serialization.JavaScriptSerializer().Deserialize<ResponseStatus>
(responseText);
                }

            }

            // Checking authentication status.
            if (status != null)
            {
                // Authentication is successful.
                if (status.Code == 0)
                {
                    return true;
                }
                // Authentication is unsuccessful.
                Console.WriteLine(status.Message);
            }
            return false;
        }
```

```
        // Application login method.
        static void Main(string[] args)
        {
            // Calling authentication method.
            Console.WriteLine("Is authentication successful?: {0}", TryLogin("User
1", "User 1"));
            Console.WriteLine("Press ENTER to close...");
            Console.ReadLine();
        }
    }
}
```

The authentication will be successful of correct user credentials were entered on calling the *TryLogin()* method (Fig. 2). If the credentials were invalid, an error message will be displayed (Fig. 2).

Fig. 2. Successful authentication

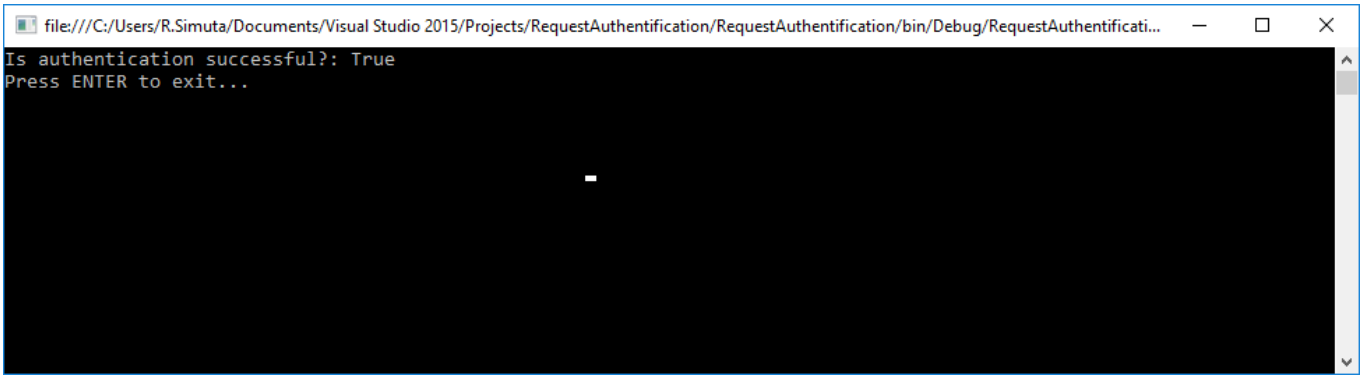
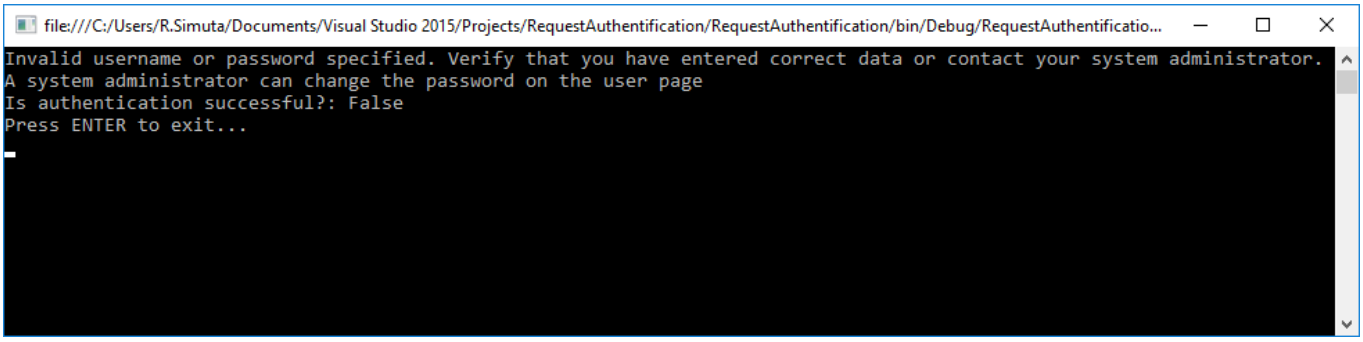
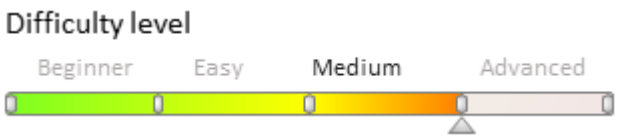


Fig. 3. Failed authentication



The AuthService.svc authentication service

Protection from CSRF attacks during integration with bpm'online



Starting with version 7.10, bpm'online has a mechanism for protection from CSRF attacks. To enable the protection, make additional changes to the integration processes that use **DataService** or **OData**.

During the integration with third-party applications, an authentication via the AuthService.svc service must be passed. After the authentication, the AuthService returns an authentication cookie that must be added to the query, as well as a cookie with a CSRF token that must be placed at the query title.

Examples of using authentication cookies are available in the "**Authenticating external requests to bpm'online services**", "**OData**" and "**DataService web service**" articles.

Previously, to protect against CSRF attacks, a method had to be created that was called as a response to a

SendingRequest context instance event (creating a new *HttpWebRequest* instance). User authentication and cookie transfer would be executed in this method. After the implementation of protection from CSRF attacks, adding of a CSRF token must be implemented in this method:

```
static void OnSendingRequestCookie(object sender, SendingRequestEventArgs e)
{
    // Calling method of the "LoginClass" class, that implements user authentication.
    LoginClass.TryLogin("BPMUserName", "BPMUserPassword");
    var req = e.Request as HttpWebRequest;
    // Adding authentication cookie to the data reaquest.
    req.CookieContainer = LoginClass.AuthCookie;
    e.Request = req;
    // Adding a CSRF token to the request title.
    CookieCollection cookieCollection = AuthCookie.GetCookies(new
Uri(authServiceUri));
    string csrfToken = cookieCollection["BPMCSRF"].Value;
    ((HttpWebRequest)e.Request).Headers.Add("BPMCSRF", csrfToken);
}
```

NOTE

An example of using the *OnSendingRequestCookie()* method is available in the “**Working with bpm'online objects over the OData protocol WCF-client**” article.

Attention!

Protection from CSRF attacks works only if the Form-authentication is used.

How to disable CSRF attack protection

To disable protection from CSRF attacks, disable the *UseCsrfToken* setting in the \Web.Config and .\Terrasoft.WebApp\Web.Config files:

```
<add key="UseCsrfToken" value="true" />
```

You can also specify service methods which will be called without checking the availability of the CSRF token. Use the *DisableCsrfTokenValidationForPaths* setting in the .\Web.Config.

Example of disabling CSRF protection for two different methods of different services:

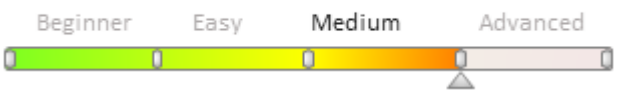
```
<add key="DisableCsrfTokenValidationForPaths"
value="/MsgUtilService.svc/Ping,/AuthService.svc/Login" />
```

Example of disabling CSRF protection for one service completely:

```
<add key="DisableCsrfTokenValidationForPaths" value="/ServiceModel/service_name" />
```

DataService web service

Difficulty level



General information

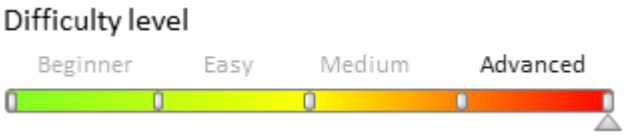
The DataService web service is used for handling the requests from the bpm'online client side.

The full list and description of the DataService data contracts is displayed on table 1.

Table 1. The bpm'online application DataService services

Service	Description
SchemaDesignerRequest	Schema designer request class Not recommended to use.
EntitySchema	Object schema class. Not recommended to use.
ClientUnitSchema	Client schema class. Not recommended to use.
RemoveEntitySchemaRequest	Remove object schema request. Not recommended to use.
RemoveClientUnitSchemaRequest	Remove client schema request. Not recommended to use.
EntitySchemaRequest	Receive object schema instance request. Not recommended to use.
ClientUnitSchemaRequest	Receive client object schema instance request. Not recommended to use.
ProcessUserTaskSchemaRequest	Receive business process user action schema request. Not recommended to use.
UpdatePackageSchemaDataRequest	Receive package schema data update request. Not recommended to use.
ProcessSchemaRequest	Receive process schema instance request. Not recommended to use.
ContractProcessSchema	Process schema contract class. Not recommended to use.
RemoveProcessSchemaRequest	Remove process schema request. Not recommended to use.
InsertQuery	Add section record query class.
UpdateQuery	Update section record query class.
DeleteQuery	Delete section record query class.
SelectQuery	Select section record query class.
BatchQuery	Package query class.
UserProfile	User profile class. Not recommended to use.
QueryProfile	Query profile class. Not recommended to use.
QuerySysSettings	Receive system settings list request. Not recommended to use.
PostSysSettingsValue	Set system setting value class. Not recommended to use.
PostSysSettingsValues	Set system setting values class. Not recommended to use.
Filters	Filter class.
QueryModuleDescriptors	Receive module descriptors query class. Not recommended to use.
ClientLoggerDataContract	Client log data class. Not recommended to use.
PostClientLog	Client log post class. Not recommended to use.
UploadFile	File upload class. Not recommended to use.
GetTelephonyConfig	Receive telephony configuration settings class. Not recommended to use.
InsertSysSettingRequest	Add system setting query class. Not recommended to use.
UpdateSysSettingRequest	Edit system setting query class. Not recommended to use.
DeleteSysSettingRequest	Delete system setting query class. Not recommended to use.
GetTests	Receive all Unit tests class. Not recommended to use.
RunTests	Run Unit tests class. Not recommended to use.

DataService. Adding records



General information

The bpm'online DataService web service is a RESTfull service. RESTful is a quite simple information management interface that doesn't use any additional internal layers, i.e., the data doesn't need to be converted to any third-party format, such as XML. In a simple RESTful service, each record is uniquely identified by a global identifier such as URL. Each URL, in turn, has a strictly specified format. However, this service is not always convenient for transferring large amounts of data.

With the use of the DataService, the data can be automatically configured in various data formats such as XML, JSON, HTML, CSV, and JSV. The data structure is determined by data contracts. A complete list of data contracts used by the DataService, can be found in the "**DataService web service**" article.

InsertQuery data contract

The *InsertQuery* data contract is used to add records to sections. The data is transferred to the DataService via HTTP by using the POST request with the following URL:

```
// URL format of the POST query to add data to DataService.
http(s)://[Bpm'online application address]/[Configuration number]/dataservice/[Data
fromat]/reply/InsertQuery
// URL example for the POST query to add data to DataService.
http(s)://example.bpmonline.com/0/dataservice/json/reply/InsertQuery
```

The InsertQuery data contract has a hierarchical structure with multiple nesting levels. In the bpm'online application server part, the InsertQuery data contract is represented by the *InsertQuery* class of the *Terrasoft.Nui.ServiceModel.DataContract* namespace of the *Terrasoft.Nui.ServiceModel.dll* class library. However, for simplicity, the hierarchical structure of the *InsertQuery* data contract is conveniently presented as a JSON format object:

```
{
    "RootSchemaName":"[Root object schema name]",
    "OperationType":[Record operation type],
    "ColumnValues":{
        "Items":{
            "Added column name":{
                "ExpressionType":[Expression type],
                "Parameter":{
                    "DataValueType":[Data type],
                    "Value":"[Column value]"
                }
            }...
        }
    }
}
```

The basic properties of the InsertQuery class and their possible values are presented in table 1.

Table 1. *InsertQuery class properties*.

Property	Description
RootSchemaName	A string containing the name of the root object schema of the added record.
OperationType	Operation type is set by the *QueryOperationType namespace Terrasoft.Nui.ServiceModel.DataContract* namespace enumeration value. For the *InsertQuery* the*QueryOperationType.Insert* value is set. *QueryOperationType* enumeration values:

Select	0
Insert	1
Update	2
Delete	3
Batch	4

ColumnValues	Contains a collection of column values of the added record. Its *ColumnValues* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace.

The *ColumnValues* class has a single *Items* property that is defined as a collection of the *Dictionary<string, ColumnExpression>* keys and values. The key is a string with the added column title, and the value is the object with the *ColumnExpression* type defined in the*Terrasoft.Nui.ServiceModel.DataContract* namespace. The basic properties of the ColumnExpression class used when adding records, are given in table 2.

Table 2. ColumnExpression class main properties

Property	Description
ExpressionType	The expression type that defines the value that will be contained in the added column. Set by the *EntitySchemaQueryExpressionType* enumeration of the *Terrasoft.Core.Entities* namespace defined in the *Terrasoft.Core* class library. For the *InsertQuery* the *EntitySchemaQueryExpressionType.Parameter* value is set. *EntitySchemaQueryExpressionType* enumeration type: *SchemaColumn* 0 *Function* 1 *Parameter* 2 *SubQuery* 3 *ArithmeticOperation* 4
Parameter	Defines the value that will be contained in the added column. Its *Parameter* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace.

The *Parameter* class has multiple properties, two of which are used to add records (table 3).

Table 3 Parameter class main properties

Property	Description
DataValueType	The data value type that defines the value that will be contained in the added column. Set by the *DataValueType* enumeration value of the*Terrasoft.Nui.ServiceModel.DataContract* namespace. *DataValueType* enumeration type: *Guid* 0 *Text* 1 *Integer* 4 *Float* 5 *Money* 6 *DateTime* 7 *Date* 8 *Time* 9 *Lookup* 10

Enum	11
Boolean	12
Blob	13
Image	14
ImageLookup	16
Color	18
Mapping	26

Value	The object that contains the added column value.

Creating a record using a third-party application example

Case description

You need to create a console application that will add the following data to the [Contact] section using the DataService service:

- Full name — John Best
- Full job title — Developer
- Business phone — +12 345 678 00 00.

Case implementation algorithm

1 Create and configure a C# console application project

Using the Microsoft Visual Studio (version 2017 and up) development environment, create a Visual C# console application project and name it *DataServiceInsertExample*. The [Target framework] project property must be set to .NET Framework 4.7.

In the References section of the project you need to add dependencies of the following libraries:

- *System.Web.Extensions.dll* is a class library included in the .NET Farmework
- *Terrasoft.Core.dll* is a main class library of the application server kernel. Can be found by the following path: [Directory with the installed application]\Terrasoft.WebApp\bin\Terrasoft.Core.dll
- Terrasoft.Nui.ServiceModel.dll class library the application services. Can be found by the following path: [Directory with the application installed]\Terrasoft.WebApp\bin\Terrasoft.Nui.ServiceModel.dll.

Add *using* directives to the application source code file:

```
using System;
using System.Text;
using System.IO;
using System.Net;
using System.Collections.Generic;
using Terrasoft.Nui.ServiceModel.DataContract;
using Terrasoft.Core.Entities;
using System.Web.Script.Serialization;
```

2 Add field declarations and constants to the application source code

To access the DataService features, you must add the following fields and constants to the application source code:

```
// Main bpm'online URL. Has to be changed to a custom one.
private const string baseUri = @"http://example.bpmonline.com";
// Query string to the Login method of the AuthService.svc service.
private const string authServiceUri = baseUri +
@"/ServiceModel/AuthService.svc/Login";
```

```
// InsertQuery query path string.
private const string insertQueryUri = baseUri +
@"/0/DataService/json/reply/InsertQuery";
// Bpm'online cookie authentication.
private static CookieContainer AuthCookie = new CookieContainer();
```

Three string constant fields that are used to carry out the authentication requests and requests to add data are declared here. The authentication data will be stored in the *AuthCookie* field.

3 Add a method that performs authentication in the bpm'online application

You need to authenticate the newly created application to access the DataService web service.

The algorithm and an implementation example of a method that performs a request to the *AuthService.svc* service for user authentication can be found in the **Authenticating external requests to bpm'online services** article.

4 Add the direct implementation of the add record query

As the previously declared *insertQueryUri* constant contains the path for sending data in JSON format, then the data sent must be pre-configured in the form of a string containing a description of the JSON object corresponding to the *InsertQuery* data contract. This can be done directly in a lowercase variable but it is much easier and safer to create an instance of the *InsertQuery* class, fill its properties, and then serialize it to a string. This can be done by adding the following source code:

```
// Query class instance.
var insertQuery = new InsertQuery()
{
    // Root schema name.
    RootSchemaName = "Contact",
    // Added column values collection.
    ColumnValues = new ColumnValues()
};
// Expression class instance of the object schema query.
// Used to configure the [Full name] column.
var columnExpressionName = new ColumnExpression
{
    // Query object schema expression type – parameter.
    ExpressionType = EntitySchemaQueryExpressionType.Parameter,
    // Query expression parameter.
    Parameter = new Parameter
    {
        // Parameter value.
        Value = "John Best",
        // Parameter data type – string.
        DataValueType = DataValueType.Text
    }
};

// Expression class instance of the object schema query.
// Used to configure the [Business phone] column.
var columnExpressionPhone = new ColumnExpression
{
    ExpressionType = EntitySchemaQueryExpressionType.Parameter,
    Parameter = new Parameter
    {
        Value = "+12 345 678 00 00",
        DataValueType = DataValueType.Text
    }
};
// Expression class instance of the object schema query.
// Used to configure the [Job title] column.
var columnExpressionJob = new ColumnExpression
{
```

```
    ExpressionType = EntitySchemaQueryExpressionType.Parameter,
    Parameter = new Parameter
    {
        // The "Developer" record GUID of the [Job title] system lookup.
        // Change to the record GUID in bpm'online.
        Value = "11D68189-CED6-DF11-9B2A-001D60E938C6",
        // Parameter data type — unique ID.
        DataValueType = DataValueType.Guid
    }
};
// Query column collection initialization.
insertQuery.ColumnValues.Items = new Dictionary<string, ColumnExpression>();
// Adding query expressions to the added column collection.
// The [Full name] column.
insertQuery.ColumnValues.Items.Add("Name", columnExpressionName);
// The [Business phone] column.
insertQuery.ColumnValues.Items.Add("Phone", columnExpressionPhone);
// The [Job title] column.
insertQuery.ColumnValues.Items.Add("Job", columnExpressionJob);
// Class instance serialization of the JSON string adding query.
var json = new JavaScriptSerializer().Serialize(insertQuery);
```

NOTE

In this example, to reduce the article size, a unique identifier of the "Developer" record of the [Position] lookup is represented as a string literal. It can be defined, for example, using the *SelectQuery* query with the job title filter.

In the final step you must perform POST query to the DataService service. To do this, create an instance of the *HttpWebRequest* class, fill in its properties, attach a previously created string with the JSON object to a request, and then execute and process the result of the query to the DataService service. To do this, add the following source code:

```
// Converting JSON string object to a byte array.
byte[] jsonArray = Encoding.UTF8.GetBytes(json);
// Creating HTTP query instance.
var insertRequest = HttpWebRequest.Create(insertQueryUri) as HttpWebRequest;
// Defining method query.
insertRequest.Method = "POST";
// Defining query content type.
insertRequest.ContentType = "application/json";
// Adding the previously received authenticated cookie to the receiveing data query.
insertRequest.CookieContainer = AuthCookie;
// Define query content length.
insertRequest.ContentLength = jsonArray.Length;
// Adding a JSON object to the query.
using (var requestStream = insertRequest.GetRequestStream())
{
    requestStream.Write(jsonArray, 0, jsonArray.Length);
}
// Executing the HTTP query and receiving answer from server.
using (var response = (HttpWebResponse)insertRequest.GetResponse())
{
    // Displaying answer in console.
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        Console.WriteLine(reader.ReadToEnd());
    }
}
```

Creating a record in bpm'online example

Case description

Add to the [Contact] section a button that when clicked invokes a method, using the *InsertQuery* class that adds a record with the following data:

- Full name — John Best
- Full job title — Developer
- Business phone — +12 345 678 00 00.

Case realization

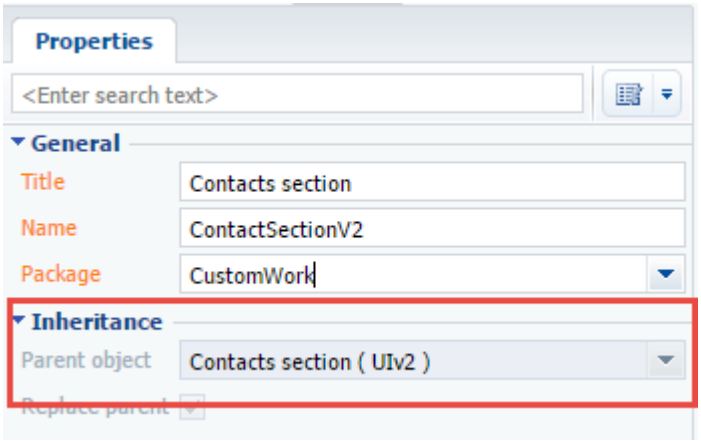
Case implementation algorithm

1 Add button to the [Contacts] section

The process of adding buttons to sections is described in the "**How to add a button to a section**" article.

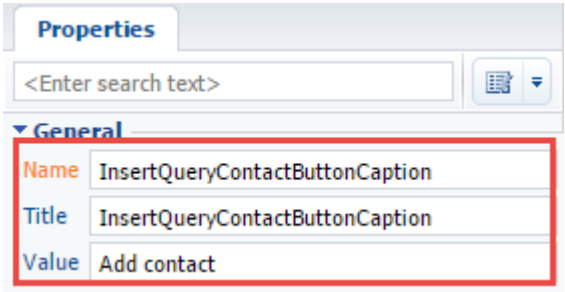
For this particular case, you need to create a replacement client module of the [Contacts] section (Fig. 1).

Fig. 1 Replacement client module properties.



In the created client schema, add the*InsertQueryContactButtonCaption* localizable string, and set the "Add contact" value to it (Fig. 2).

Fig. 2 Localizable string properties



Add a configuration object with the button location settings to the *diff* array.

```
diff: /**SCHEMA_DIFF*/[
    // Metadata to be added to the custom button section.
    {
        // The element is added to the page.
        "operation": "insert",
        // Parent interface element name to which the button is added.
        "parentName": "ActionButtonsContainer",
        // The button is added to the interface element collection
        // of the parent element (its metaname is specified in parentName).
        "propertyName": "items",
        // Added button metaname.
```

```
        "name": "InsertQueryContactButton",
        // Additional element properties.
        "values": {
            // Added element type — button.
            itemType: Terrasoft.ViewItemType.BUTTON,
            // Binding button caption to the localizable schema string.
            caption: { bindTo: "Resources.Strings.InsertQueryContactButtonCaption" },
            // Binding method of processing the button click.
            click: { bindTo: "onInsertQueryContactClick" },
            "layout": {
                "column": 1,
                "row": 6,
                "colSpan": 1
            }
        }
    }
]/**SCHEMA_DIFF*/
```

2 Add the processing method for the button click event

In order for a record with the necessary data to be added when a button created in the section is clicked, add the following method to the *methods* section of the replacement client schema:

```
methods: {
    // Method of processing the button click.
    onInsertQueryContactClick: function() {
        // Creating the Terrasoft.InsertQuery class instance.
        var insert = Ext.create("Terrasoft.InsertQuery", {
            // Root schema name.
            rootSchemaName: "Contact"
        });
        // Setting the Terrasoft.ParameterExpression value-parameters.
        // A value-parameter instance is created and added to the column value
collection.
        // Creating a value-parameter instance for the [Job title] column.
        insert.setParameterValue("Name", "John Best", Terrasoft.DataValueType.TEXT);
        // Creating a value-parameter instance for the [Business phone] column.
        insert.setParameterValue("Phone", "+12 345 678 00 00",
Terrasoft.DataValueType.TEXT);
        // Creating a value-parameter instance for the [Job title] column.
        insert.setParameterValue("Job", "11D68189-CED6-DF11-9B2A-001D60E938C6",
Terrasoft.DataValueType.GUID);
        // Data update query.
        insert.execute(function(response) {
            // Displaying server answer
            window.console.log(response);
        });
        // Updating list data.
        this.reloadGridData();
    }
}
```

> NOTE
>
> Unlike the previous example, authorization is not required because the code is executed directly in the bpm'online application.

The implementation of the *InsertQuery* class for the client part of the application kernel is different from the implementation of the *InsertQuery* in its back end. So, to create the parameters, the *setParameterValue* method is used, and for the query execution — the *execute* method. Learn about all the available properties and methods of the *InsertQuery* class implemented in the kernel client part in the API documentation.

DataService. Reading records

```
        "CacheLevel":[Caching level],
        "CacheGroup":[Caching group],
        "CacheItemName":[Record key in repository]
    },
    "IsPageable":[Indicates page-by-page],
    "IsDistinct":[Indicates uniqueness],
    "RowCount":[Number of selected records],
    "ConditionalValues":[Conditions for building a pageable query],
    "IsHierarchical":[Indicates hierarchical data selection],
    "HierarchicalMaxDepth":[Maximum nesting level of the hierarchical query],
    "HierarchicalColumnName":[Column name used to create hierarchical query],
    "HierarchicalColumnValue":[Initial value of hierarchical column],
    "Filters":[Filters]
    }
}
```

Primary properties of the SelectQuery class and their possible values are available in table 1.

Table 1. SelectQuery class properties

Property	Type	Notes
RootSchemaName	*string*	String that contains root schema name of the added record object.
OperationType	*QueryOperationType*	Type of write operation. Specified as a QueryOperationType enumeration value of the Terrasoft.Nui.ServiceModel.DataContract name space. The QueryOperationType.Select value is set for SelectQuery. Values of the QueryOperationType enumeration: *Select* 0 *Insert* 1 *Update* 2 *Delete* 3 *Batch* 4
Columns	*SelectQueryColumns*	Contains a collection of the record columns being read. It has the SelectQueryColumns type defined in the Terrasoft.Nui.ServiceModel.DataContract name space. It must be configured if the *AllColumns* checkbox is set to *false* and a set of specific root schema columns, which does not include the [Id] column, is required.
AllColumns	*bool*	Indicates if all columns are selected. If the value is set to true, all columns of the root schema will be selected by the query.
ServerESQCache Parameters	*ServerESQCache Parameters*	Parameters of *EntitySchemaQuery* caching on server. The *ServerESQCacheParameters* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space.
IsPageable	*bool*	Indicates whether the data is selected page-by-page.
IsDistinct	*bool*	Indicates whether duplicates must be eliminated in the resulting data set.
RowCount	*int*	Number of selected strings. By default, the value is -1, i.e. all strings are selected.
ConditionalValues	*ColumnValues*	Conditions of creating a page-by-page query. The *ColumnValues* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space.
IsHierarchical	*bool*	Indicates whether the data is selected hierarchically.

HierarchicalMaxDepth	*int*	Maximum nesting level of a hierarchical query.
hierarchicalColumnName	*string*	Name of the column used for creating a hierarchical query.
hierarchicalColumnValue	*string*	Initial value of hierarchical column from which the hierarchy will be built.
Filters	*Filters*	Collection of query filters. The *Filters* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space.
ColumnValues	*ColumnValues*	Contains collection of column values for the added record. The *ColumnValues* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space.

The *SelectQueryColumns* class has a single Items property, defined as a collection of keys and values *Dictionary<string, SelectQueryColumn>*. The key is the string with the name of the added column. The value is an instance of the *SelectQueryColumn* class, defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. The properties of the *SelectQueryColumn* are available in the table 2.

Table 2. SelectQueryColumn class properties

Property	Type	Notes
OrderDirection	*OrderDirection*	Sorting order. Specified with a value from the *OrderDirection* enumeration of the *Terrasoft.Common* name space defined in the *Terrasoft.Common* class library.
OrderPosition	*int*	Sets position number in the collection of the query columns, by which the sorting is done.
Caption	*string*	Column title.
Expression	*ColumnExpression*	Property that defines expression of the type of selected column.

The *ColumnExpression* class defines expression that sets the type of the schema column. The class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space of the *Terrasoft.Nui.ServiceModel* library. The properties of an instance of this class are filled in depending on the *ExpressionType* property, which sets the expression type. The full list of the *ColumnExpression* class properties is available in table 3.

Table 3. Primary properties of the ColumnExpression class

Property	Type	Notes
ExpressionType	*EntitySchemaQuery ExpressionType*	Type of expression that determines the value that the added column will contain. Specified with a value from the *EntitySchemaQueryExpressionType* enumeration of the *Terrasoft.Core.Entities* name space defined in the *Terrasoft.Core* class library. The *EntitySchemaQueryExpressionType.Parameter* value is set for *InsertQuery*. Values of the *EntitySchemaQueryExpressionType* enumeration: *SchemaColumn* 0 Schema column *Function* 1 Function *Parameter* 2 Parameter *SubQuery* 3 Subquery *ArithmeticOperation* 4 Arithmetic operation
ColumnPath	*string*	Path to the column in relation to the root schema. Rules for building paths are available in the "**The use of EntitySchemaQuery for creation of queries in database**" article.
Parameter	*Parameter*	Determines the value that the added column will contain. The

		Parameter type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space.
FunctionType	*FunctionType*	Function type. Specified with a value from the *FunctionType* enumeration, which is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. Values of the *FunctionType* enumeration: *None* 0 Not defined *Macros* 1 Macro *Aggregation* 2 Aggregate function *DatePart* 3 Part of date value *Length* 4 Length
MacrosType	*EntitySchemaQuery MacrosType*	Macro type. Specified with a value of the *EntitySchemaQueryMacrosType* enumeration, which is defined in the *Terrasoft.Core.Entities* name space.
FunctionArgument	*BaseExpression*	Function argument. Accepts a value if the function is defined with a parameter. The *BaseExpression* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space, is an ancestor for the *ColumnExpresion* class and has the same set of properties.
DatePartType	*DatePart*	Part of date value Specified with a value from the *DatePart* enumeration, which is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. Values of the *DatePart* enumeration: *None* 0 Not defined *Day* 1 Day *Week* 2 Week *Month* 3 Month *Year* 4 Year *Weekday* 5 Week day *Hour* 6 Hour *HourMinute* 7 Minute
AggregationType	*AggregationType*	Aggregate function type. Specified with a value from the *AggregationType* enumeration defined in the *Terrasoft.Common* name space defined in the *Terrasoft.Common* class library.
AggregationEvalType	*AggregationEvalType*	Aggregate function scope. Specified with a value from the *AggregationEvalType* enumeration defined in the *Terrasoft.Common* name space defined in the *Terrasoft.Common* class library.
SubFilters	*Filters*	Collection of subquery filters. The *Filters* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space.

The *Parameter* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. Its properties are available in table 4.

Table 4. Parameter class properties

Property	Type	Notes
DataValueType	DataValueType	Type of data for the value that the added column will contain. Specified as a *DataValueType* enumeration value of the *Terrasoft.Nui.ServiceModel.DataContract* name space. Values of the *DataValueType* enumeration: *Guid* 0 *Text* 1 *Integer* 4 *Float* 5 *Money* 6 *DateTime* 7 *Date* 8 *Time* 9 *Lookup* 10 *Enum* 11 *Boolean* 12 *Blob* 13 *Image* 14 *ImageLookup* 16 *Color* 18 *Mapping* 26
Value	*object*	The object that contains the value of the added column.
ArrayValue	*string[]*	Array of the added column values. Used when serializing arrays and BLOBs.
ShouldSkipConvertion	*bool*	Indicates the need to skip the process of providing the type for the *Value* property.

The *ServerESQCacheParameters* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. Its properties are available in table 5.

Table 5. ServerESQCacheParameters class properties

Property	Type	Notes
CacheLevel	int	Data allocation level in the EntitySchemaQuery cache.
CacheGroup	*string*	Caching group.
CacheItemName	*string*	Repository record key.

The *Filters* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. For details on the properties of this class and its use, please see the "**DataService. Data filtering**" article.

Example of reading records in a third-party application

Case description

Create a console application that uses DataService to read records from the [Contact] section with the following

columns:

- Id
- Full name
- Number of activities – aggregate column that displays the number of activities of this contact.

Case implementation algorithm

1. Create and set up a C# application project

Using the Microsoft Visual Studio development environment (version 2017 and up), create a Visual C# console application project and specify project name, for example, DataServiceSelectExample. Set ".NET Framework 4.7" for the project property [Target framework].

In the References section of the project, add dependencies from the following libraries:

- *System.Web.Extensions.dll* – class library included in .NET Farmework;
- *Terrasoft.Core.dll* – library of base bpm'online server core classes. It can be found using the following path: [Bpm'online setup catalog]\Terrasoft.WebApp\bin\Terrasoft.Core.dll;
- *Terrasoft.Nui.ServiceModel.dll* — application service class library. It can be found using the following path: [Bpm'online setup catalog]\Terrasoft.WebApp\bin\Terrasoft.Nui.ServiceModel.dll;
- *Terrasoft.Common.dll* – library of base bpm'online server core classes. It can be found using the following path: [Bpm'online setup catalog]\Terrasoft.WebApp\bin\Terrasoft.Common.dll.

Add the "using" directives to the application source code file:

```
using System;
using System.Text;
using System.IO;
using System.Net;
using System.Collections.Generic;
using Terrasoft.Nui.ServiceModel.DataContract;
using Terrasoft.Core.Entities;
using System.Web.Script.Serialization;
using Terrasoft.Common;
```

2. Add fields and constants and field declarations to the source code

To access DataService features, add the following fields and constants to the application source code:

```
// Bpm'online primary application URL. Must be replaced with a custom one.
private const string baseUri = @"http://example.bpmonline.com";
// Query string to the Login method of the AuthService.svc service.
private const string authServiceUri = baseUri +
@"/ServiceModel/AuthService.svc/Login";
// SelectQuery path string.
private const string selectQueryUri = baseUri +
@"/0/DataService/json/SyncReply/SelectQuery";
// Bpm'online authentication cookie.
private static CookieContainer AuthCookie = new CookieContainer();
```

Here, three string fields are declared. These fields will be used to form authentication query and read data queries execution paths. Authentication data will be saved in the AuthCookie field.

3. Add method that performs bpm'online application authentication

Authentication is required to enable access of the created application to the DataService.

Both the algorithm and example of implementation method, which contains query to AuthService.svc for user authentication, are available in the "**Authenticating external requests to bpm'online services**" article.

4. Add implementation of the record add query

Because the *selectQueryUri* constant declared earlier contains a path for sending data in the JSON format, sent data

must be configured beforehand as a string that contains a JSON object that corresponds to the *SelectQuery* data contract. This can be done directly in a string variable, although a much more secure and convenient way of doing this would be to create an instance of the *SelectQuery* class, fill out its properties and then serialize it to a string. This can be done with the help of the following source code:

```
// Instance of the query class.
var selectQuery = new SelectQuery()
{
    // Root schema name.
    RootSchemaName = "Contact",
    // Collection of query columns.
    Columns = new SelectQueryColumns()
};
// Expression that specifies the type of [[Full name] column.
var columnExpressionName = new ColumnExpression()
{
    // Expression type – schema column.
    ExpressionType = EntitySchemaQueryExpressionType.SchemaColumn,
    // Path to column.
    ColumnPath = "Name"
};
// Configuring the [Name] column.
var selectQueryColumnName = new SelectQueryColumn()
{
    //Title.
    Caption = "Full name",
    // Sorting order – ascending.
    OrderDirection = OrderDirection.Ascending,
    // Sorting order position.
    OrderPosition = 0,
    // Expression that specifies column type.
    Expression = columnExpressionName
};
// Expression that specifies [Number of activities] column type.
var columnExpressionActivitiesCount = new ColumnExpression()
{
    // Expression type – subquery.
    ExpressionType = EntitySchemaQueryExpressionType.SubQuery,
    // Path to column in relation to root schema.
    ColumnPath = "[Activity:Contact].Id",
    // Function type – aggregation.
    FunctionType = FunctionType.Aggregation,
    // Aggregation type – quantity.
    AggregationType = AggregationType.Count
};
// Configuring the [Number of activities] column.
var selectQueryColumnActivitiesCount = new SelectQueryColumn()
{
    //Title.
    Caption = "Number of activities",
    // Sorting direction – ascending.
    OrderDirection = OrderDirection.Ascending,
    // Sorting order position.
    OrderPosition = 1,
    // Expression, which specifies column type.
    Expression = columnExpressionActivitiesCount
};

// Adding columns to query.
selectQuery.Columns.Items = new Dictionary<string, SelectQueryColumn>()
{
    {
```

```
            "Name",
            selectQueryColumnName
        },
        {
            "ActivitiesCount",
            selectQueryColumnActivitiesCount
        }
};
// Serialization of an instance of query class to add to JSON string.
var json = new JavaScriptSerializer().Serialize(selectQuery);
```

The next step is to execute POST DataService query. To do this, create an instance of the HttpWebRequest class, fill its properties and connect the string with JSON object, created earlier, after which – execute the DataService query and process its result. To do this, add the following source code:

```
// Converting a JSON object string to a byte array.
byte[] jsonArray = Encoding.UTF8.GetBytes(json);
// Creating an instance of HTTP request.
var selectRequest = HttpWebRequest.Create(selectQueryUri) as HttpWebRequest;
// Defining request method.
selectRequest.Method = "POST";
// Defining request content type.
selectRequest.ContentType = "application/json";
// Adding earlier received authentication cookies to a data fetch query.
selectRequest.CookieContainer = AuthCookie;
// Set length for request content.
selectRequest.ContentLength = jsonArray.Length;
// Placing JSON object to request content.
using (var requestStream = selectRequest.GetRequestStream())
{
    requestStream.Write(jsonArray, 0, jsonArray.Length);
}
// Executing HTTP request and getting reply from server.
using (var response = (HttpWebResponse)selectRequest.GetResponse())
{
    // Displaying reply in console.
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        Console.WriteLine(reader.ReadToEnd());
    }
}
```

Example of reading records in bpm'online application

Case description

In the [Contacts] section, add a button which will open the method that will use DataService to read records in the [Contacts] section with the following columns:

- Id
- Full name
- Number of activities – aggregate column, which displays the number of activities of this contact.

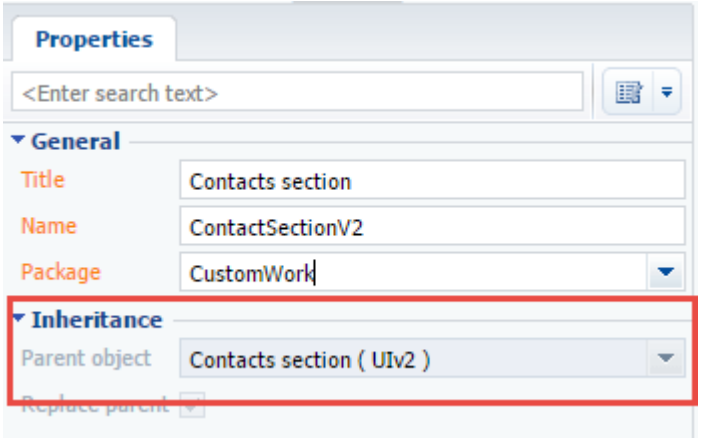
Case implementation algorithm

1. Add a button in the [Contacts] section

The process of adding a button in a section, is covered in the "**How to add a button to a section**".

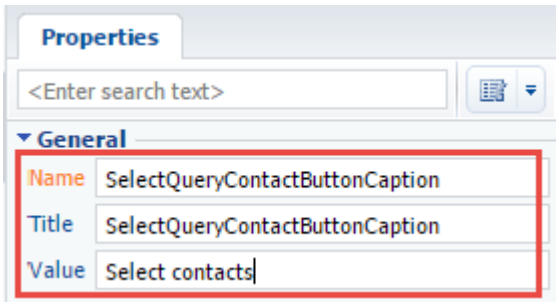
Create a replacing client module of the [Contacts] section (Fig. 1).

Fig. 1. Properties of the replacing client module



In the created client schema, add SelectQueryContactButtonCaption localizable string and set its value to "Select contacts" (Fig. 2).

Fig. 2. Localizable string properties



Add a configuration object with the settings determining the button position to the diff array.

```
//Setup of section button display.
diff: /**SCHEMA_DIFF*/[
    // Metadata for adding a custom button in a section.
    {
        // Indicates that an elementis added on a page.
        "operation": "insert",
        // Meta name of the parent control element where the button is added.
        "parentName": "ActionButtonsContainer",
        // Indicates that the button is added to the control element collection
        // of parent element (meta-name specified in parentName).
        "propertyName": "items",
        // Meta-name of the added button.
        "name": "SelectQueryContactButton",
        // Additional properties of the element.
        "values": {
            // Type of added element - button.
            itemType: Terrasoft.ViewItemType.BUTTON,
            // Binding button title to a schema localizable string.
            caption: { bindTo: "Resources.Strings.SelectQueryContactButtonCaption" },
            // Binding of the button pressing handler method.
            click: { bindTo: "onSelectQueryContactClick" },
            "layout": {
                "column": 1,
                "row": 6,
                "colSpan": 1
            }
        }
    }

]/**SCHEMA_DIFF*/
```

2. Add handler method for the button pressing event

To enable reading the records when the button is clicked, add the following method to the *methods* section of the replacing client schema:

```
methods: {
    // Handler method for button click.
    onSelectQueryContactClick: function() {
        // Creating an instance of the Terrasoft.InsertQuery class.
        var select = Ext.create("Terrasoft.EntitySchemaQuery", {
            // Root schema name.
            rootSchemaName: "Contact"
        });
        // Adding the [Full name] column to query.
        select.addColumn("Name");
        // Adding [Number of activities] aggregate column to a query.
        select.addAggregationSchemaColumn(
            // Path to column in relation to the root schema.
            "[Activity:Contact].Id",
            // Aggregation type — quantity.
            Terrasoft.AggregationType.COUNT,
            // Column title.
            "ActivitiesCount",
            // Aggregation function scope - for all elements.
            Terrasoft.AggregationEvalType.ALL);
        // Update query to server
        // Getting whole collection of records and displaying it in the browser
console.
        select.getEntityCollection(function(result) {
            if (!result.success) {
                // Processing/logging of error.
                this.showInformationDialog("Data query error");
                return;
            }
            // Displayed message.
            var message = "";
            // Analyzing resulting collection and generating displayed message.
            result.collection.each(function(item) {
                message += "Full name: " + item.get("Name") +
                ". Number of activities: " + item.get("ActivitiesCount") + "\n";
            });
            // Displaying message in console.
            window.console.log(message);
        }, this);
    }
}
```

> NOTE
>
> Unlike the previous example, authentication is not needed in this case, because the program code is executed by bpm'online directly.

In the client of the application core, there is not a class like the server core *SelectQuery* class. To select data from a section, use the Terrasoft.EntitySchemaQuery class. For more information on this class methods and properties, please see the "**The use of EntitySchemaQuery implementation on client**" article .

DataService. Data filtering

Difficulty level

General information

During the execution of DataService operations, it is often necessary to filter data. For example, when reading section records, you need to fetch only those records that meet certain criteria. Bpm'online provides the *Filters* class to form these criteria .

The Filters class

The *Filters* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace of the *Terrasoft.Nui.ServiceModel.dll* class library. For simplicity, the hierarchical structure of the *Filters* data filter is conveniently presented as a JSON format object:

```
"Filters":{
    "RootSchemaName":["Root schema name"],
    "FilterType":[Filter type],
    "ComparisonType":[Comparison type],
    "LogicalOperation":[Logical operation],
    "IsNull":[Completeness checkbox],
    "IsEnabled":[Activation checkbox],
    "IsNot":[Negation operator checkbox],
    "SubFilters":[Subquery filters],
    "Items":[Filter group collection],
    "LeftExpression":[Expression to be checked],
    "RightExpression":[Filtration expression],
    "RightExpressions":[Filtration expressions array],
    "RightLessExpression":[Initial filtration range expression],
    "RightGreaterExpression":[Final filtration range expression],
    "TrimDateTimeParameterToDate":[Cutting time for date/time parameters checkbox],
    "Key":["Filter key in the filter collection"],
    "IsAggregative":[Aggregating filter checkbox],
    "LeftExpressionCaption":["Expression title to be checked"],
    "ReferenceSchemaName":["Reference schema name"]
}
```

The basic properties of the *Filters* class and their possible values are presented in table 1.

Table 1. *Filters class properties.*

Property	Type	Description
RootSchemaName	*string*	A string containing the name of the root object schema of the added record.
FilterType	*FilterType*	Filter type. Set by the *FilterType* enumeration value of the *Terrasoft.Nui.ServiceModel.DataContract* namespace. *FilterType* enumeration values: *None* 0 Filter type not defined. CompareFilter 1 Comparison filter. Used to compare expression results. IsNullFilter 2 The filter that defines whether an expression is empty. Between 3 The filter that defines whether an expression is one of the expressions.

InFilter	4	The filter that defines whether an expression equals one of the expressions.
Exists	5	Existence filter.
FilterGroup	6	Filter group. Filter groups can be nested in one another, i.e., the collection itself can be an element of another collection.

ComparisonType	*FilterComparisonType*	Comparison operation type. Set by the *FilterComparisonType* enumeration value of the *Terrasoft.Core.Entities* namespace.
LogicalOperation	*LogicalOperationStrict*	Logical operation. This type does not allow the None value specified in the *LogicalOperationStrict* enumeration of the *Terrasoft.Common* namespace.
IsNull	*bool*	Expression completion checkbox.
IsEnabled	*bool*	Checkbox that defines whether the filter is active and will be taken into account when building a request.
IsNot	*bool*	Specifies whether to use the negation logical operator.
SubFilters	*Filters*	Subrequest filters. Cannot contain filters with other subrequests.
Items	*Dictionary<string, Filter>*	Collection containing a filter group.
LeftExpression	*BaseExpression*	The expression in the left part of the comparison, i.e. the expression to be tested. The *BaseExpression* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace.
RightExpression	*BaseExpression*	The filter expression that will be compared to the expression contained in the *LeftExpression* property.
RightExpressions	*BaseExpression[ ]*	The expression array that will be compared to the expression contained in the *LeftExpression* property.
RightLessExpression	*BaseExpression*	Initial filtration range expression.
RightGreaterExpression	*BaseExpression*	Final filtration range expression.
TrimDateTime ParameterToDate	*bool*	Checkbox indicating whether to cut time from the date-time parameters.
Key	*string*	Filter key in the collection of Items filters.
IsAggregative	*bool*	Aggregating filter checkbox.
LeftExpressionCaption	*string*	Left comparison part title.
ReferenceSchemaName	*string*	The object schema name referenced by the left part of the filter if the column type is lookup.

The *BaseExpression* class is the base expression class. It is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace of the *Terrasoft.Nui.ServiceModel* library. The properties of this class instance are populated depending on the *ExpressionType* property that specifies the expression type. A complete list of the BaseExpression class properties is given in table. 2.

Table 2. BaseExpression class main properties

Property	Type	Description
ExpressionType	*EntitySchemaQuery ExpressionType*	The expression type that defines the value that will be contained in the added column. Set by the

		EntitySchemaQueryExpressionType enumeration of the *Terrasoft.Core.Entities* namespace defined in the *Terrasoft.Core* class library. For the *InsertQuery* the *EntitySchemaQueryExpressionType.Parameter* value is set. The *EntitySchemaQueryExpressionType* enumeration values: *SchemaColumn* 0 Schema column. *Function* 1 Function *Parameter* 2 Parameter *SubQuery* 3 Subquery *ArithmeticOperation* 4 Arithmetic operation
ColumnPath	*string*	The path to a column relative to the root schema. The rules for building the paths can be found in the "**The use of EntitySchemaQuery for creation of queries in database**" article.
Parameter	*Parameter*	Defines the value that will be contained in the added column. Its *Parameter* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace.
FunctionType	*FunctionType*	Function type. Set by the value from the *FuctionType* enumeration defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace. *FunctionType* enumeration values: *None* 0 Not defined *Macros* 1 Macro *Aggregation* 2 Aggregating function *DatePart* 3 Date part *Length* 4 Length
MacrosType	*EntitySchemaQuery MacrosType*	Macro type. Set by the value from the *EntitySchemaQueryMacrosType* enumeration defined in the *Terrasoft.Core.Entities* namespace.
FunctionArgument	*BaseExpression*	Function argument. Takes the value if the function is defined with a parameter. The *BaseExpression* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace and is the ancestor of the *ColumnExpresion* class and has the same set of properties.
DatePartType	*DatePart*	Date part. Set by the value from the *DatePart* enumeration defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace. *DatePart* enumeration values: *None* 0 Not defined *Day* 1 Day *Week* 2 Week *Month* 3 Month *Year* 4 Year *Weekday* 5 Day of the week

		Hour	6	Hour
		HourMinute	7	Minute
AggregationType	*AggregationType*	Aggregating function type. Sets the value of AggregationType enumeration defined in the namespace Terrasoft.Common defined in the class library Terrasoft.Common		
AggregationEvalType	*AggregationEvalType*	Aggregating function Set by the value from the *AggregationEvalType* enumeration defined in the *Terrasoft.Core.DB* namespace defined in the *Terrasoft.Core* class library.		
SubFilters	*Filters*	Subquery filter collection. Its *Filter* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace.		

Learn more about filters in the "**EntitySchemaQuery filters handling**" article. Next, there is an example of using filters in requests to the DataService service from a third-party application.

Using filters in a third-party application example

Case description

You need to create a console application that will read the following data from the [Contact] section using the DataService service:

- Id
- Full name
- Number of activities is an aggregating column that shows the number of activities of this contact.

It is necessary to filter the data so that only those contacts whose number of activities is in the range of 1 to 3, and the [Full name] column value starting with "H" are read.

Case implementation algorithm

1. Create and configure a C# console application project that reads records

To perform this step, you must perform the example of reading records in a third-party application, described in the "**DataService. Reading records**" article.

The result of the query class implementation instance to read the records with the columns in an abbreviated form:

```
// Query class instance.
var selectQuery = new SelectQuery ()
{
    // Root schema name.
    RootSchemaName = "Contact",
    // Adding columns to query.
    Columns = new SelectQueryColumns ()
    {
        // Column collection.
        Items = new Dictionary <string, SelectQueryColumn> ()
        {
            // Column [Full name].
            {
                // Key.
                "Name",
                // Value.
                new SelectQueryColumn ()
                {
                    // An expression that specifies the column type.
                    Expression = new ColumnExpression ()
                    {
                        // Expression type - schema column.
```

```
                        ExpressionType =
EntitySchemaQueryExpressionType.SchemaColumn,
                        // Path to the column.
                        ColumnPath = "Name"
                    }
                }
            }
            // Column [Number of activities].
            {
                "ActivitiesCount",
                new SelectQueryColumn ()
                {
                    Expression = new ColumnExpression ()
                    {
                        // Expression - subquery.
                        ExpressionType = EntitySchemaQueryExpressionType.SubQuery,
                        // Path to the column relative to the root schema.
                        ColumnPath = "[Activity: Contact] .Id",
                        // Function type - aggregating.
                        FunctionType = FunctionType.Aggregation,
                        // Aggregation type - number.
                        AggregationType = AggregationType.Count
                    }
                }
            }
        }
    }
};
```

2. Add filter implementation

In order to filter data, you must create an instance of the *Filters* collection class instance, fill in the necessary properties, and then pass the link to this instance to the *Filters* property of the query class instance that you created in the previous step.

Filter collection class implementation example:

```
// Query filters.
var selectFilters = new Filters ()
{
    // Filter Type - group.
    FilterType = Terrasoft.Nui.ServiceModel.DataContract.FilterType.FilterGroup,
    // Filter collection.
    Items = new Dictionary <string, Filter>
    {
// Filter Implementation.
    }
};
// Adding filter to query.
selectQuery.Filters = selectFilters;
// Query class instance serialization to read data from the JSON string.
var json = new JavaScriptSerializer () Serialize (selectQuery).;
```

The Items property must contain the key-value type collection. The key is a string containing the filter name, and the value is an instance of the Filter class that contains a direct implementation of the filter.

To implement a filter that selects only those contacts that have a number of activities within a range of 1 to 3, you must add the following instance to the collection of filters:

```
// Filtration by activity.
{
    // Key.
    "FilterActivities",
```

```
    // Value.
    new Filter
    {
        // Filter type - range filter.
        FilterType = Terrasoft.Nui.ServiceModel.DataContract.FilterType.Between,
        // Comparison type - range.
        ComparisonType = FilterComparisonType.Between,
        // An expression to be tested.
        LeftExpression = new BaseExpression ()
        {
            // Expression type - subquery.
            ExpressionType = EntitySchemaQueryExpressionType.SubQuery,
            // Path to the column relative to the root schema.
            ColumnPath = "[Activity: Contact] .Id",
            // Function type - aggregating.
            FunctionType = FunctionType.Aggregation,
            // Aggregation type - number.
            AggregationType = AggregationType.Count
        }
        // Filter range final expression.
        RightGreaterExpression = new BaseExpression ()
        {
            // Expression type - parameter.
            ExpressionType = EntitySchemaQueryExpressionType.Parameter,
            // Expression parameter.
            Parameter = new Parameter ()
            {
                // Parameter data type - integer.
                DataValueType = DataValueType.Integer,
                // Parameter value.
                Value = 3
            }
        }
        // Filter range initial expression.
        RightLessExpression = new BaseExpression ()
        {
            ExpressionType = EntitySchemaQueryExpressionType.Parameter,
            Parameter = new Parameter ()
            {
                DataValueType = DataValueType.Integer,
                Value = 1
            }
        }
    }
}
```

Add the following instance to the filter collection to filter contact records where the [Full name] column value begins with "H":

```
// Filtering by name.
{
    // Key.
    "FilterName",
    // Value.
    new Filter
    {
        // Filter type - comparison filter.
        FilterType =
Terrasoft.Nui.ServiceModel.DataContract.FilterType.CompareFilter,
        // Comparison type - starts with an expression.
        ComparisonType = FilterComparisonType.StartWith,
        // Expression to be tested.
```

```
        LeftExpression = new BaseExpression ()
        {
            // Expression type - schema column.
            ExpressionType = EntitySchemaQueryExpressionType.SchemaColumn,
            // Path to the column.
            ColumnPath = "Name"
        }
        // Filtration expression.
        RightExpression = new BaseExpression ()
        {
            // Expression type - parameter.
            ExpressionType = EntitySchemaQueryExpressionType.Parameter,
             // Expression parameter.
            Parameter = new Parameter ()
            {
                // Parameter data type - text.
                DataValueType = DataValueType.Text,
                // Parameter value.
                Value = "CH"
            }
        }
    }
}
```

DataService. Using macros

Difficulty level

General provisions

During execution of DataService operations data often needs to be filtered for a certain period of time. Macros simplify such tasks and help to avoid creating unnecessary custom methods. The macros are implemented in a form of special classes that are designed for calculating typical values in query expressions, such as calculating the start and end date of the current quarter. Macros can be used only if the query expression type is a function. For more information about macro expression types, please see the **DataService. Data filtering** article.

Types of macros

When creating queries to DataService, both parameterized (ie requiring an argument) and non-parameterized macros can be used. Macro types that must be used in the macro expressions are defined in the *EntitySchemaQueryMacrosType* enumeration in the *Terrasoft.Core.Entities* name space. Enumeration values of macro types and their descriptions are available in table 1.

Table 1. Values of the *EntitySchemaQueryMacrosType* enumeration and their descriptions

Macro	Value	Description
CurrentHalfYear	16	Current half-year (January-June or July-December).
CurrentHour	21	Current hour.
CurrentMonth	10	Current month.
CurrentQuarter	13	Current quarter.
CurrentUser	1	Current user.
CurrentUserContact	2	Contact record of the current user.
CurrentWeek	7	Current week.

CurrentYear	19	Current year.
DayOfMonth	28	Day of month. Requires parameterization.
DayOfWeek	29	Week day. Requires parameterization.
Hour	30	Hour. Requires parameterization.
HourMinute	31	Time. Requires parameterization.
Month	32	Month. Requires parameterization.
NextHalfYear	17	Next half-year (January-June or July-December).
NextHour	22	Next hour.
NextMonth	11	Next month.
NextNDays	24	Next N days. Requires parameterization.
NextNHours	26	Next N hours. Requires parameterization.
NextQuarter	14	Next quarter.
NextWeek	8	Next week.
NextYear	23	Next year.
None	0	Type of macro not defined.
PreviousHalfYear	15	Previous half-year (January-June or July-December).
PreviousHour	20	Previous hour.
PreviousMonth	9	Previous month.
PreviousNDays	25	Previous N days. Requires parameterization.
PreviousNHours	27	Previous N hours. Requires parameterization.
PreviousQuarter	12	Previous quarter.
PreviousWeek	6	Previous week.
PreviousYear	18	Previous year.
Today	4	Today.
Tomorrow	5	Tomorrow.
Year	33	Year. Requires parameterization.
Yesterday	3	Yesterday.

Example of using macros

Case description

Create a console application that uses DataService to read records from the [Contacts] section with the following columns:

- Id
- Full name
- Birth date

The data must be filtered, so that only contacts who were born in 1992 are shown.

Case implementation algorithm

1. Create and set up a C# application project that reads records

To execute this step, execute the record reading case covered in the "**DataService. Reading records**" article.

The result of implementing an instance of query class for reading records:

```
// Instance of query class.
var selectQuery = new SelectQuery()
{
    // Root schema name.
    RootSchemaName = "Contact",
    // Adding columns to query.
    Columns = new SelectQueryColumns()
    {
        // Collection of columns.
        Items = new Dictionary<string, SelectQueryColumn>()
        {
            //Column [Full name].
            {
                // Key.
                "Name",
                // Value.
                new SelectQueryColumn()
                {
                    // Expression that specifies the column type.
                    Expression = new ColumnExpression()
                    {
                        // Type of expression - schema column.
                        ExpressionType =
EntitySchemaQueryExpressionType.SchemaColumn,
                        // Pat to column.
                        ColumnPath = "Name"
                    }
                }
            },
            // Column [Number of activities].
            {
                "ActivitiesCount",
                new SelectQueryColumn()
                {
                    Expression = new ColumnExpression()
                    {
                        // Expression type – subquery.
                        ExpressionType = EntitySchemaQueryExpressionType.SubQuery,
                        // Path to column relative to the oot schema.
                        ColumnPath = "[Activity:Contact].Id",
                        // Function tyoe – aggregation.
                        FunctionType = FunctionType.Aggregation,
                        // Aggregation type – quantity.
                        AggregationType = AggregationType.Count
                    }
                }
            }
        }
    }
};
```

2. Add a filter implementation with macros

To filter the data, create an instance of the *Filters* collection class, fill out the properties with corresponding values, and then pass the instance link to the Filters property of the query class created on the previous step.

An example of filter collection class implementation:

```
// Query filters.
var selectFilters = new Filters()
{
    // Filter type – group.
```

```
    FilterType = Terrasoft.Nui.ServiceModel.DataContract.FilterType.FilterGroup,
    // Filter collection.
    Items = new Dictionary<string, Filter>
    {

        // Filter by year of birth.
        {
            // Key.
            "FilterYear",
            // Value.
            new Filter
            {
                // Filter type — comparison.
                FilterType =
Terrasoft.Nui.ServiceModel.DataContract.FilterType.CompareFilter,
                // Comparison type — equal.
                ComparisonType = FilterComparisonType.Equal,
                // Expression to check.
                LeftExpression = new BaseExpression()
                {
                    // Expression type — schema column.
                    ExpressionType = EntitySchemaQueryExpressionType.SchemaColumn,
                    // Path to schema.
                    ColumnPath = "BirthDate"
                },
                // Expression with which the checked value is compared.
                RightExpression = new BaseExpression
                {
                    // Expression type — function.
                    ExpressionType = EntitySchemaQueryExpressionType.Function,
                    // Function type — macro.
                    FunctionType = FunctionType.Macros,
                    // Macro type — year.
                    MacrosType = EntitySchemaQueryMacrosType.Year,
                    // Function argument.
                    FunctionArgument = new BaseExpression
                    {
                        // Type of expression that determines the argument —
parameter.
                        ExpressionType = EntitySchemaQueryExpressionType.Parameter,
                        // Parameter initialization.
                        Parameter = new Parameter
                        {
                            // Parameter type — integer.
                            DataValueType = DataValueType.Integer,
                            // Parameter value.
                            Value = "1992"
                        }
                    }
                }
            }
        }
    }
};
// Adding filters to query.
selectQuery.Filters = selectFilters;
// Serialization of select query class instance in a JSON string.
var json = new JavaScriptSerializer().Serialize(selectQuery);
```

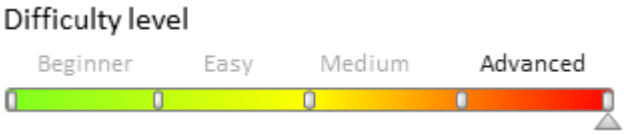
The collection contains a single filter with the "FilterYear" key. Because only those records that have their year of birth equal to 1992 must be selected from the collection, the type of filter is set as a comparison filter. The type of comparison is set as an equality of values. As a verified expression, set the [Date of birth] column. Specify the macro

function as the expression to compare with.

In this case using a macro is optimal because the birth date is stored in the database in "YYYY-MM-DD" format. The macro automatically determines the year value, so the developer does not need to write additional program code.

Because the *EntitySchemaQueryMacrosType.Year* macro is parametric, the *FunctionArgument* property must be initialized and assigned a link to an instance of the *BaseExpression* class. In it, the integer parameter with value "1992" is defined.

DataService. Updating records



General provisions

The DataService web service of bpm'online is a RESTful (Representational State Transfer, REST) service. The RESTful data management interface does not require converting data to an external format, such as XML. In a simple RESTful service, each information unit is determined by a global Identifier such as URL. Each URL, in its turn, has a strictly specified format. This is not an optimal way to transfer large arrays of data.

With the use of the DataService, the data can be automatically configured in various data formats such as XML, JSON, HTML, CSV, and JSV. The data structure is determined by data contracts. A complete list of data contracts used by the DataService, can be found in the "**DataService web service**" article.

UpdateQuery data contract

The UpdateQuery data contract is used for updating section records. The query data is transferred to DataService via HTTP, with the help of POST by the following URL:

```
// URL format of the POST query to DataService to update data.
http(s)://[Bpm'online application address]/[Configuration number]/dataservice/[Data
fromat]/reply/UpdateQuery
// URL example of the POST query to DataService to update data.
http(s)://example.bpmonline.com/0/dataservice/json/reply/UpdateQuery
```

The UpdateQuery data contract has a hierarchical structure with a number of nesting levels. In the bpm'online server core, it is represented by a UpdateQuery class of theTerrasoft.Nui.ServiceModel.DataContract namespace of the Terrasoft.Nui.ServiceModel.dll library of classes. For the hierarchical data structure of the UpdateQuery data contract can be conveniently viewed in JSON format:

```
{
    "RootSchemaName":"[Root schema]",
    "OperationType":[Type of operation with record],
    "IsForceUpdate":[Force update],
   "ColumnValues":{
        "Items":{
            "Name of the added column":{
                "ExpressionType":[Expression type],
                "Parameter":{
                    "DataValueType":[Data type],
                    "Value":"[Column value]"
                }
            }...
        }
    },
    "Filters":[Request filters]
}
```

Primary properties of the UpdateQuery class and their possible values are available in table 1.

Table 1. UpdateQuery class properties

Property	Type	Notes
RootSchemaName	*string*	String that contains root schema name of added record object.
OperationType	*QueryOperationType*	Type of write operation. Specified as a QueryOperationType enumeration value of the Terrasoft.Nui.ServiceModel.DataContract name space. The *QueryOperationType.Insert* value is set for *InsertQuery*. Values of the QueryOperationType enumeration: *Select* 0 *Insert* 1 *Update* 2 *Delete* 3 *Batch* 4
IsForceUpdate	*bool*	Indicates force update. If the value is *true*, the entity will be saved on the server even if column values have been modified. Default value: false.
ColumnValues	*ColumnValues*	Contains collection of column values for the added record. The *ColumnValues* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space.
Filters	*Filters*	Collection of query filters. The *Filters* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space.

The *ColumnValues* class has a single Items property, defined as a collection of keys and values *Dictionary<string, ColumnExpression>*. The key is the string with the name of the added column. The value is an object of the *ColumnExpression* type, defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. General properties of the *ColumnExpression* class used when adding records are available in table 2.

Table 2. Primary properties of the ColumnExpression class

Property	Description
ExpressionType	Type of expression that determines the value that the added column will contain. Specified with a value from the *EntitySchemaQueryExpressionType* enumeration of the *Terrasoft.Core.Entities* name space defined in the *Terrasoft.Core* class library. The *EntitySchemaQueryExpressionType.Parameter* value is set for *InsertQuery*. Values of the *EntitySchemaQueryExpressionType* enumeration: *SchemaColumn* 0 *Function* 1 *Parameter* 2 *SubQuery* 3 *ArithmeticOperation* 4
Parameter	Determines the value that the added column will contain. The *Parameter* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space.

The Parameter class has a number of properties, only two of which are used for adding records (table 3).

Table 3. Primary properties of the Parameter class

Property	Description
DataValueType	Type of data for the value that the added column will contain. Specified as a *DataValueType* enumeration value of the *Terrasoft.Nui.ServiceModel.DataContract* name space. Values of the *DataValueType* enumeration: *Guid* 0 *Text* 1 *Integer* 4 *Float* 5 *Money* 6 *DateTime* 7 *Date* 8 *Time* 9 *Lookup* 10 *Enum* 11 *Boolean* 12 *Blob* 13 *Image* 14 *ImageLookup* 16 *Color* 18 *Mapping* 26
Value	The object that contains the value of the added column. Has the *Object* type.

The *Filters* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. For details on the properties of this class and its use, please see the "**DataService. Data filtering**" article.

> NOTE
>
> An instance of the *UpdateQuery* class must contain a link to a correctly initialized instance of the *Filters* class in the *Filters* property. Otherwise, new column values from the *ColumnValues* property will be set for ALL section records.

Example of updating records in a third-party application

Case description

Create a console application that used DataService to update the "John Smith" record added in the example of the "**DataService. Adding records**" article. Add "j.smith@bpmonline.com" as the value in the [Email] column of this record.

Case implementation algorithm

1. Create and set up a C# application project

Using the Microsoft Visual Studio development environment (version 2017 and up), create a Visual C# console application project and specify project name, for example, DataServiceUpdateExample. Set ".NET Framework 4.7" for the project property [Target framework].

In the References section of the project, add dependencies from the following libraries:

- *System.Web.Extensions.dll* – class library included in .NET Farmework;
- *Terrasoft.Core.dll* – library of base bpm'online server core classes. It can be found using the following path: [Bpm'online setup catalog]\Terrasoft.WebApp\bin\Terrasoft.Core.dll;
- *Terrasoft.Nui.ServiceModel.dll* — application service class library. It can be found using the following path: [Bpm'online setup catalog]\Terrasoft.WebApp\bin\Terrasoft.Nui.ServiceModel.dll;
- *Terrasoft.Common.dll* – library of base bpm'online server core classes. It can be found using the following path: [Bpm'online setup catalog]\Terrasoft.WebApp\bin\Terrasoft.Common.dll.

Add the "using" directives to the application source code file:

```
using System;
using System.Text;
using System.IO;
using System.Net;
using System.Collections.Generic;
using Terrasoft.Nui.ServiceModel.DataContract;
using Terrasoft.Core.Entities;
using System.Web.Script.Serialization;
using Terrasoft.Common;
```

2. Add fields and constants and field declarations to the source code

To access DataService features, add the following fields and constants to the application source code:

```
// Primary URL of bpm'online application. Must be repoaced with a custom one.
private const string baseUri = @"http://example.bpmonline.com";
// Request string to the Login methid of the AuthService.svc service.
private const string authServiceUri = baseUri +
@"/ServiceModel/AuthService.svc/Login";
// Path string for the UpdateQuery.
private const string updateQueryUri = baseUri +
@"/0/DataService/json/reply/UpdateQuery";
// Bpm'online authentication cookie.
private static CookieContainer AuthCookie = new CookieContainer();
```

Here, three string fields are declared. These fields will be used to form authentication query and read data queries execution paths. Authentication data will be saved in the AuthCookie field.

3. Add method that performs bpm'online application authentication

Authentication is required to enable access to the DataService for the created application.

Both the algorithm and example of implementation method, which contains query to AuthService.svc for user authentication, are available in the "**Authenticating external requests to bpm'online services**" article.

4. Add implementation of the record add query

Because the *updateQueryUri* constant declared earlier contains a path for sending data in the JSON format, sent data must be configured beforehand as a string that contains a JSON object that corresponds to the *UpdateQuery* data contract. This can be done directly in a string variable, although a much more secure and convenient way of doing this would be to create an instance of the *UpdateQuery* class, fill out its properties and then serialize it to a string. This can be done with the help of the following source code:

```
// Instance of the request class.
 var updateQuery = new UpdateQuery()
 {
     // Root schema name.
     RootSchemaName = "Contact",
     // New column values.
     ColumnValues = new ColumnValues()
     {
         // Key-value collection.
         Items = new Dictionary<string, ColumnExpression>()
```

```
        {
            // [Email] column.
            {
                // key.
                "Email",
                // Value – instance of object schema request class.
                // Configuration of [Email] column.
                new ColumnExpression()
                {
                    // Type of expression of obkect schema query – parameter.
                    ExpressionType = EntitySchemaQueryExpressionType.Parameter,
                    // Query expression parameter.
                    Parameter = new Parameter()
                    {
                        // Parameter value.
                        Value = "j.smith@bpmonline.com",
                        // Parameter data type – string.
                        DataValueType = DataValueType.Text
                    }
                }
            }
        }
    },
    // Query filters.
    Filters = new Filters()
    {
        // Filter type – group.
        FilterType = Terrasoft.Nui.ServiceModel.DataContract.FilterType.FilterGroup,
        // Filter collection.
        Items = new Dictionary<string, Filter>()
        {
             // Filter by name.
            {
                // Key.
                "FilterByName",
                // Value.
                new Filter
                {
                    // Filter type – comparison filter.
                    FilterType =
Terrasoft.Nui.ServiceModel.DataContract.FilterType.CompareFilter,
                    // Comparison type – starts with expression.
                    ComparisonType = FilterComparisonType.Equal,
                    // Expression to check.
                    LeftExpression = new BaseExpression()
                    {
                        // Expression type - schema column.
                        ExpressionType =
EntitySchemaQueryExpressionType.SchemaColumn,
                        // Path to column.
                        ColumnPath = "Name"
                    },
                    // Filtering expression.
                    RightExpression = new BaseExpression()
                    {
                        // Expression type - parameter.
                        ExpressionType = EntitySchemaQueryExpressionType.Parameter,
                         // Expression parameter.
                        Parameter = new Parameter()
                        {
                            // Parameter data type - text.
                            DataValueType = DataValueType.Text,
```

```
                            // Parameter value.
                            Value = "John Smith"
                        }
                    }
                }
            }
        }
    }
};
// Serialization of update query class instance in a JSON string.
var json = new JavaScriptSerializer().Serialize(updateQuery);
```

Here, an instance of the *UpdateQuery* class is created. In the *ColumnValues* property, the "j.smith@bpmonline.com" value is set for the [Email] column. To apply this value to a specific record or group of records, specify a link to a correctly initialized Filters class in the Filters property. In this case, a single filter is added to the filters collection to select only records that have the "John Smith" value in the [Full name] column.

The next step is to execute DataService POST-query. To do this, create an instance of the HttpWebRequest class, fill its properties and connect the string with the JSON object created earlier then execute the DataService query and process its result. To do this, add the following source code:

```
// Converting a JSON object string to a byte array.
byte[] jsonArray = Encoding.UTF8.GetBytes(json);
// Creating an insrance of HTTP request.
var updateRequest = HttpWebRequest.Create(updateQueryUri) as HttpWebRequest;
// Defining a request method.
updateRequest.Method = "POST";
// Determining type of request content.
updateRequest.ContentType = "application/json";
// Adding authentication cookie received earlier to a request.
updateRequest.CookieContainer = AuthCookie;
// Set length for request content.
updateRequest.ContentLength = jsonArray.Length;
// Plase a JSON-object to request content.
using (var requestStream = updateRequest.GetRequestStream())
{
    requestStream.Write(jsonArray, 0, jsonArray.Length);
}
// Executing HTTP request and getting a response from server.
using (var response = (HttpWebResponse)updateRequest.GetResponse())
{
    // Displaying response in console.
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        Console.WriteLine(reader.ReadToEnd());
    }
}
```

DataService. Deleting records

Difficulty level

General information

The bpm'online DataService web service is a RESTfull service. RESTful is a quite simple information management interface that doesn't use any additional internal layers, i.e., the data doesn't need to be converted to any third-party format, such as XML. In a simple RESTful service, each record is uniquely identified by a global identifier such as a URL. Each URL has a strictly specified format. However, this service is not always convenient for transferring large

amounts of data.

With the use of the DataService, the data can be automatically configured in various data formats such as XML, JSON, HTML, CSV, and JSV. The data structure is determined by data contracts. A complete list of data contracts used by the DataService, can be found in the "**DataService web service**" article.

DeleteQuery data contract

The *DeleteQuery* contract is used to delete sections. The data is transferred to the DataService via HTTP by using the POST request with the following URL:

```
// URL format of the POST query to DataService to delete data.
http(s)://[Bpm'online application address]/[Configuration number]/dataservice/[Data
fromat]/reply/DeleteQuery
// URL example of the POST query to DataService to delete data.
http(s)://example.bpmonline.com/0/dataservice/json/reply/DeleteQuery
```

The *DeleteQuery* data contract has a hierarchical structure with multiple nesting levels. In the bpm'online application server part, the *DeleteQuery* data contract is represented by the *InsertQuery* class of the *Terrasoft.Nui.ServiceModel.DataContract* namespace of the *Terrasoft.Nui.ServiceModel.dll* class library. However, for simplicity, the hierarchical structure of the *DeleteQuery* data contract is conveniently presented as a JSON format object:

```
{
    "RootSchemaName":"[Root schema]",
    "OperationType":[Record operation type],
    "ColumnValues":[Column values. Not used.],
    "Filters":[Query filters]
}
```

The basic properties of the *DeleteQuery* class and their possible values are presented in table 1.

Table 1. *DeleteQuery class properties.*

Property	Type	Description
RootSchemaName	*string*	A string containing the name of the root object schema of the added record.
OperationType	*QueryOperationType*	Operation type is set by the *QueryOperationType namespace Terrasoft.Nui.ServiceModel.DataContract* namespace enumeration value. For the *InsertQuery* the*QueryOperationType.Insert* value is set. *QueryOperationType* enumeration values: *Select* 0 *Insert* 1 *Update* 2 *Delete* 3 *Batch* 4
ColumnValues	*ColumnValues*	Contains a collection of column values of the added record. Inherited from the *BaseQuery* parent class. Not used in this type of queries.
Filters	*Filters*	Query filter collection. Its *Filter* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace.

The *Filters* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace. The properties of this class are described in the "**DataService. Data filtering**" article.

IMPORTANT

The *DeleteQuery* query class instance must contain a link to the correctly initialized *Filters* class instance in the *Filters* property. Otherwise ALL section records will be deleted.

Deleting records using a third-party application example

Case description

You need to create a console application that, using the DataService service, will delete the "John Best" contact record added in the example of the "**DataService. Adding records**" article.

Case implementation algorithm

1. Create and configure a C# console application project

Using the Microsoft Visual Studio (version 2017 and up) development environment, create a Visual C# console application project and name it *DataServiceDeleteExample*. The [Target framework] project property must be set to .NET Framework 4.7.

In the References section of the project you need to add dependencies of the following libraries:

- *System.Web.Extensions.dll* is a class library included in the .NET Farmework
- *Terrasoft.Core.dll* is a main class library of the application server kernel. Can be found by the following path: [Directory with the installed application]\Terrasoft.WebApp\bin\Terrasoft.Core.dll
- Terrasoft.Nui.ServiceModel.dll class library the application services. Can be found by the following path: [Directory with the application installed]\Terrasoft.WebApp\bin\Terrasoft.Nui.ServiceModel.dll.
- *Terrasoft.Common.dll* is a main class library of the application server kernel. Can be found by the following path: [Directory with the installed application]\Terrasoft.WebApp\bin\Terrasoft.Common.dll

Add *using* directives to the application source code file:

```
using System;
using System.Text;
using System.IO;
using System.Net;
using System.Collections.Generic;
using Terrasoft.Nui.ServiceModel.DataContract;
using Terrasoft.Core.Entities;
using System.Web.Script.Serialization;
using Terrasoft.Common;
```

2. Add field declarations and constants to the application source code

To access the DataService features, you must add the following fields and constants to the application source code:

```
 // Main bpm'online URL. Has to be changed to a custom one.
 private const string baseUri = @"http://example.bpmonline.com";
// Query string to the Login method of the AuthService.svc service.
 private const string authServiceUri = baseUri +
@"/ServiceModel/AuthService.svc/Login";
 // DeleteQuery query path string.
 private const string deleteQueryUri = baseUri +
@"/0/DataService/json/reply/DeleteQuery";
 // Bpm'online cookie authentication.
 private static CookieContainer AuthCookie = new CookieContainer();
```

Three string constant fields that are used to carry out the authentication requests and requests to read data are declared here. The authentication data will be stored in the *AuthCookie* field.

3. Add a method that performs authentication in the bpm'online application

You need to authenticate the newly created application to access the DataService web service.

The algorithm and an implementation example of a method that performs a request to the *AuthService.svc* service for user authentication can be found in the **Authenticating external requests to bpm'online services** article.

4. Implement a query to add a record

As the previously declared *updateQueryUri* constant contains the path for sending data in JSON format, the data sent must be pre-configured in the form of a string containing a description of the JSON object corresponding to the *UpdateQuery* data contract. This can be done directly in a lowercase variable but it is much easier and safer to create an instance of the *UpdateQuery* class, fill its properties, and then serialize it to a string. This can be done by adding the following source code:

```
// Query class instance.
var deleteQuery = new DeleteQuery()
{
    // Root schema name.
    RootSchemaName = "Contact",
    // Query filters.
    Filters = new Filters()
    {
        // Filter type — group.
        FilterType = Terrasoft.Nui.ServiceModel.DataContract.FilterType.FilterGroup,
        // Filter collection.
        Items = new Dictionary<string, Filter>()
        {
             // Filtration by name.
            {
                // Key.
                "FilterByName",
                // Value.
                new Filter
                {
                    // Filter type — comparison filter.
                    FilterType =
Terrasoft.Nui.ServiceModel.DataContract.FilterType.CompareFilter,
                    // Comparison type — starts with an expression.
                    ComparisonType = FilterComparisonType.Equal,
                    // Expression to be checked.
                    LeftExpression = new BaseExpression()
                    {
                        // Expression type — schema column.
                        ExpressionType =
EntitySchemaQueryExpressionType.SchemaColumn,
                        // Column path.
                        ColumnPath = "Name"
                    },
                    // Filtration expression.
                    RightExpression = new BaseExpression()
                    {
                        // Expression type — parameter.
                        ExpressionType = EntitySchemaQueryExpressionType.Parameter,
                         // Expression parameter.
                        Parameter = new Parameter()
                        {
                            // Parameter data type — text.
                            DataValueType = DataValueType.Text,
                            // Parameter value.
                            Value = "John Best"
                        }
                    }
                }
            }
        }
```

```
    }
};
// Class instance serialization of the JSON string adding query.
var json = new JavaScriptSerializer().Serialize(updateQuery);
```

This creates an instance of the *DeleteQuery* class. The "Contact" value is set in the *RootSchemaName* property. To delete a particular record or group of records, you need to set a link to the correctly initialized *Filters* class instance to the Filters property. In this case, a single filter that selects only records with the "John Best" value in the [Full name] column is added to the filter collection.

In the final step you must perform POST query to the DataService service. To do this, create an instance of the *HttpWebRequest* class, fill in its properties, attach a previously created string with the JSON object to a request, and then execute and process the result of the query to the DataService service. To do this, add the following source code:

```
// Converting a JSON object string to a byte array.
byte[] jsonArray = Encoding.UTF8.GetBytes(json);
// Creating an insrance of HTTP request.
var updateRequest = HttpWebRequest.Create(updateQueryUri) as HttpWebRequest;
// Defining a request method.
updateRequest.Method = "POST";
// Determining type of request content.
updateRequest.ContentType = "application/json";
// Adding authentication cookie received earlier to a request.
updateRequest.CookieContainer = AuthCookie;
// Set length for request content.
updateRequest.ContentLength = jsonArray.Length;
// Plase a JSON-object to request content.
using (var requestStream = updateRequest.GetRequestStream())
{
    requestStream.Write(jsonArray, 0, jsonArray.Length);
}
// Executing HTTP request and getting a response from server.
using (var response = (HttpWebResponse)updateRequest.GetResponse())
{
    // Displaying response in console.
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        Console.WriteLine(reader.ReadToEnd());
    }
}
```

DataService. Batch queries

Difficulty level

General provisions

The DataService web service of bpm'online is a RESTful (Representational State Transfer, REST) service. The RESTful data management interface does not require converting data to an external format, such as XML. In a simple RESTful service, each information unit is determined by a global Identifier such as URL. Each URL, in its turn, has a strictly specified format. This is not an optimal way to transfer large arrays of data.

With the use of the DataService, the data can be automatically configured in various data formats such as XML, JSON, HTML, CSV, and JSV. The data structure is determined by data contracts. A complete list of data contracts used by the DataService, can be found in the "**DataService web service**" article.

Bach queries

Batch queries are used to minimize requests to DataServise, which improves application performance. Packet query is a collection that contains a custom set of DataService requests. The query data is transferred to DataService via HTTP, with the help of POST by the following URL:

```
// URL format of the batch POST query to DataService.
http(s)://[Bpm'online application address]/[Configuration number]/dataservice/[Data
fromat]/reply/BatchQuery
// URL example of the batch POST query to DataService.
http(s)://example.bpmonline.com/0/dataservice/json/reply/BatchQuery
```

The data that comprises a batch query can be passed in different formats. One of the more convenient formats is JSON: The structure of a batch query in JSON format is as follows:

```
{
    "items": [
        {
            "__type": "[Full qualified name of the query type]",
            //One-time query contents.
            ...
        },
        // Other one-time queries.
        ...
    ]
}
```

To generate the contents of one-time queries that comprise a batch query, use the following data constants: *InsertQuery*, *SelectQuery*, *UpdateQuery* and *DeleteQuery*.

Example of using a batch query in a third-party application

Case description

Create a console application that will use DataService to:

- Add a contact record with the value "John Smith" in the [Full name] column;
- Change the value of the [Business phone] column to 012 345 67 89 for all contact records that have "John Smith" as the value in the [Full name] column.

Records must be added and modified via a batch query.

Case implementation algorithm

1. Create and set up a C# application project

Using the Microsoft Visual Studio development environment (version 2017 and up), create a Visual C# console application project and specify the project name, for example, *DataServiceBatchExample*. Set ".NET Framework 4.7" for the project property [Target framework].

In the References section of the project, add dependencies from the following libraries:

- *System.Web.Extensions.dll* – class library included in .NET Farmework;
- *Terrasoft.Core.dll* – library of base bpm'online server core classes. It can be found using the following path: [Bpm'online setup catalog]\Terrasoft.WebApp\bin\Terrasoft.Core.dll;
- *Terrasoft.Nui.ServiceModel.dll* — application service class library. It can be found using the following path: [Bpm'online setup catalog]\Terrasoft.WebApp\bin\Terrasoft.Nui.ServiceModel.dll;
- *Terrasoft.Common.dll* – library of base bpm'online server core classes. It can be found using the following path: [Bpm'online setup catalog]\Terrasoft.WebApp\bin\Terrasoft.Common.dll.

Add the "using" directives to the application source code file:

```
using System;
using System.Text;
```

```
using System.IO;
using System.Net;
using System.Collections.Generic;
using Terrasoft.Nui.ServiceModel.DataContract;
using Terrasoft.Core.Entities;
using System.Web.Script.Serialization;
using Terrasoft.Common;
```

2. Add fields and constants and field declarations to the source code

To access DataService features, add the following fields and constants to the application source code:

```
// Primary URL of bpm'online application. Must be repoaced with a custom one.
private const string baseUri = @"http://example.bpmonline.com";
// Request string to the Login methid of the AuthService.svc service.
private const string authServiceUri = baseUri +
@"/ServiceModel/AuthService.svc/Login";
// Path string for the BatchQuery.
private const string batchQueryUri = baseUri +
@"/0/DataService/json/reply/BatchQuery";
// Bpm'online authentication cookie.
private static CookieContainer AuthCookie = new CookieContainer();
```

Here, three string fields are declared. These fields will be used to form authentication query and read data queries execution paths. Authentication data will be saved in the AuthCookie field.

3. Add method that performs bpm'online application authentication

Authentication is required to enable access of the created application to the DataService.

Both the algorithm and example of implementation method, which contains the query to AuthService.svc for user authentication, are available in the "**Authenticating external requests to bpm'online services**" article.

4. Implement query adding request

Because the *batchQueryUri* constant declared previously earlier contains a path for sending data in the JSON format, sent data must be configured beforehand as a string that contains a JSON object description. Use data contract classes to create separate queries then serialize them in a string.

For a query to add a contact record with the name "John Smith", add the following program code:

```
// Insert query.
var insertQuery = new InsertQuery()
{
    RootSchemaName = "Contact",
    ColumnValues = new ColumnValues()
    {
        Items = new Dictionary<string, ColumnExpression>()
        {
            {
                "Name",
                new ColumnExpression()
                {
                    ExpressionType = EntitySchemaQueryExpressionType.Parameter,
                    Parameter = new Parameter
                    {
                        Value = "John Smith",
                        DataValueType = DataValueType.Text
                    }
                }
            }
        }
    }
```

```
};
```

For more information on the InsertQuery data contract please see the "**DataService. Adding records**" article.

To change the value of the [Business phone] column to 012 345 67 89 for all contact records that have "John Smith" value in the [Full name] column, add the following code:

```
 // Update query.
var updateQuery = new UpdateQuery()
{
    RootSchemaName = "Contact",
    ColumnValues = new ColumnValues()
    {
        Items = new Dictionary<string, ColumnExpression>()
        {
            {
                "Phone",
                new ColumnExpression()
                {
                    ExpressionType = EntitySchemaQueryExpressionType.Parameter,
                    Parameter = new Parameter()
                    {
                        Value = "0123456789",
                        DataValueType = DataValueType.Text
                    }
                }
            }
        }
    },
    Filters = new Filters()
    {
        FilterType = Terrasoft.Nui.ServiceModel.DataContract.FilterType.FilterGroup,
        Items = new Dictionary<string, Filter>()
        {
            {
                "FilterByName",
                new Filter
                {
                    FilterType =
Terrasoft.Nui.ServiceModel.DataContract.FilterType.CompareFilter,
                    ComparisonType = FilterComparisonType.Equal,
                    LeftExpression = new BaseExpression()
                    {
                        ExpressionType =
EntitySchemaQueryExpressionType.SchemaColumn,
                        ColumnPath = "Name"
                    },
                    RightExpression = new BaseExpression()
                    {
                        ExpressionType = EntitySchemaQueryExpressionType.Parameter,
                        Parameter = new Parameter()
                        {
                            DataValueType = DataValueType.Text,
                            Value = "John Smith"
                        }
                    }
                }
            }
        }
    }
};
```

For more information on the UpdateQuery data contract, please see the "**DataService. Updating records**"

article.

After serializing the created instances of the query class, add information about the qualified name of the corresponding data contract to the strings with JSON objects. Compose the string with batch query:

```
// Serialization of update query class instance in a JSON string.
var jsonInsert = new JavaScriptSerializer().Serialize(insertQuery);
// Inserting query type in a JSON string.
jsonInsert = jsonInsert.Insert(1, @"""__type"":
""Terrasoft.Nui.ServiceModel.DataContract.InsertQuery"",");
// Serialization of instance of the update query class in a JSON string.
var jsonUpdate = new JavaScriptSerializer().Serialize(updateQuery);
// Inserting query type in a JSON string.
jsonUpdate = jsonUpdate.Insert(1, @"""__type"":
""Terrasoft.Nui.ServiceModel.DataContract.UpdateQuery"",");
// Creating batch query.
var json = @"{""items"": [" + jsonInsert + "," + jsonUpdate + "]}";
```

The next step is to execute the POST DataService query. To do this, create an instance of the HttpWebRequest class, fill its properties and connect the string with JSON object, created earlier, then execute the DataService query and process its result. To do this, add the following source code:

```
// Converting a JSON object string in a byte array.
byte[] jsonArray = Encoding.UTF8.GetBytes(json);
// Creating an instance of HTTP request.
var batchRequest = HttpWebRequest.Create(deleteQueryUri) as HttpWebRequest;
// Defining request method.
batchRequest.Method = "POST";
// Determining request content.
batchRequest.ContentType = "application/json";
// Adding authentication cookie received earlier to a query.
batchRequest.CookieContainer = AuthCookie;
// Set the ContentLength property of the WebRequest.
batchRequest.ContentLength = jsonArray.Length;

// Adding JSON object to the query contents.
using (var requestStream = batchRequest.GetRequestStream())
{
    requestStream.Write(jsonArray, 0, jsonArray.Length);
}
// Executing HTTP request and getting reply from server.
using (var response = (HttpWebResponse)batchRequest.GetResponse())
{
    // Displaying response in console.
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        Console.WriteLine(reader.ReadToEnd());
    }
}
// Application execution delay.
Console.ReadKey();
```

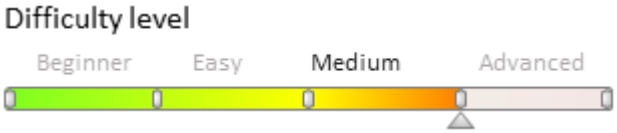
OData

Contents

- **Possibilities for the bpm'online integration over the OData protocol**
- **Working with bpm'online objects over the OData protocol using Http request**
- **Working with bpm'online objects over the OData protocol WCF-client**
- **Examples of requests for filter selection**

- **Executing OData queries using Fiddler**

Possibilities for the bpm'online integration over the OData protocol



General

Open Data (OData) protocol is an open web-protocol for requesting and updating data based on the REST architectural trekking using the Atom/XML and JSON standards .

The access to the bpm'online data and objects over the OData protocol may be received by any third-party application, which supports exchange with HTTP messages and may process XML or JSON data. In this case, data are available in the form of resources addressed over URI .

The access to data and its modification is implemented with the help of standard HTTP – GET, PUT/MERGE, POST and DELETE commands .

> ⚠ ATTENTION
>
> Using PUT and DELETE HTTP methods will cause “405 Method not allowed” error until WebDAV HTTP extension is switched off in application Web.Config.

Working over the OData has several features conditioned by the specifics of the REST approach :

- The application server does not store session status. All data required for a request processing is contained in the request itself .
- OData objects have the idempotence property. This means that a repeated action over an object does not modify the latter.
- When the GET request receives the object value, no modification of this or any other object must occur.

Today, a large number of client libraries for work with OData have been developed for popular application and mobile platforms, including:

- .NET
- Silverlight
- JavaScript/HTML5
- Java
- PHP
- Ruby
- WP7
- Android
- iOS

All client libraries for working with OData may be downlowded using the following link http://www.odata.org/libraries.

Implementation of the OData protocol in bpm'online

The bpm'online application supports the following operations with objects and their collections over the OData protocol:

Group of operations	Operations
Operations with objects	• Adding an object • Updating an object

	• Deleting an object • Adding relationships between objects • Deleting relationships between objects • Receiving metadata — description of all business entities • Receiving an objects collection • Receiving a specific object • Receiving a separate field of a specific object • Selecting several fields of a specific object • Extending an object with fields from lookup objects • Sorting objects • Receiving first N objects of the collection • Returning objects collections by bypassing first N objects (with N+1 of the object) • All arithmetical and logic operations supported by the protocol • Grouping filters
Functions of work with strings	• bool substringof(string p0, string p) • string toupper(string p0) • bool endswith(string p0, string p1) • bool startswith(string p0, string p1) • int length(string p0) • string trim(string p0)
Functions of work with date and time	• int year (DateTime p0) • int month(DateTime p0) • int day(DateTime p0)

NOTE

Please note that bpm'online implements the forced paging when returning resultant objects collections. A request returns first 40 objects by default. To modify the default paging implementation, standard structures of OData requests may be used: $top, $skip, $orderby .

Below are the examples of building requests for access to bpm'online objects over the OData protocol .

Work with bpm'online objects over the OData protocol

OData service for access to bpm'online objects

The access to bpm'online entities over the OData protocol is provided by the EntityDataService.svc web-service.

The address of the EntityDataService.svc service is as follows:

```
http[s]://<name_address_of_bpm'online>/0/ServiceModel/EntityDataService.svc
```

Example

```
https://myserver.com/BpmonlineWebApp/0/ServiceModel/EntityDataService.svc
```

The data model of the EntityDataService.svc service is described in its metadata which can be received using the standard OData syntaxes structure — $metadata.

Example

```
https://myserver.com/BpmonlineWebApp/0/ServiceModel/EntityDataService.svc/$metadata
```

Request authetication

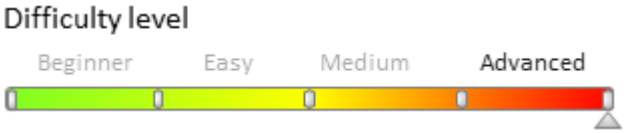
All requests to bpm'online must be authenticated .

The authentication methods supported by bpm'online are described in the article **Authenticating external requests to bpm'online services**.

Examples of implementing the access to bpm'online objects over the OData protocol

- **Working with bpm'online objects over the OData protocol using Http request**
- **Working with bpm'online objects over the OData protocol WCF-client**

Working with bpm'online objects over the OData protocol using Http request



The following issues will be considered in this article:

- **Operations of working with objects and object collections**
- **Functions of work with strings**
- **Functions of working with date and time**

To ensure successful compilation of the examples below, the following must be added to the software code:

Using directives

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Xml;
using System.Xml.Linq;
```

Declaration of variables and constants

```
// String of address bpm’online OData servise.
private const string serverUri = "http://<server_name>/<BPMonline
application_name>/0/ServiceModel/EntityDataService.svc/";
private const string authServiceUtri = "http://<server_name>/<BPMonline
application_name>/ServiceModel/AuthService.svc/Login";

// Links to XML name spaces.
private static readonly XNamespace ds =
"http://schemas.microsoft.com/ado/2007/08/dataservices";
private static readonly XNamespace dsmd =
"http://schemas.microsoft.com/ado/2007/08/dataservices/metadata";
private static readonly XNamespace atom = "http://www.w3.org/2005/Atom";
```

Operations of working with objects and object collections

Receiving the objects collection

To receive the object collection, the HTTP-method GET is used.

Records are returned by pages, 40 records per page. If a request is supposed to return more than 40 records, the

reception of the next page must be ensured to reach the end of the current page.

The example below demonstrates the use the $select structure to receive separate object fields (see OData Version 3.0 Core Protocol). The example shows that the request implementation results in the return of the contacts collection with the *Id* and *Name fields*.

The example below also uses *Forms* authentication. The bpm'online user name and password are transmitted in parameters of the GetOdataCollectionByAuthByHttpExample(string userName, string userPassword) method.

```
// Request string
// GET <BPMonline application
address>/0/ServiceModel/EntityDataService.svc/ContactCollection?select=Id,Name

public static void GetOdataCollectionByAuthByHttpExample(string userName, string
userPassword)
{
    // Creating an authentication request.
    var authRequest = HttpWebRequest.Create(authServiceUtri) as HttpWebRequest;
    authRequest.Method = "POST";
    authRequest.ContentType = "application/json";
    var bpmCookieContainer = new CookieContainer();
    // Including the cookie use into the request.
    authRequest.CookieContainer = bpmCookieContainer;
    // Receiving a stream associated with the authentication request.
    using (var requestStream = authRequest.GetRequestStream())
    {
        // Recording the BPMonline user accounts and additional request parameters
into the stream.
        using (var writer = new StreamWriter(requestStream))
        {
            writer.Write(@"{
                                ""UserName"":""" + userName + @""",
                                ""UserPassword"":""" + userPassword + @""",
                                ""SolutionName"":""TSBpm"",
                                ""TimeZoneOffset"":-120,
                                ""Language"":""En-us""
                                }");
        }
    }
    // Receiving an answer from the server. If the authentication is successful,
cookies will placed in the
    // bpmCookieContainer object and they may be used for further requests.
    using (var response = (HttpWebResponse)authRequest.GetResponse())
    {
        // Creating a request for data reception from the OData service.
        var dataRequest = HttpWebRequest.Create(serverUri + "ContactCollection?
select=Id, Name")
                                as HttpWebRequest;
        // The HTTP method GET is used to receive data.
        dataRequest.Method = "GET";
        // Adding pre-received authentication cookie to the data receipt request.
        dataRequest.CookieContainer = bpmCookieContainer;
        // Receiving a response from the server.
        using (var dataResponse = (HttpWebResponse)dataRequest.GetResponse())
        {
            // Uploading the server response to an xml-document for further
processing.
            XDocument xmlDoc = XDocument.Load(dataResponse.GetResponseStream());
            // Receiving the collection of contact objects that comply with the
request condition.
            var contacts = from entry in xmlDoc.Descendants(atom + "entry")
                           select new
                               {
```

```
                                    Id = new Guid(entry.Element(atom + "content")
                                                      .Element(dsmd + "properties")
                                                      .Element(ds + "Id").Value),
                                    Name = entry.Element(atom + "content")
                                                .Element(dsmd + "properties")
                                                .Element(ds + "Name").Value
                                };
            foreach (var contact in contacts)
            {
                // Implementing actions with contacts.
            }
        }
    }
```

If the request is required to return more than 40 records at once, this may be implemented using the $top parameter, where the required number of records returned by the request is specified. The example below forms a string of the request to the server to receive the first 60 objects of the contacts collection.

Example of using the $top parameter

```
string requestUri = serverUri + "ContactCollection?$top=60";
```

Bpm'online supports the use of the $skip parameter, which allows requesting resources from the service by bypassing the set number of records.

The example below demonstrates the formation of a string of the request to the service to receive the contacts collection starting with the eleventh record.

Example of using the $skip parameter

```
string requestUri = serverUri + "ContactCollection?$skip=10";
```

The service resources may be received in the sorted form. For this purpose, the $orderby [asc|desc] parameter must be used in a request. The field, by which the results are to be sorted, must be specified in the parameter. In addition, one of the following sorting directions may be specified for this parameter:

- ascending (*asc*)
- descending (*desc*)

The ascending sorting (*asc*) is used by default.

The example below forms a string of the request to the service to receive the contacts collection sorted by the ascending *Name* field value.

Example of using the $orderby parameter for ascending sorting

```
string requestUri = serverUri + "ContactCollection?$orderby=Name";
```

The $top, $skip, $orderby parameters may be used in various combinations to receive a certain fragment of the collection (see OData Version 3.0 Core Protocol).

Example of using the combined $orderby, $top, $skip parameters

```
string requestUri = serverUri + "ContactCollection?
$top=4&$skip=1&$orderby=City/Name";
```

Receiving an object with set features

The *HTTP-method GET* is used to receive an object.

A certain object which meets specific conditions (for example, contact with the set *Id* or account with a certain name, etc.) may be received by several methods (the examples below use *Basic* authentication of requests).

Setting the Id of the sought object as a parameter of the collection

```
// Request string:
// GET <BPMonline application
```

```
address>/0/ServiceModel/EntityDataService.svc/ContactCollection(guid'00000000-0000-
0000-0000-000000000000')

public static void GetOdataObjectByIdExample()
{
    // Id of the sought object.
    string contactId = "00000000-0000-0000-0000-000000000000";

    // Forming a string of the request to the service.
    string requestUri = serverUri + "ContactCollection(guid'" + contactId + "')";

    // Creating an object of the request to the service.
    var request = HttpWebRequest.Create(requestUri) as HttpWebRequest;
    request.Method = "GET";
    request.Credentials = new NetworkCredential("BPMUserName", "BPMUserPassword");
    using (var response = request.GetResponse())
    {
        // Receiving a response from the service in the xml format.
        XDocument xmlDoc = XDocument.Load(response.GetResponseStream());
        // Receiving the contact objects collection satsifying the request condition.
        var contacts = from entry in xmlDoc.Descendants(atom + "entry")
                       select new
                           {
                               Id = new Guid(entry.Element(atom + "content")
                                                  .Element(dsmd + "properties")
                                                  .Element(ds + "Id").Value),
                               Name = entry.Element(atom + "content")
                                           .Element(dsmd + "properties")
                                           .Element(ds + "Name").Value
                               // Initiating the object properties required for
further use.
                           };
        foreach (var contact in contacts)
        {
            // Implementing actions over the contact.
        }
    }
}
```

This method may be used only if an object with the set Id must be received.

If the sought object parameter is not Id or the sought object is determined by several parameters, the $filter structure must be used to determine the parameters.

Using the $filter structure to form a complex condition for the object selection

The $filter structure allows building logic expressions using the sought object selection conditions .

The $filter expressions may use links to properties and literals, strings, numbers and logical expressions (*true, false*). The $filter expressions use arithmetical, logical operations, grouping operations, operations with strings, date and time. The full list of operations implemented by the $filter structure is provided in the OData specification.

Below is the example of receiving the object with the set *Id*, using the $filter structure to set the condition.

```
// Request string:
// GET <BPMonline application
address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=Id eq
guid'00000000-0000-0000-0000-000000000000'

public static void GetOdataObjectByFilterConditionExample()
{
    // Id of the sought object.
```

```
    string contactId = "00000000-0000-0000-0000-000000000000";
    // Forming a string of the request to the service.
    string requestUri = serverUri + "ContactCollection?$filter = Id eq guid'" +
contactId + "'";
    // Creating an object of the request to the service.
    var request = HttpWebRequest.Create(requestUri) as HttpWebRequest;
    request.Method = "GET";
    request.Credentials = new NetworkCredential("BPMUserName", "BPMUserPassword");
    using (var response = request.GetResponse())
    {
        // Receiving a response from the service in the xml format.
        XDocument xmlDoc = XDocument.Load(response.GetResponseStream());
        // Receiving the contact objects collection satisfying the request condition.
        var contacts = from entry in xmlDoc.Descendants(atom + "entry")
                       select new
                       {
                           Id = new Guid(entry.Element(atom + "content")
                                              .Element(dsmd + "properties")
                                              .Element(ds + "Id").Value),
                           Name = entry.Element(atom + "content")
                                       .Element(dsmd + "properties")
                                       .Element(ds + "Name").Value
                           // Initiating the object properties required for further
use.
                       };
        foreach (var contact in contacts)
        {
           // Implementing actions over the contact.
        }
    }
}
```

Complex conditions for several fields of an object may be created using the $filter structure.

Below is the example of returning the contact objects collections created by a *SomeUserName* user after *2012-11-01*.

```
// Request string:
// GET <BPMonline applicationn
address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=CreatedOn gt
datetime'2012-11-01' and CreatedBy/Name eq 'SomeUserName'

public static void GetOdataObjectByFilterDiffConditionExample()
{
    // Name of the user that created the objects.
    string userName = "BPMUserName";
    // Objects creation date.
    string datetime = "2012-11-01";
    // Forming a string of the request to the service.
    string requestUri = serverUri + "ContactCollection?$filter=CreatedOn gt
datetime'" + datetime +
                                    "'and CreatedBy/Name eq '" + userName + "'";
    // Creating an object of the request to the service.
    var request = HttpWebRequest.Create(requestUri) as HttpWebRequest;
    request.Method = "GET";
    request.Credentials = new NetworkCredential(userName, "BPMUserPassword");
    using (var response = request.GetResponse())
    {
        // Receiving a response from the service in the xml format.
        XDocument xmlDoc = XDocument.Load(response.GetResponseStream());
        // Receiving the contact objects collection satisfying the request condition.
        var contacts = from entry in xmlDoc.Descendants(atom + "entry")
                       select new
```

```
                        {
                            Id = new Guid(entry.Element(atom + "content")
                                              .Element(dsmd + "properties")
                                              .Element(ds + "Id").Value),
                            Name = entry.Element(atom + "content")
                                        .Element(dsmd + "properties")
                                        .Element(ds + "Name").Value
                            // Initiating the object properties required for further
use.
                        };
            foreach (var contact in contacts)
            {
                // Implementing actions over the contact.
            }
        }
}
```

More examples of building requests using the $filter structure may be found in the article "**Examples of requests for filter selection**".

Creating a new object

The *HTTP-method POST* is used to create an object.

In this case, the request subject must be formed in the *Atom/XML* or *JSON* format so that it contains all required object fields. All possible fields of the created object are described in the service metadata .

Below is the example of creating a new contact. The example uses *Basic* authentication of the request .

```
// Request string:
// POST <BPMonline application
address>/0/ServiceModel/EntityDataService.svc/ContactCollection/

public static void CreateBpmEntityByOdataHttpExample()
{
    // Creating a xml message containing data on the created object.
    var content = new XElement(dsmd + "properties",
                  new XElement(ds + "Name", "Jhon Gilts"),
                  new XElement(ds + "Dear", "Jhon"));
    var entry = new XElement(atom + "entry",
                new XElement(atom + "content",
                new XAttribute("type", "application/xml"), content));
    Console.WriteLine(entry.ToString());
    // Creating a request to the service which will add a new object to the contacts
collection.
    var request = (HttpWebRequest)HttpWebRequest.Create(serverUri +
"ContactCollection/");
    request.Credentials = new NetworkCredential("BPMUserName", "BPMUserPassword");
    request.Method = "POST";
    request.Accept = "application/atom+xml";
    request.ContentType = "application/atom+xml;type=entry";
    // Recording the xml message to the request stream.
    using (var writer = XmlWriter.Create(request.GetRequestStream()))
    {
        entry.WriteTo(writer);
    }
    // Receiving a response from the service regarding the operation implementation
result.
    using (WebResponse response = request.GetResponse())
    {
        if (((HttpWebResponse)response).StatusCode == HttpStatusCode.Created)
        {
            // Processing the operation implementation result.
```

```
        }
    }
}
```

Modifying an existing object

The *PUT* (or *MERGE* in the latest OData versions) *HTTP-method* is used to modify a record.

New values of the fields to be modified are transmitted in the request subject. The collection whose object is modified must be specified in the request string and the modified object Id must be specified as the collection parameter.

Below is the example of modifying the name of the contact with the Id *oooooooo-oooo-oooo-oooo-oooooooooooo* from the *ContactCollection* contacts collection. The example uses the Basic authentication of requests.

```
// Request string:
// PUT <BPMonline application
address>/0/ServiceModel/EntityDataService.svc/ContactCollection(guid'00000000-0000-
0000-0000-000000000000')
// or
// MERGE <Адрес приложения
BPMonline>/0/ServiceModel/EntityDataService.svc/ContactCollection(guid'00000000-0000-
0000-0000-000000000000')
public static void UpdateExistingBpmEnyityByOdataHttpExample()
{
    // Id of the object record to be modified.
    string contactId = "00000000-0000-0000-0000-000000000000";
    // Creating an xml message containing data on the modified object.
    var content = new XElement(dsmd + "properties",
            new XElement(ds + "Name", "New name")
    );
    var entry = new XElement(atom + "entry",
            new XElement(atom + "content",
                    new XAttribute("type", "application/xml"),
                    content)
            );
    // Creating a request to the service which will modify the object data.
    var request = (HttpWebRequest)HttpWebRequest.Create(serverUri
            + "ContactCollection(guid'" + contactId + "')");
    request.Credentials = new NetworkCredential("BPMUserName", "BPMUserPassword");
    // or request.Method = "MERGE";
    request.Method = "PUT";
    request.Accept = "application/atom+xml";
    request.ContentType = "application/atom+xml;type=entry";
    // Recording the xml message to the request stream.
    using (var writer = XmlWriter.Create(request.GetRequestStream()))
    {
        entry.WriteTo(writer);
    }
    // Receiving a response from the service regarding the operation implementation
result.
    using (WebResponse response = request.GetResponse())
    {
        // Processing the operation implementation result.
    }
}
```

Deleting an object

The *HTTP-method DELETE* is used to delete a record .

The collection whose object is deleted must be specified in the request string and the deleted object Id must be specified as the collection parameter .

Below is the example of deleting the contact with the Id *ooooooo0-oooo-oooo-oooo-ooooooooooo0* from the *ContactCollection* contacts collection. The example uses the Basic authentication of requests .

```
// Request string:
// DELETE <BPMonline application
address>/0/ServiceModel/EntityDataService.svc/ContactCollection(guid'00000000-0000-
0000-0000-000000000000')

public static void DeleteBpmEntityByOdataHttpExample()
{
    // Id of the object record to be deleted.
    string contactId = "00000000-0000-0000-0000-000000000000";
    // Creating a request to the service which will delete the data.
    var request = (HttpWebRequest)HttpWebRequest.Create(serverUri
            + "ContactCollection(guid'" + contactId + "')");
    request.Credentials = new NetworkCredential("BPMUserName", "BPMUserPassword");
    request.Method = "DELETE";
    // Receiving a response from the service regarding the operation implementation
result.
    using (WebResponse response = request.GetResponse())
    {
        // Processing the operation implementation result.
    }
}
```

Functions of work with strings

BPMonline supports the following functions of work with the OData protocol strings which may be used for building expressions of the $filter structure.

Function	Example of the request string	Request implementation result
bool substringof(string po, string p)	<Service address >/ContactCollection?$filter=substringof('Smith', Name)	Collection of contacts whose name contains the 'Smith' sub-string.
string toupper(string p0)	<Service address > /ContactCollection?$filter=toupper(Name) eq 'TEST USER'	Collection of contacts whose name is equal to 'TEST USER' in the upper case.
bool endswith(string p0, string p1)	<Service address > /ContactCollection?$filter=endswith(Name, 'User')	Collection of contacts whose name ends with the 'User' sub-string.
int length(string p0)	<Service address > /ContactCollection?$filter=length(Name) gt 10	Collection of contacts whose name length exceeds 10 characters.
string trim(string p0)	<Service address > / ContactCollection?$filter=trim(Name) eq 'Test User'	Collection of contacts whose name is equal to 'Test User' after removing the initial and end spaces.

The full list of functions for working with the OData protocol strings is provided in the official OData specification.

Functions of working with date and time

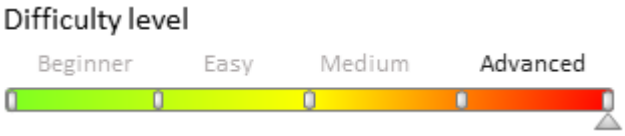
BPMonline supports the following functions of work with the OData protocol dates which may be used for building

expressions of the $filter structure.

Function	Example of the request string	Request implementation result
int year(DateTime po)	<Service address >/ContactCollection?$filter=year(BirthDate) ge 1950 and year(BirthDate) le 1990	Collection of contacts whose year of birth is within the range of the year 1950 to 1990, inclusive.
int month(DateTime po)	<Service address >/ContactCollection?$filter=month(BirthDate) eq 5	Collection of contacts born in May.
int day(DateTime po)	<Service address >/ContactCollection?$filter=day(BirthDate) eq 15	Collection of contacts born on the 15th day.

The full list of functions for working with the *OData* protocol dates is provided in the official OData specification.

Working with bpm'online objects over the OData protocol WCF-client



General

The access to bpm'online entities over the *OData* protocol is provided by the *EntityDataService.svc* web-service:

Address of the OData service

```
http[s]://<server_name>/<bpm'online_application_name> + "/0/ServiceModel/EntityDataService.svc/"
```

Example of the OData service address

```
http://myserver.com/bpmonlineWebApp/0/ServiceModel/EntityDataService.svc
```

Generating proxy classes of the EntityDataService.svc service

General

The key point in the organization of the WCF client operation is receiving metadata of the service and creating client proxy classes. A client application will use these mediation classes to exchange data with the web-service.

The following must be performed to implement the *.NET* client application that could work with the *OData* service of bpm'online:

1. Create a *.NET* project where the integration with bpm'online will be implemented.
2. Generate client proxy classes of the *EntityDataService.svc* service.
3. Create an instance of the execute environment for the *EntityDataService.svc* service.
4. Implement the client business logic of integration using the methods of the created proxy class instance.

Proxy classes may be generated on the client by several methods considered below.

Creating proxy classes using the DataServiceModel Metadata Utility Tool (DataSvcutil.exe) utility

DataSvcUtil.exe is a command string program provided by *WCF Data Services services*. The program uses the *OData* channel and forms client classes of the data service required for access to the data service from the .NET

Framework client application. This program forms data classes using the following sources of metadata:

- WSDL — a document of metadata of the service describing the data model provided by the data service.
- Data model file (CSDL) determined using the language for determining the conceptual schema (CSDL) described in the specification [MC–CSDL]: format of the file determining the conceptual schema.
- *EDMX* file created with the help of programs for work with the *EDM* model being a part of *Entity Framework*. Additional data are given in the specification [MC–EDMX]: EDM model for the data service package format.

The *DataSvcUtil.exe* program is installed in the .NET Framework catalogue.

This is usually the folder *C:\Windows\Microsoft.NET\Framework\v4.0*. For 64-bit system versions this is the folder *C:\Windows\Microsoft.NET\Framework64\v4.0*.

Format of calling the DataSvcutil.exe utility

```
datasvcutil /out:file [/in:file | /uri:serviceuri] [/dataservicecollection]
[/language:devlang] [/nologo] [/version:ver] [/help]
```

The *DataSvcutil.exe* utility is covered in the corresponding section of the MSDN.

Creating proxy classes in the project of the Visual Studio client application

Proxy classes for the *WCF* client may be created directly from *Visual Studio*. For this purpose, the following sequence of actions must be implemented:

1. Right-click the project where the integration with bpm'online must be implemented, select the *Add Service Reference...* item in the right-click menu
2. Enter the full address of the *OData* service, namely *EntityDataService.svc* in the *Address* field in the opened window.
3. Press the *Go* button. As a result, the service authentication window will open. The name and password of the bmp'online user must be specified in the window. If the authentication is successful, the entities supported by the service will be displayed in the *Services* window.
4. In the *Namespase* field, enter the name of the names space where generated proxy classes will be located. For example, bpm'onlineServiceReference. A link to this names space must be added afterwards to the *using* block of the project code.
5. Press the *OK* button. Proxy classes will be generated. A new *Reference.cs* code file containing the description of proxy classes will be added to the project. These classes may be used now for addressing and interacting with the data service resources as with objects.

> ⚠ *NOTE*
>
> Visual Studio may generate proxy classes of the service from the service metadata file saved on the disk. For this purpose, fulfill part. 1 of the instructions. Then, enter the full path to the metadata file with the prefix "file://" in the *Address* dialogue window.
>
> Example: file://C:/metadata.xml"
>
> After this, fulfill part. 3—5 of the instructions.

Examples of working with bmp'online entities in the WCF client

When proxy classes of the service are generated, a link to the *Microsoft.Data.Services.Client.dll* assembly is added to the project. This assembly supports the *OData v.3* protocol. If, by any reason, the earlier version of the protocol must be used in the client application, a link to the corresponding assembly must be added manually.

This client library allows making requests to the *EntityDataService.svc* data service using standard software templates .NET Framework, and include the use of the *LINQ* request language.

To ensure successful compilation of the examples below, the following must be added to the project code:

Using directives

```
using System;
using System.Data.Services.Client;
using System.Net;
using Terrasoft.Sdk.Examples.BPMonlineServiceReference;
using System.Linq;
```

Declaring the variable of the OData service address

```
private static Uri serverUri = new
Uri("http://<server_name>/<application_name>/0/ServiceModel/EntityDataService.svc/");
```

Receiving the objects collection of the service

To receive the objects collection of the service, the *DataServiceQuery* universal class is used. This class represents a request to the service, which returns the collection of a certain type of entities.

To implement the request to the *EntityDataService.svc* data service, an instance of the environment context object for the bmp'online must be created.

One ought to bear in mind that all external requests to bpm'online web-services must be authenticated. Detailed methods of authentication may be found in the article **Authenticating external requests to bpm'online services**.

Forms authentication implemented on the basis of the example in the clause above will be in further examples.

To implement the forms authentication, the LoginClass class with authServiceUri fields (string of a request to the Login method of the AuthService.svc authenticated service) and AuthCookie (Cookie authentications of bpm'online) were created. The TryLogin(string userName, string userPassword) method implementing the user authentication and saving the server response in the AuthCookie field may also be used.

In addition, the OnSendingRequestCookie (object sender, SendingRequestEventArgse) method is created. The method will be called in response to an event of the SendingRequest context instance (creating a new HttpWebRequest instance).

The OnSendingRequestCookie method authenticates the user and Cookies received in response are added to the data receipt request.

```
static void OnSendingRequestCookie(object sender, SendingRequestEventArgs e)
{
    // Calling the method of the LoginClass class, which authenticates the user
method transmitted in parameters.
    LoginClass.TryLogin("BPMUserName", "BPMUserPassword");
    var req = e.Request as HttpWebRequest;
    // Adding pre-received authentication cookie to the data receipt request.
    req.CookieContainer = LoginClass.AuthCookie;
    e.Request = req;
}
```

A request to the service may be implemented by one of the following methods:

- Implementing a *LINQ* request to the *DataServiceQuery* named object received from the service context.
- Implicit listing of the *DataServiceQuery* object received from the service context.
- Explicit call of the *Execute* method of the *DataServiceQuery* or *BeginExecute* object for asynchronous execution.

Below are the examples of access to *EntityDataService.svc* objects by one of the above methods.

1) Example of receiving the contacts collection via a LINQ request

This example demonstrates how the *LINQ* request returning all contact entities of the *EntityDataService.svc* service must be determined and implemented.

```
public static void GetOdataCollectioByLinqWcfExample()
{
    // Creating the context of the BPMonline application.
```

```
    var context = new BPMonline(serverUri);
    // Determining the method which adds authentication cookie when creating a new
request.
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>
(OnSendingRequestCookie);
    try
    {
        // Building a LINQ request to receive the contacts collection.
        var allContacts = from contacts in context.ContactCollection
                          select contacts;
        foreach (Contact contact in allContacts)
        {
            // Implementing actions with contacts.
        }
    }
    catch (Exception ex)
    {
        // Error processing.
    }
}
```

2) Example of receiving the contacts collection via an implicit request to the OData service via the context object

This example demonstrates how the context must be used to implement the implicit request returning all contact entities of the EntityDataService.svc service.

```
public static void GetOdataCollectionByImplicitRequestExample()
{
    // Creating a context object of the BPMonline application.
    var context = new BPMonline(serverUri);
    // Determining the method which adds authentication cookie when creating a new
request.
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>
(OnSendingRequestCookie);
    try
    {
        // Determining an implicit request to the service to receive the contacts
collection.
        DataServiceQuery<Contact> allContacts = context.ContactCollection;
        foreach (Contact contact in allContacts)
        {
            // Implementing actions with contacts.
        }
    }
    catch (Exception ex)
    {
        // Error processing.
    }
}
```

3) Example of receiving the contacts collection via an explicit request to the OData service via the context object

This example demonstrates how the *DataServiceContext* context must be used to implement a request to the *EntityDataService.svc* service returning all contact entities.

```
public static void GetOdataCollectionByExplicitRequestExample()
{
    // Determining a Uri request to the service which returns the contacts
collection.
    Uri contactUri = new Uri(serverUri, "ContactCollection");
```

```
    // Creating a context object of the BPMonline application.
    var context = new BPMonline(serverUri);
    // Determining the method which adds authentication cookie when creating a new
request.
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>
(OnSendingRequestCookie);
    try
    {
        // Implementing an explicit request to the service by calling the Execute<>()
method.
        foreach (Contact contact in context.Execute<Contact>(contactUri))
        {
            // Implementing actions with contacts.
        }
    }
    catch (Exception ex)
    {
        // Error processing.
    }
}
```

Receiving an object with set features

```
public static void GetOdataObjectByWcfExample()
{
    // Creating the context of the BPMonline application.
    var context = new BPMonline(serverUri);
    // Determining the method which adds authentication cookie when creating a new
request.
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>
(OnSendingRequestCookie);
    var contact = context.ContactCollection.Where(c =>
c.Name.Contains("User")).First();
    // Implementing actions over the contact.
}
```

Creating a new object

```
public static void CreateBpmEntityByOdataWcfExample()
{
    // Creating a new contact, initiating properties.
    var contact = new Contact()
    {
        Id = Guid.NewGuid(),
        Name = "New Test User"
    };
    // Creating and initiating properties of a new account, to which the created
contact refers.
    var account = new Account()
    {
        Id = Guid.NewGuid(),
        Name = "Some Company"
    };
    contact.Account = account;
    // Creating the context of the BPMonline application.
    var context = new BPMonline(serverUri);
    // Determining the method which adds authentication cookie when creating a new
request.
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>
(OnSendingRequestCookie);
    // Adding the created contact to the contacts collection of the service data
model.
```

```
    context.AddToAccountCollection(account);
    // Adding the created account to the accounts collection of the service data
model.
    context.AddToContactCollection(contact);
    // Setting the relationship between the created contact and account in the
service data model.
    context.SetLink(contact, "Account", account);
    // Saving the modification of data in BPMonline by one request.
    DataServiceResponse responces = context.SaveChanges(SaveChangesOptions.Batch);
    // Processing the server responses.
}
```

Modifying an existing object

```
public static void UpdateBpmEntityByOdatetWcfExample()
{
    // Creating the context of the BPMonline application.
    var context = new BPMonline(serverUri);
    // Determining the method which adds authentication cookie when creating a new
request.
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>
(OnSendingRequestCookie);
    // The contact on which basis the data will be modified is selected from the
contacts collection.
    var updateContact = context.ContactCollection.Where(c =>
c.Name.Contains("Test")).First();
    // Modifying the selected contact properties.
    updateContact.Notes = "New updated description for this contact.";
    updateContact.Phone = "123456789";
    // Saving the modifications in the service data model.
    context.UpdateObject(updateContact);
    // Saving the modification of data in BPMonline by one request.
    var responces = context.SaveChanges(SaveChangesOptions.Batch);
}
```

Deleting an object

```
public static void DeleteBpmEntityByOdataWcfExample()
{
    // Creating the context of the BPMonline application.
    var context = new BPMonline(serverUri);

    context.SendingRequest += new EventHandler<SendingRequestEventArgs>
(OnSendingRequestCookie);
    // The object which will be deleted is selected from the contacts collection.
    var deleteContact = context.ContactCollection.Where(c =>
c.Name.Contains("Test")).First();
    // Deleting the selected object from the service data model.
    context.DeleteObject(deleteContact);
    // Saving the modification of data in BPMonline by one request.
    var responces = context.SaveChanges(SaveChangesOptions.Batch);
    // Processing the server responses.
}
```

Examples of requests for filter selection

Difficulty level

We recommend familiarization with the article "**Working with bpm'online objects over the OData protocol using Http request**".

The structure $filter of the OData protocol is used to build up data filter conditions. The full list of operations implemented by the $filter structure is provided in the OData specification.

Links, literals, strings, numbers and logical expressions (*true, false*) may be used in $filter expressions. $filter expressions support arithmetical, logical operations, grouping operations, operations with strings, date and time .

Logical operators

Template	Name	Description
eq	Equals	All contacts whose [Name] field equals to the 'SomeUserName' value. Request string : GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=Name eq 'SomeUserName'
ne	Does not equal	All contacts whose [Name] field is not equal to the 'SomeUserName' value. Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=Name ne 'SomeUserName'
gt	More	All contacts whose [BirthDate] field exceeds the '1990-12- 12T12:00' value. Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=BirthDate gt datetime'1990-12-12T12:00'
ge	More or equals	All contacts whose [BirthDate] field exceeds or equals to the '1990-12-12T12:00' value. Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=BirthDate ge datetime'1990-12-12T12:00'
lt	Less	All contacts whose [BirthDate] field is less than the '1990-12- 12T12:00' value. Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=BirthDate lt datetime'1990-12-12T12:00'
le	Less or equals	All contacts whose [BirthDate] field is less or equals to the '1990-12-12T12:00' value. Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=BirthDate le datetime'1990-12-12T12:00'

Template	Name	Description
and	And	All contacts whose [Name] field equals to the 'SomeUserName' value and the [BirthDate] field is less than the '1990-12-12T12:00' value. Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=Name eq 'SomeUserName' and BirthDate le datetime'1990-12-12T12:00'
or	Or	All contacts whose [Name] field equals to the 'SomeUserName' value and the [BirthDate] field is less than the '1990-12-12T12:00' value. Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=Name eq 'SomeUserName' and BirthDate le datetime'1990-12-12T12:00'
not	Not	All contacts whose [Name] field does not end with 'ame'. Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ContactCollection?$filter=not endswith(Name, 'ame')

Arithmetic operators

Template	Name	Description
add	Addition	Select all products, for which the price (the [Price] field) satisfies the condition (Price + 2) = 35.55 Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ProductCollection?$filter=Price add 2.00m eq 35.55m
sub	Subtraction	Select all products, for which the price (the [Price] field) meets the condition (Price - 2) = 35.55 Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ProductCollection?$filter=Price sub 2.00m eq 35.55m
mul	Multiplication	Select all products, for which the price (the [Price] field) meets the condition (Price * 2) = 35.55 Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ProductCollection?$filter=Price mul 2.00m eq 35.55m
div	Division	Select all products, for which the price (the [Price] field) meets the condition (Price / 2) = 35.55 Request string: GET <BPMonline application

Template	Name	Description
		address>/0/ServiceModel/EntityDataService.svc/ProductCollection?$filter=Price div 2 eq 35.55m

To build up complex conditions of data filter in the $filters structure, various functions may be used:

- arithmetical;

- functions of work with strings;

- functions of work with date and time;

- functions of works with objects collection.

Learn more about these functions in the "**Working with bpm'online objects over the OData protocol using Http request**" article.

The full list of OData protocol functions is represented in OData official specification.

Operator any

Any operator applies logic expressions to each collection of items and returns a value of true, if an expression is correct for at least one collection item. Any operator will return a true value without an argument, if the collection contains at least one item.

Example.

Select all invoices that contain at least one product with the 'SomeProductName' name.

Request string:

GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/InvoiceCollection?$filter=InvoiceProductCollectionByInvoice/any(d:d/Product/Name eq 'SomeProductName')

OData protocol data types

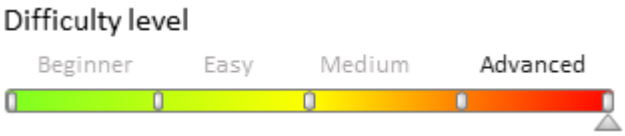
You should take into account data types that are filtered upon making queries. The literal character, located in the right part of a logical expression, should be of the same type as the field on the left part. The same rule applies to the use of mathematical functions. Strings, numbers, literal characters that are used in building of the expression, should have equal types of data.

Data type	Definition	Examples
Edm.Binary	Binary data of fixed or floating length. **Entry mask:** binary'[A-Fa-f0-9][A-Fa-f0-9]*' X'[Fa-f0-9][A-Fa-f0-9]*' NOTE X and binary are case-sensitive. The space should be absent between the functional word and the value.	Example: X'23AB' Example: binary'23ABFF'
Edm.Boolean	Represents logical value. **Entry mask:** true | false	Example: true Example: false
Edm.Byte	Represents 8-byte unsigned integer value. **Entry mask:**	Example: 255

Data type	Definition	Examples
	[0-9]+	
Edm.DateTime	Represents date and time within the range from 12:00;00 midnight, January 1, 1753 A.D. to 11:59:59 P.M., December 9999 A.D. **Entry mask:** datetime'yyyy-mm-ddThh:mm[:ss[.fffffff]]' NOTE datetime is case-sensitive. The space should be absent between the functional word and the value.	Example: datetime'2000-12-12T12:00'
Edm.Decimal	Numerical value with fixed accuracy. This type describes the value within the range from negative 10^255+1 to positive 10^255-1. **Entry mask:** [0-9]+.[0-9]+M|m	Example: 2.345M
Edm.Double	Represents numerical value with floating point up to 15 characters. **Entry mask:** [0-9]+ ((.[0-9]+) | [E[+ | -][0-9]+])d	Example: 1E+10d Example: 2.029d Example: 2.0d
Edm.Single	Represents numerical value with floating point up to 7 characters Entry mask. **Entry mask:** [0-9]+.[0-9]+f	Example: 2.0f
Edm.Guid	Represents 128-bit value, unique identifier. **Entry mask:** guid'dddddddd-dddd-dddd-dddd-dddddddddddd', where d — [A-Fa-f0-9] NOTE Guid is case-sensitive. The space should be absent between the functional word and the value.	Example: guid'12345678-aaaa-bbbb-cccc-ddddeeeeffff'
Edm.Int16	Represents signed 16-bit value. **Entry mask:** [-][0-9]+	Example: 16 Example: -16
Edm.Int32	Represents signed 32-bit integer value. **Entry mask:** [-] [0-9]+	Example: 32 Example: -32

Data type	Definition	Examples
Edm.Int64	Represents signed 64-bit integer value. **Entry mask:** [-] [0-9]+L	Example: 64L Example: -64L
Edm.SByte	Represents signed 8-bit integer value. **Entry mask:** [-] [0-9]+	Example: 8 Example: -8
Edm.String	Character data of floating or fixed length. **Entry mask:** '<any UTF-8 character>'	Example: 'Hello OData'

Executing OData queries using Fiddler



Introduction

Integration with bpm’online using the OData protocol requires executing HTTP requests to the *EntityDataService.svc*. Requests can be compiled in any programming language: C#, PHP, etc. However, it is recommended to use HTTP request debugging tools, such as PostMan or Fiddler for better understanding of general principles for request formatting. This article covers examples of requests composed with the help of Fiddler.

More information about OData protocol can be found in the “**Possibilities for the bpm'online integration over the OData protocol**” article.

Preliminary settings

ATTENTION

If the user who is making requests to *EntityDataService.svc* is not a member of the system administrator user group, add this user to the [Access to OData] group in the [Operation permissions] section (Fig. 1 and 2).

Also, enable access to object for external services for this user to work with bpm’online objects via OData (Fig. 3).

Fig. 1. The [Access to OData] group in the [Operation permissions] section

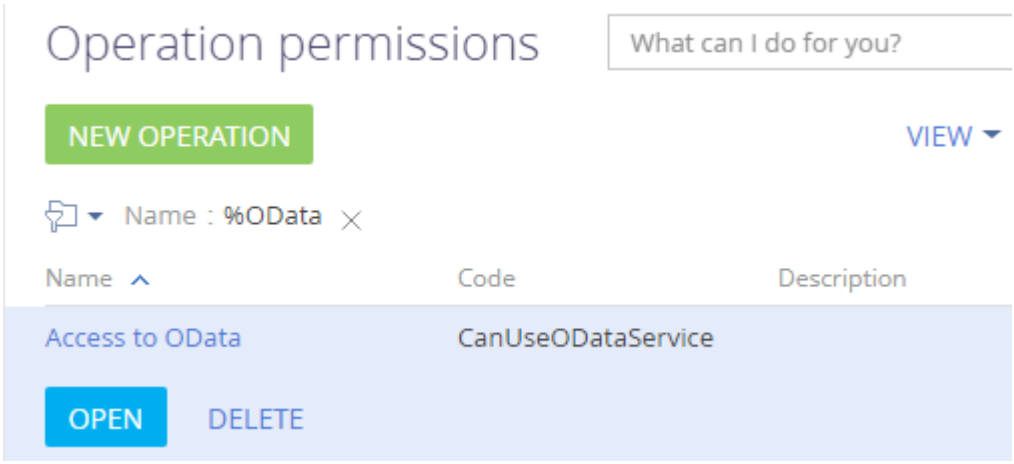


Fig. 2. User in the [Access to OData] group

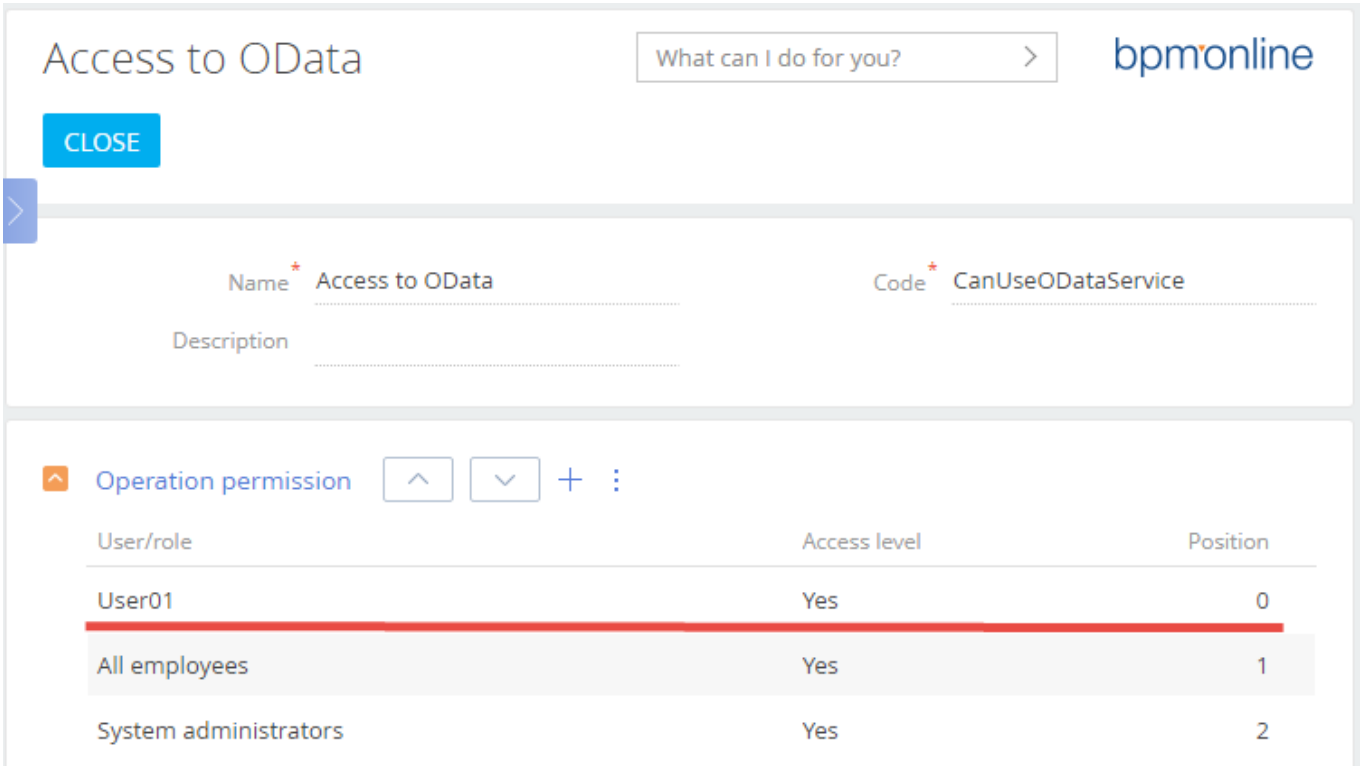
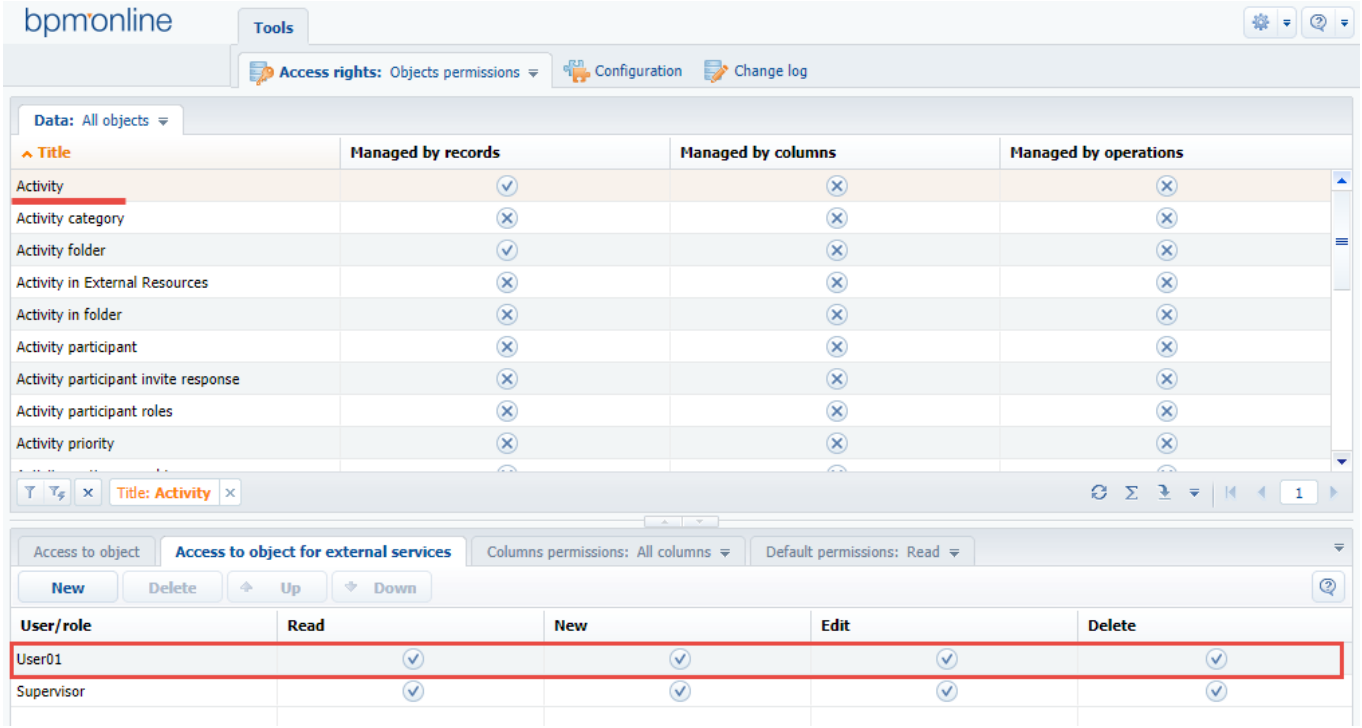


Fig. 3. Enabling access to object for external services for the [Activity] object



Authentification

Before making requests to *EntityDataService.svc*, a third party application must be authenticated and receive the corresponding *cookies*. Bpm’online’s authentication uses a separate web service: *AuthService.svc*. For more information about this service, please see the “**The AuthService.svc authentication service**” article.

To execute a request to *AuthService.svc* using Fiddler, go to the [Composer] tab and execute the following (Fig. 4):

1. Select HTTP method POST.

2. Specify the authentication service URL generated according to the following mask:

```
http(s)://[bpm'online application address]/ServiceModel/AuthService.svc/Login
```

Example:

```
https://012496-sales-enterprise.bpmonline.com/ServiceModel/AuthService.svc/Login
```

3. Specify HTTP protocol version 1.1.

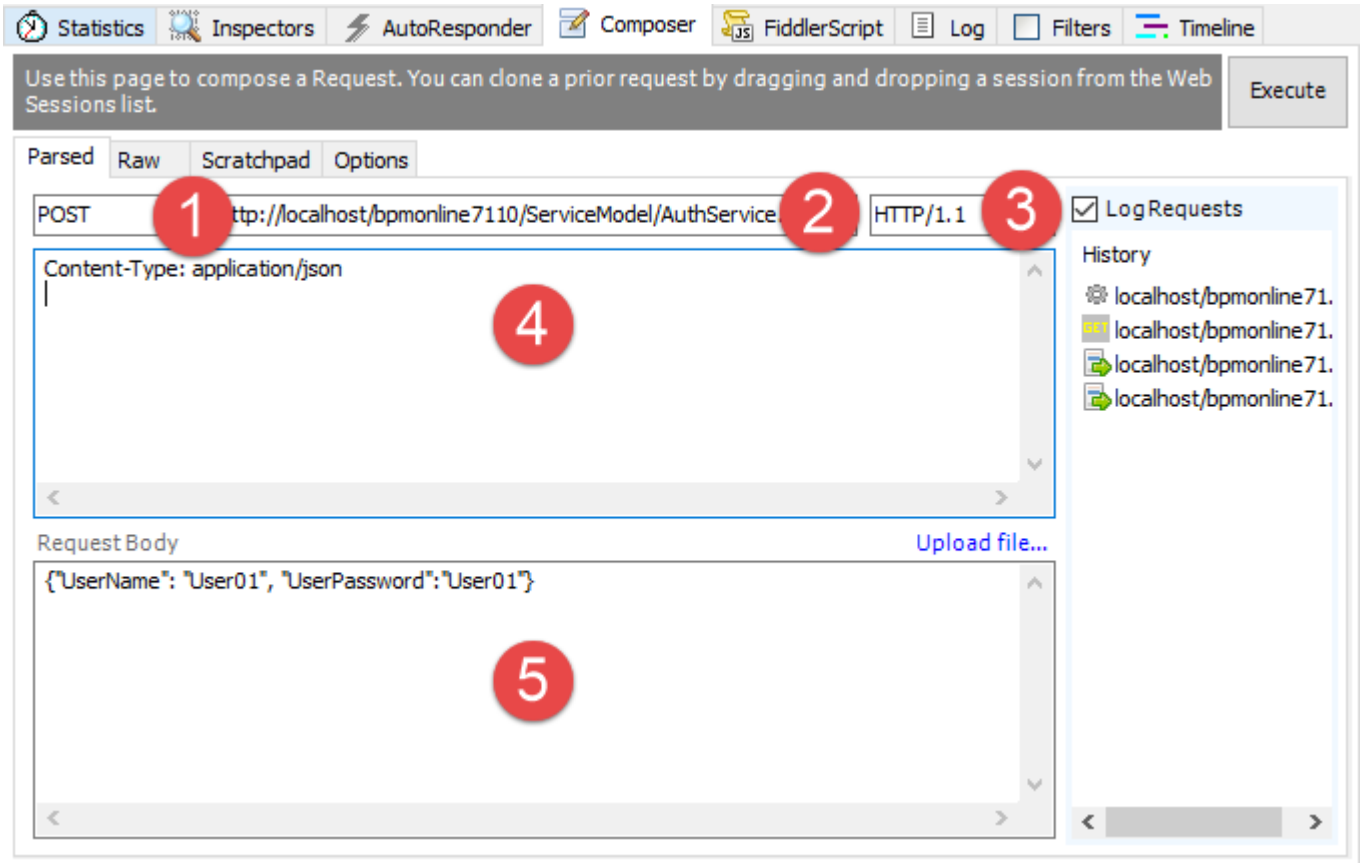
4. Specify the type of the request body:

```
Content-Type: application/json
```

5. Add the request body – a JSON object with the authentication data (login and password):

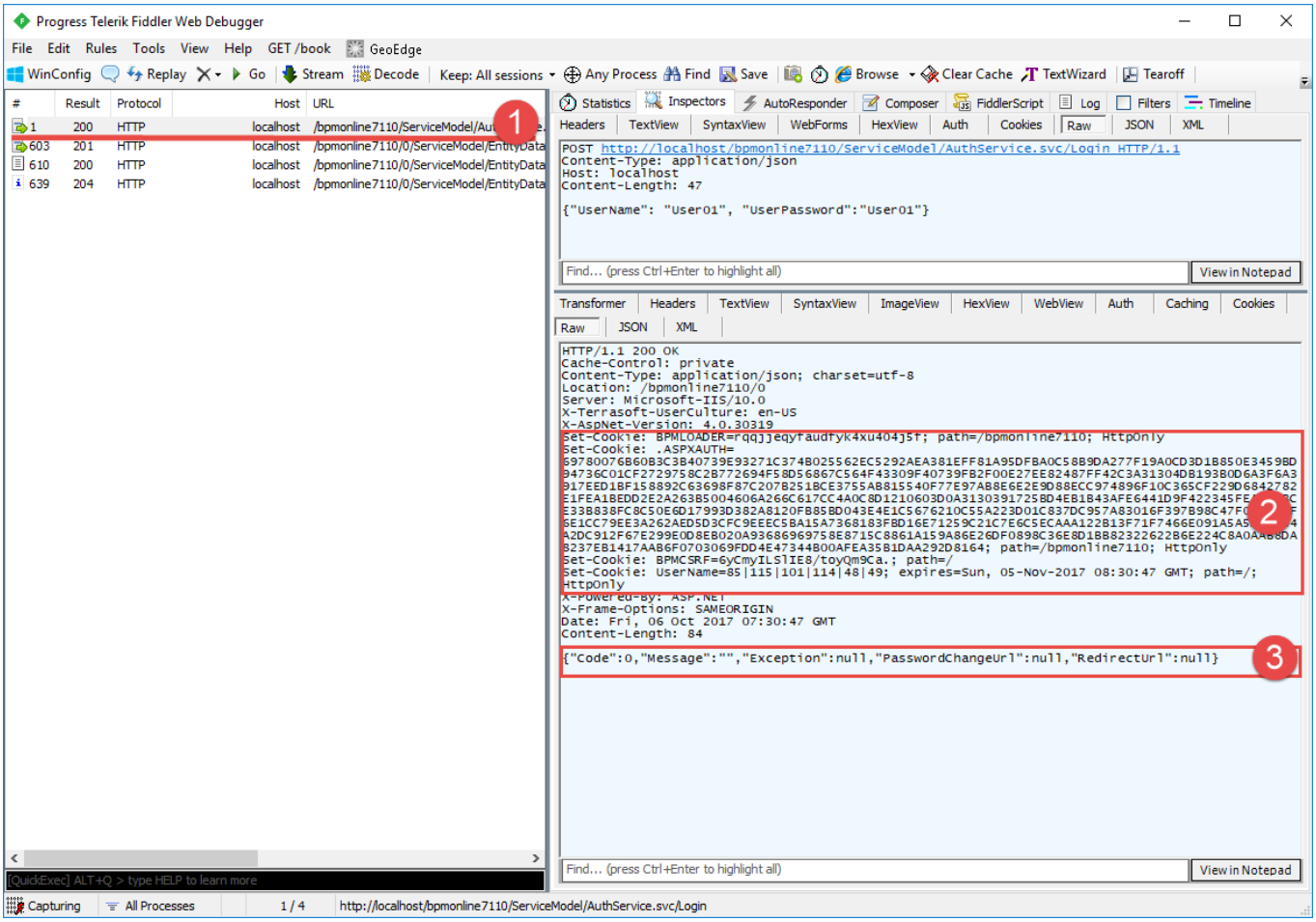
```
{"UserName": "User01", "UserPassword":"User01"}
```

Fig. 4. Generating authentication request



Execute the request by clicking the [Execute] button. As a result, the Fiddler session window will display a response from the *AuthService.svc* service (Fig. 5). Double-click the reply string (1) to open the [Inspectors] tab with the response properties.

Fig. 5. Properties of HTTP response from the AuthService.svc



> **ATTENTION**
>
> The *cookies* received in the HTTP response (BPMLOADER, .ASPXAUTH and BPMCSRF) are to be used in all further requests to bpm'online, that require authentication data (Fig. 5, 2).

> **ATTENTION**
>
> If the authentication has been successful, the response body will contain a JSON object whose *Code* property will be set to “0” (Fig. 5, 3). In case of errors, JSON obkect properties will contain corresponding code and message.

Adding data

Example: add a new activity. Fill out the [Title], [Owner] and [Notes] columns.

To compose a request to add such data using Fiddler, go to the [Composer] tab and execute the following (Fig. 6):

1. Select HTTP method POST.

2. Specify the *EntityDataService.svc* service URL generated according to the following mask:

```
http(s)://[bpm'online application
address]/0/ServiceModel/EntityDataService.svc/ActivityCollection/
```

3. Specify HTTP protocol version 1.1.

4. Add the following in the request title:

- Query content type – application/json.
- Required cookies (BPMLOADER, .ASPXAUTH, BPMSESSIONID и BPMCSRF).

- CSRF token BPMCSRF that contains the value of the corresponding *cookie* (BPMCSRF).

Example of request's HTTP title:

```
Accept: application/atom+xml
Content-Type: application/atom+xml;type=entry
Cookie: BPMSESSIONID=cxa54p2dsb4wnqbbzvgyxcoo; BPMCSRF=6yCmyILSlIE8/toyQm9Ca.;
BPMLOADER=rqqjjeqyfaudfyk4xu404j5f; .ASPXAUTH=697...A292D8164;
BPMCSRF: 6yCmyILSlIE8/toyQm9Ca.
```

> **ATTENTION**
>
> If protection from CSRF attacks is enabled, use both the BPMCSRF *cookie* and BPMCSRF token. For more information, see "**Protection from CSRF attacks during integration with bpm'online**".
>
> Protection from CSRF attacks is disabled on bpm'online trial websites. Therefore, there is no need to use both BPMCSRF *cookie* and token in the request titles.

> **ATTENTION**
>
> User session is created only upon the first request to the *EntityDataService.svc*, after which the BPMSESSIONID *cookie* will be returned in the response (Fig. 8, 2). Therefore, there is no need to add BPMSESSIONID *cookie* to the title of the first request (Fig. 6, 4).
>
> If you do not add BPMSESSIONID *cookie* to each subseqnent request, then each request will create a new user session. Significant frequency of requests (several or more requests a minute) will increase the RAM consumption which will decrease performance.

5. Add contents in XML format to the HTTP request body:

```
<?xml version="1.0" encoding="utf-8"?>
<entry xmlns="http://www.w3.org/2005/Atom">
    <content type="application/xml">
        <properties
xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
            <Title
xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices">process the incomming
website form request</Title>
            <Notes
xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices">please, email to
client@gmail.com and process the following request: clients request</Notes>
            <OwnerId
xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices">64844c83-c6c2-4eee-
a0e9-e26cef529d2f</OwnerId>
        </properties>
    </content>
</entry>
```

This request fills out all required object columns.

> **ATTENTION**
>
> If an object column is a lookup, specify the lookup database Id instead of lookup name. Add the "Id" suffix to the lookup column name in the request. In the current example, the lookup column is [Owner], and the "OwnerId" identifier is specified for it in the request.
>
> You can view the OwnerId value in the browser, by opening corresponding record for editing (Fig. 7) obtain via a query.

Fig. 6. Composing an insert query

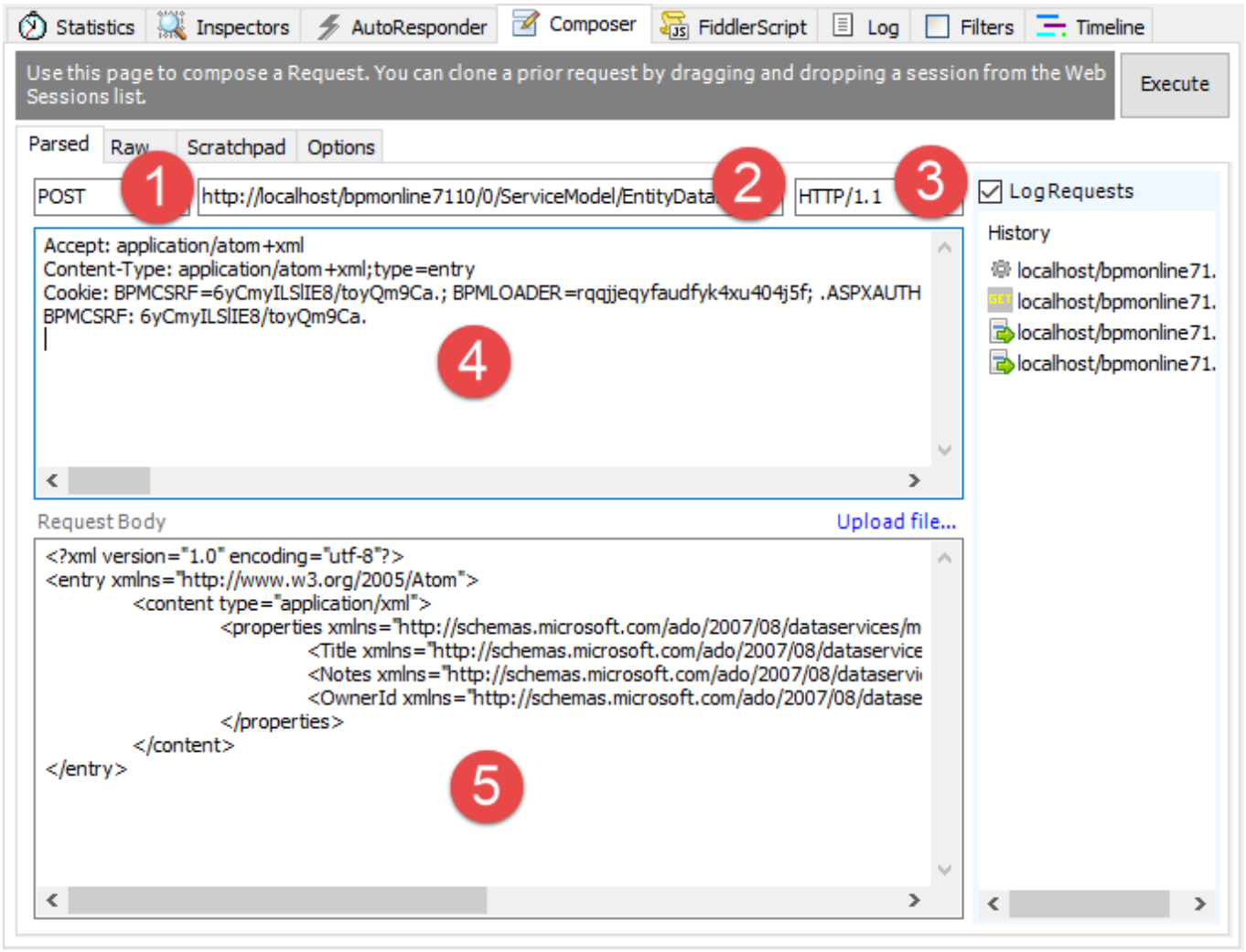
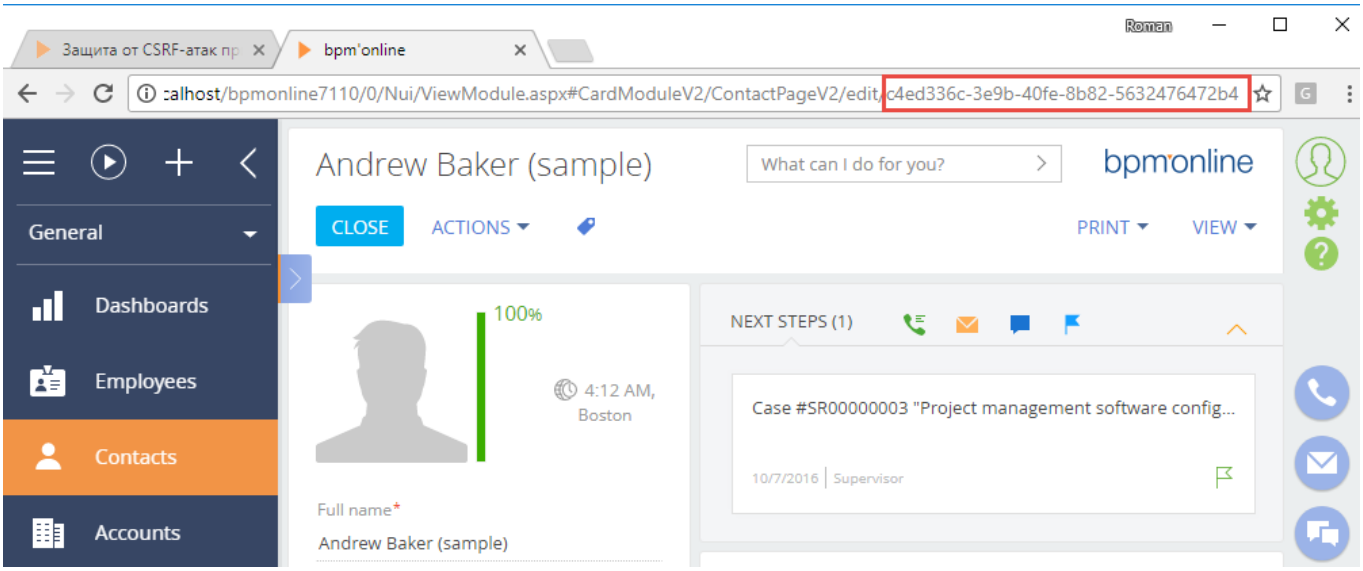
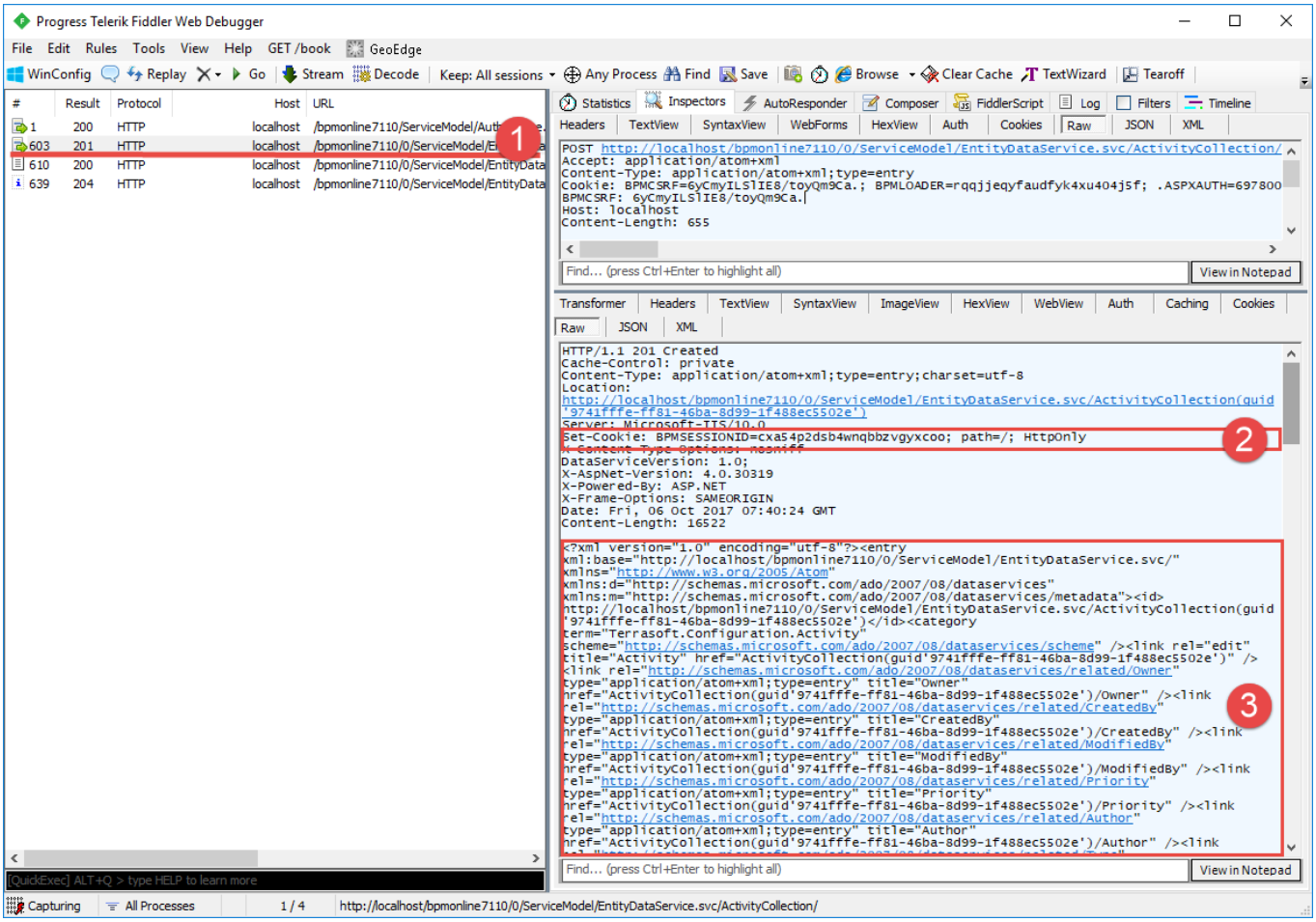


Fig. 7. Contact Id displayed in the browser



Execute the request by clicking the [Execute] button. As a result, the Fiddler session window will display a response from the *EntityDataService.svc* service (Fig. 8). Double-click the reply string (1) to open the [Inspectors] tab with the response properties.

Fig. 8. Properties of HTTP response from the EntityDataService.svc

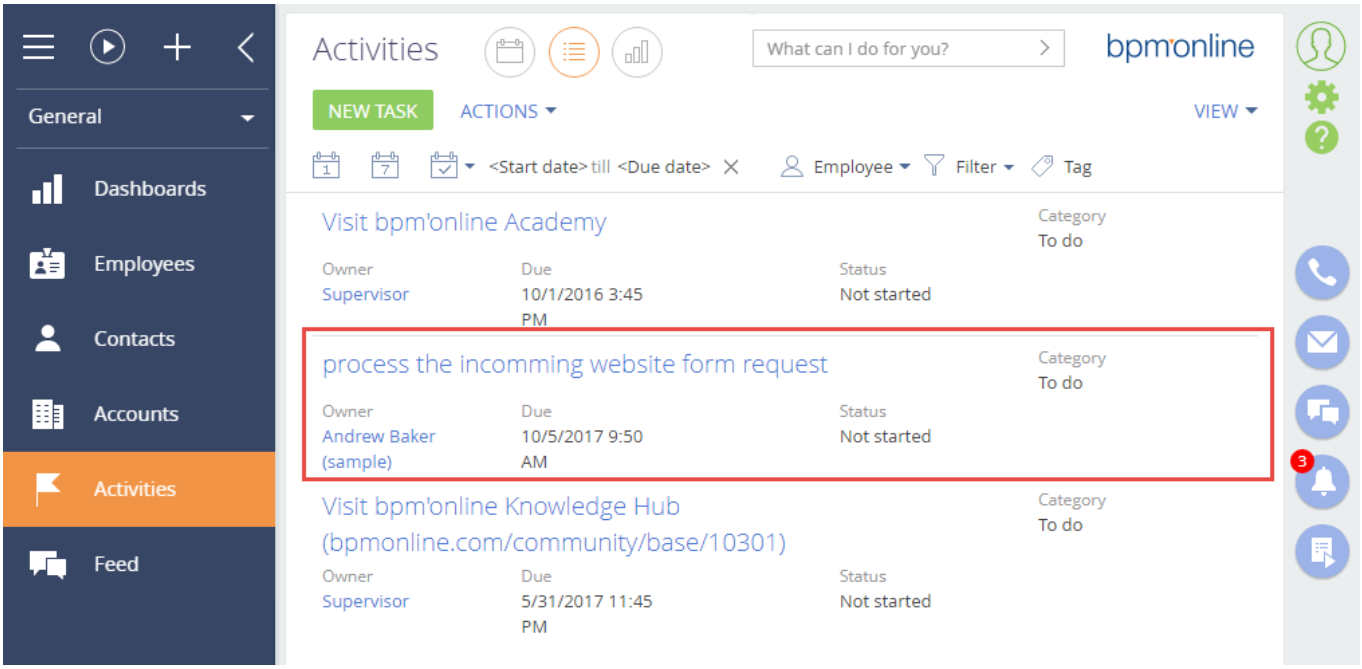


> **NOTE**
>
> The response from the *EntityDataService.svc* service may contain the BPMSESSIONID *cookie* if the request is executed for the first time (Fig. 8, 2).

The response body contains the added record in the XML format (Fig. 8, 3). The <id> XML element contains the identifier of the added activity, that can be used in other requests that need to work with this record.

As a result, a new record will be added in the [Activities] section (Fig. 9).

Fig. 9. Results of the activity add request



Data selection

The data select query does not have body. Data filtering is done based on the URL parameter values. For more information on the data select queries with filters, see the “**Examples of requests for filter selection**” article.

To compose a request to select data using Fiddler, go to the [Composer] tab and execute the following (Fig. 10):

1. Select HTTP method POST.

2. Specify the *EntityDataService.svc* service URL generated according to the following mask:

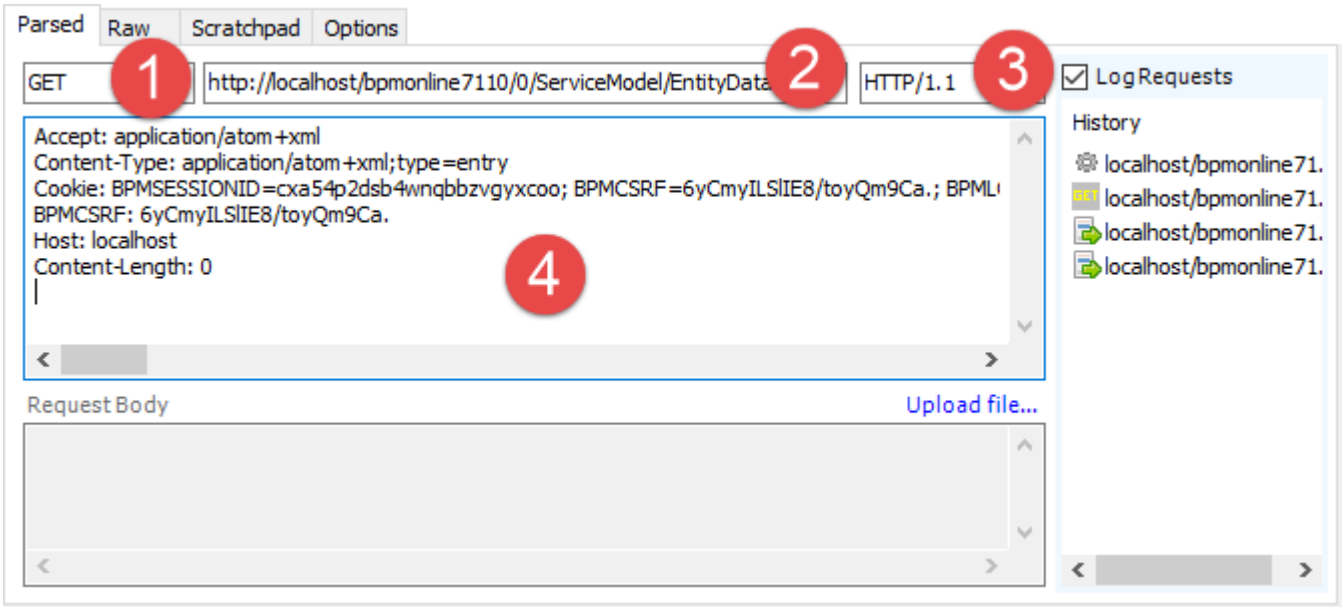
```
http(s)://[bpm'online application
address]/0/ServiceModel/EntityDataService.svc/ActivityStatusCollection?
$filter=Code%20eq%20'InProgress'
```

3. Specify HTTP protocol version 1.1.

4. In the request title, specify the request body type as application/atom+xml. Add all necessary cookies to the request title (BPMLOADER, .ASPXAUTH, BPMSESSIONID, BPMCSRF) and the BPMCSRF token:

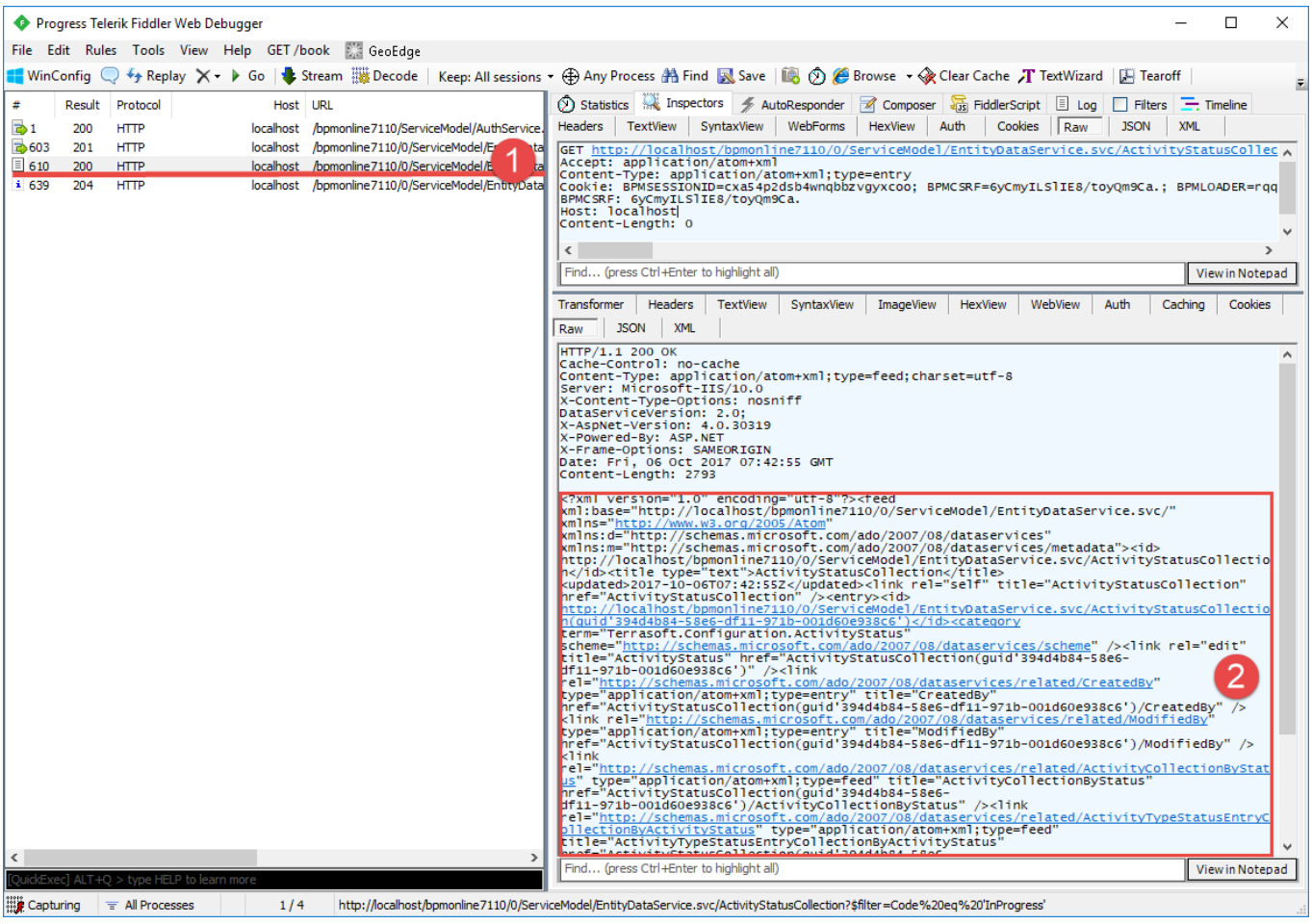
```
Accept: application/atom+xml
Content-Type: application/atom+xml;type=entry
Cookie: BPMSESSIONID=cxa54p2dsb4wnqbbzvgyxcoo; BPMCSRF=6yCmyILSlIE8/toyQm9Ca.;
BPMLOADER=rqqjjeqyfaudfyk4xu404j5f; .ASPXAUTH=697...A292D8164;
BPMCSRF: 6yCmyILSlIE8/toyQm9Ca.
```

Fig. 10. Composing data select query



Execute the request by clicking the [Execute] button. As a result, the Fiddler session window will display a response from the *EntityDataService.svc* service (Fig. 11). Double-click the reply string (1) to open the [Inspectors] tab with the response properties. The body of the HTTP response contains the selection result (2).

Fig. 11. Properties of HTTP response from the EntityDataService.svc



Update data

Example: modify the title of the added activity.

To compose a request to add data using Fiddler, go to the [Composer] tab and execute the following (Fig. 12):

1. Specify HTTP method POST.

> **ATTENTION**
>
> Using HTTP methods PUT and DELETE will cause the "405 Method not allowed” error if HTTP extension WebDAV is disabled in the application’s Web.Config file.

2. Specify the *EntityDataService.svc* service URL generated according to the following mask:

```
http(s)://[bpm'online application
address]/0/ServiceModel/EntityDataService.svc/ActivityCollection(guid'XXXXXXXX-XXXX-
XXXX-XXXX-XXXXXXXXXXXX')
```

Use the unique identifier of the added record (a record Id looks like this: 9741fffe-ff81-46ba-8d99-1f488ec5502e), which can be obtained in an HTTP response to rhe add request. You can also view record Id in a browser, by opening a record for editing (Fig. 13).

3. Specify HTTP protocol version 1.1.

4. In the request title, specify the request body type as application/atom+xml. Add all necessary cookies to the request title (BPMLOADER, .ASPXAUTH, BPMSESSIONID, BPMCSRF) and the BPMCSRF token:

```
Accept: application/atom+xml
Content-Type: application/atom+xml;type=entry
Cookie: BPMSESSIONID=cxa54p2dsb4wnqbbzvgyxcoo; BPMCSRF=6yCmyILSlIE8/toyQm9Ca.;
BPMLOADER=rqqjjeqyfaudfyk4xu404j5f; .ASPXAUTH=697...A292D8164;
BPMCSRF: 6yCmyILSlIE8/toyQm9Ca.
```

5. Add contents in the XML format to the request body.

```
<?xml version="1.0" encoding="utf-8"?>
<entry xmlns="http://www.w3.org/2005/Atom">
    <content type="application/xml">
        <properties
xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
            <Title
xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices">process the incomming
website form request (Updated)</Title>
        </properties>
    </content>
</entry>
```

> **NOTE**
>
> It is recommended to specify only columns that can be modified.

Fig. 12. Composing data update query

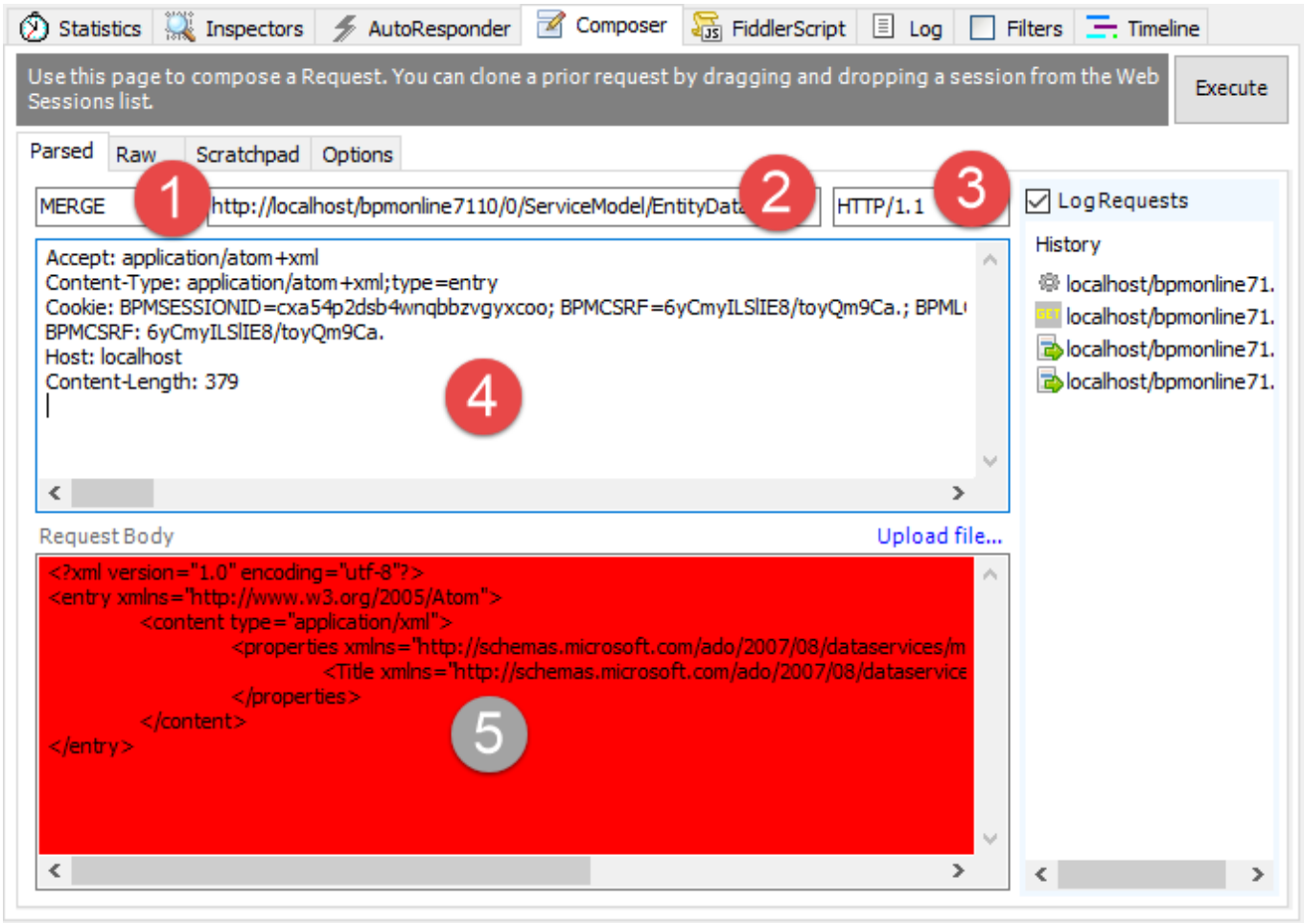
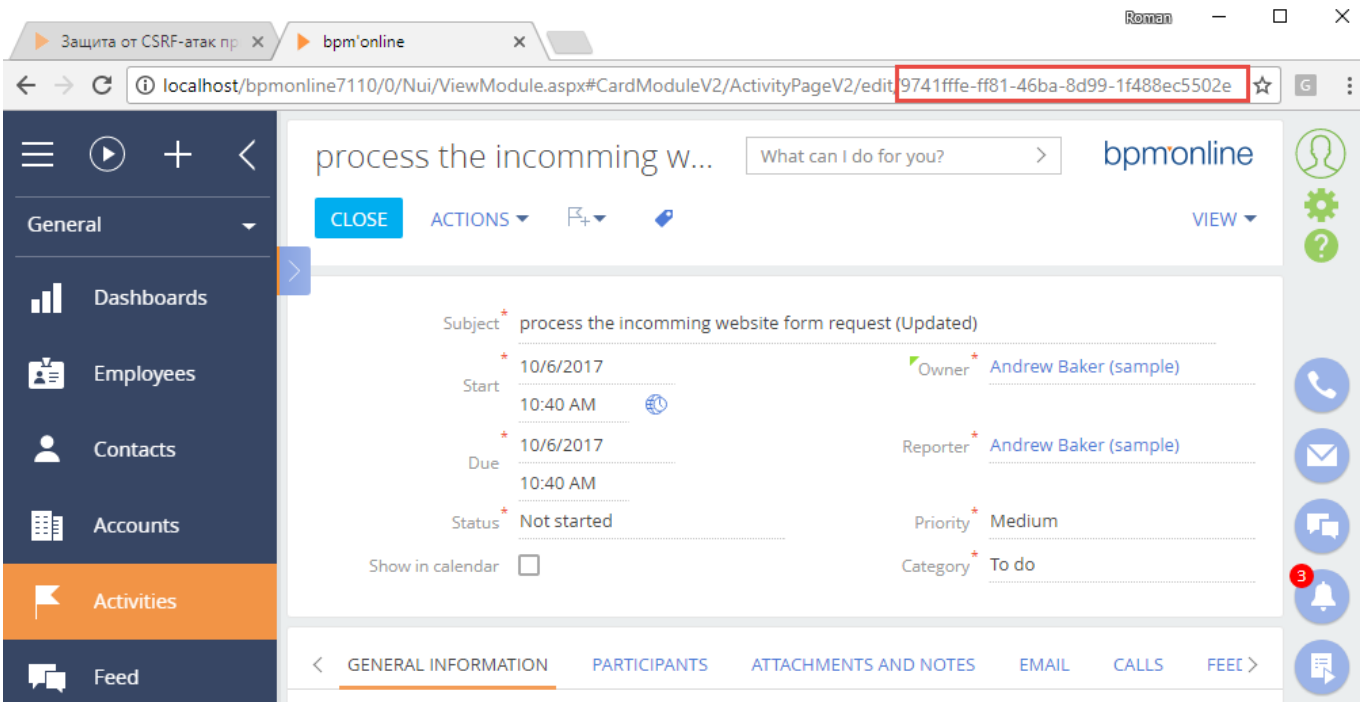
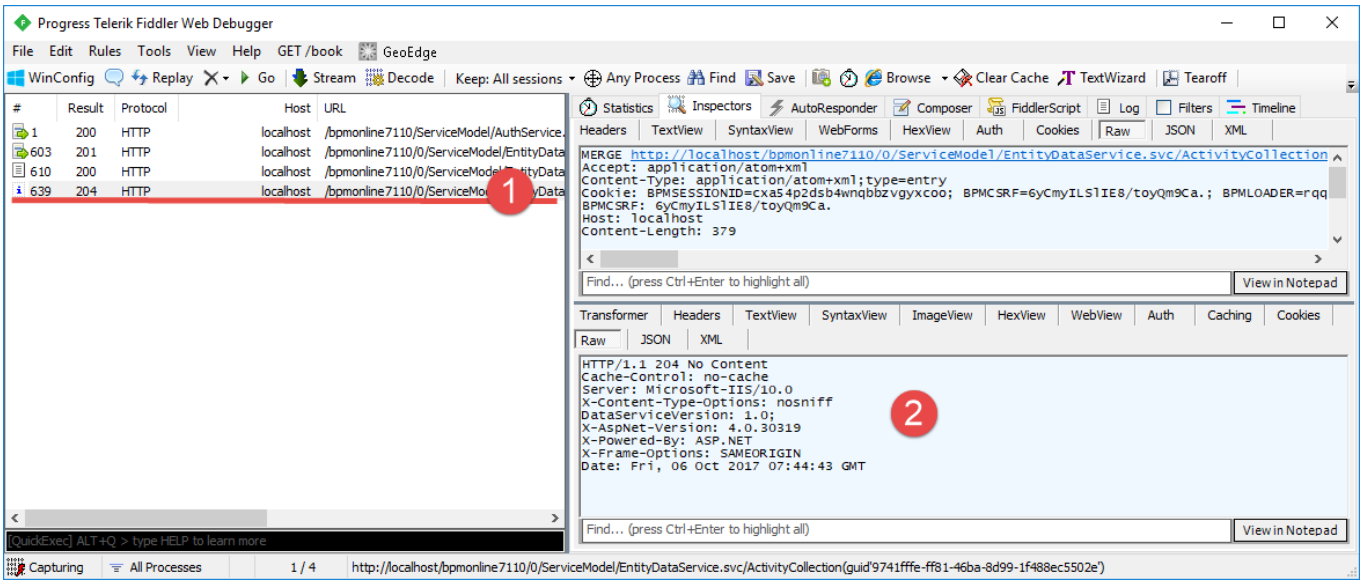


Fig. 13. Activity Id displayed in the browser



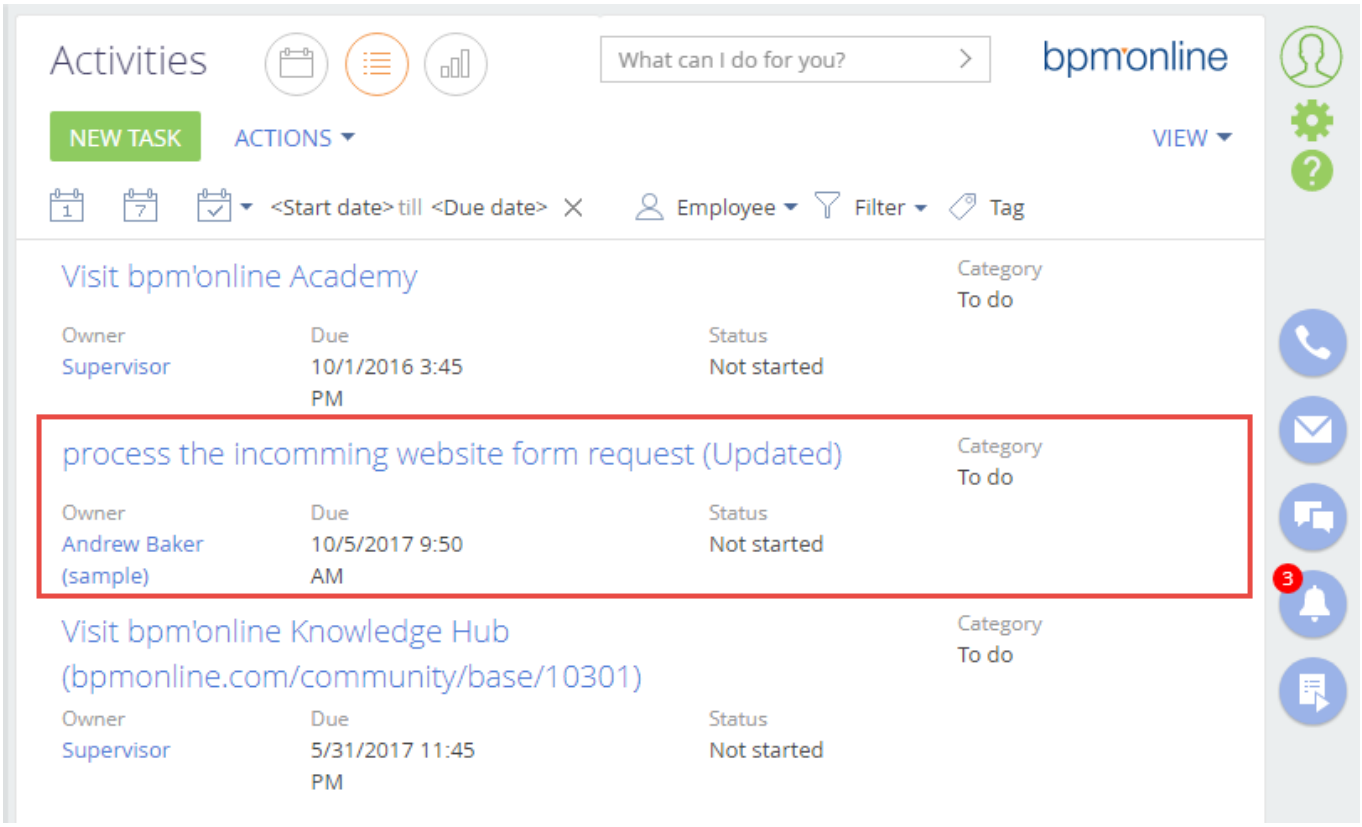
Execute the request by clicking the [Execute] button. As a result, the Fiddler session window will display a response from the *EntityDataService.svc* service (Fig. 14). Double-click the reply string (1) to open the [Inspectors] tab with the response properties. The body of the HTTP response is empty (2).

Fig. 14. Properties of HTTP response from the EntityDataService.svc



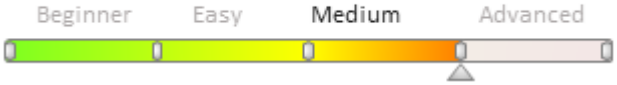
As a result, the title of the record will be modified (Fig. 15).

Fig. 15. Results of the activity edit request



Integration of third-party sites via iframe

Difficulty level



Introduction

One way to integrate external solutions in the bpm’online is to use the *iframe* HTML element.

The *iframe* HTML element is used to display third-party web page inside the web page where the element is placed. The *iframe* element is implemented in the HTML code of the page via the <*iframe*> and </*iframe*> tags. URL of the displayed page is set using the *src* attribute. More information about this element can be found in the article.

The third-party web application can be implemented to bpm’online with the *iframe* element. The advantage of this approach is the convenience of viewing the third-party web resources (pages, video, etc.) directly from the bpm’online. The main disadvantage is the need of a custom implementation of data exchange between the bpm’online and the web resource displayed in the *iframe*.

ATTENTION

Note, that some sites prohibit uploading of their pages into the *iframe* element.

ATTENTION

To exchange data between bpm'online and third-party web applications it is recommended to use the **DataService** service or the **OData** protocol.

The *Terrasoft.controls.IframeControl* component is implemented in the client part of the bpm’online core. This component is used to display custom HTML markup in the bpm’online. For example, it is used, on the email templates edit page of the [Email templates] lookup. The disadvantage of this component is the lack of the ability to bind data to the *src* property, that is, the inability to display a third-party web resource.

An alternative to using the *Terrasoft.controls.IframeControl* component is to add view model schemas of the common container to the **diff array** and specify the iframe element in it’s HTML property. The case of adding a container with the iframe element is described in the “Developing an advanced marketplace application” marketplace development article. The disadvantage of this approach is the inability to reuse the developed code in other sections of the bpm'online.

Integration case

Case description

Create a [WEB] tab on the record edit page in the [Accounts section]. The tab will contain a site which URL will be specified in the [Web] field.

Case implementation algorithm

1. Create a component where the displaying of the web page by specified URL will be implemented.

For this, create a new module in the custom package. The procedure for creating a module is covered in the “**Creating a custom client module schema**” article. Set the following parameter values for created module:

- [Name] — UsrIframeControl.
- [Title] — UsrIframeControl.

Create the *Terrasoft.controls.UsrIframeControl* class in this module using the *Ext.define()* method. The class should inherit the *Terrasoft.Component* as parent class. Main properties of the new class:

- *tpl* – an array of strings that contains a template of HTML markup of the component that will use the iframe element.
- *id* – a string that contains id of the component.
- *src* – a string that contains URL of the site to display in the iframe.

- *wrapClass* – a string with the name of the component CSS class.

The complete source code of the schema is available below:

```
Ext.define("Terrasoft.controls.UsrIframeControl", {
    extend: "Terrasoft.Component",
    alternateClassName: "Terrasoft.UsrIframeControl",
    // HTML template of a component.
    tpl: [
        /*jshint quotmark:true */
        '<iframe id="{id}" src="{src}" class="{wrapClass}"></iframe>'
        /*jshint quotmark:false */
    ],
    // Component ID.
    id: null,
    // URL of the website implemented in the iframe.
    src: null,
    // CSS class of the component.
    wrapClass: ["usr-iframe"],
    // Sets the URL of a website.
    setIframeSrc: function(value) {
        value = value || "";
        if (this.src !== value) {
            this.src = value;
            this.safeRerender();
        }
    },
    // Initializes a component.
    init: function() {
        this.callParent(arguments);
        var selectors = this.selectors = this.selectors || {};
        selectors.wrapEl = selectors.wrapEl || "#" + this.id;
    },
    // Loads a website to the iframe.
    LoadPageBySrc: function() {
        var iframe = this.getWrapEl();
        iframe.dom.src = this.src;
    },
    // The event handler for the first drawing of the component.
    onAfterRender: function() {
        this.callParent(arguments);
        this.LoadPageBySrc();
    },
    // The event handler of re-drawing of the component.
    onAfterReRender: function() {
        this.callParent(arguments);
        this.LoadPageBySrc();
    },
    // Returns the configuration object of binding the component poperties.
    getBindConfig: function() {
        var bindConfig = this.callParent(arguments);
        return Ext.apply(bindConfig, {
            src: {
                changeMethod: "setIframeSrc"
            }
        });
    },
    // Returns data about the component template.
    getTplData: function() {
        var tplData = this.callParent(arguments);
        return Ext.apply(tplData, {
            src: this.src,
```

```
            wrapClass: this.wrapClass
        });
    }
});
```

Add the CSS styles for correct displaying of the component. To do this, add the following code to the LESS tab of the created module:

```
.usr-iframe {
width: 100%;
height: 600px;
}
```

Save the module schema to apply changes.

2. Place the component on the record edit page of the [Accounts] section.

For this, create the [Account edit page] replacing schema in the custom package. The procedure for creating a replacing schema is covered in the "**Creating a custom client module schema**". Add the following source code to the replacing schema:

```
// Add a module in which the component is implemented in an array of dependencies.
define("AccountPageV2", ["UsrIframeControl", "css!UsrIframeControl"], function() {
    return {
        entitySchemaName: "Account",
        diff: /**SCHEMA_DIFF*/[
            // Adding the [WEB] tab.
            {
                "operation": "insert",
                "name": "WebTab",
                "values": {
                    "caption": "WEB",
                    "items": []
                },
                "parentName": "Tabs",
                "propertyName": "tabs",
                "index": 1
            },
            // Adding a custom component.
            {
                "operation": "insert",
                "parentName": "WebTab",
                "propertyName": "items",
                "name": "UsrIframe",
                "values": {
                    "generator": function() {
                        return {
                            "className": "Terrasoft.UsrIframeControl",
                            "src": {"bindTo": "getSource"}
                        };
                    }
                }
            }
        ]/**SCHEMA_DIFF*/,
        methods: {
            // Used to bind data.
            getSource: function() {
                return this.get("Web");
            }
        }
    };
```

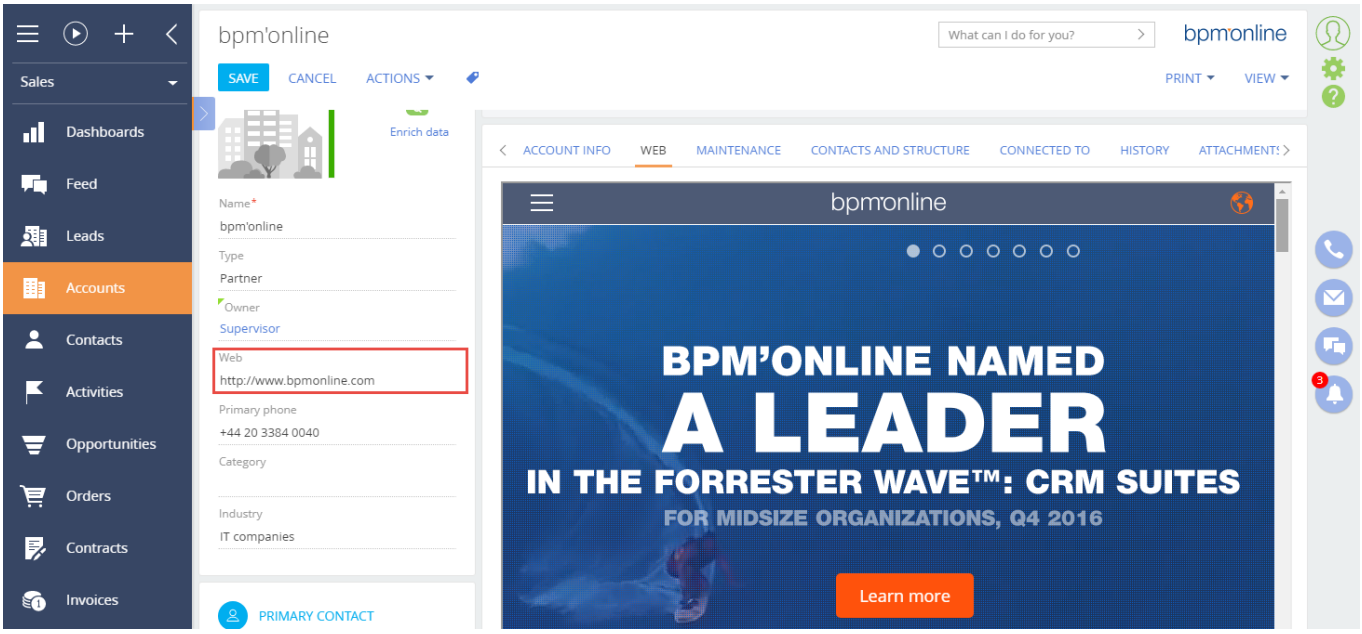
```
});
```

Here, the configuration objects of the [WEB] tab and the custom component for displaying a component are added to the array of modifications of the view model

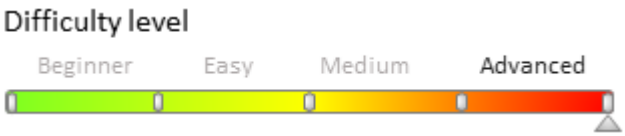
Terrasoft.UsrIframeControl. Binding the data of the [Web] column to the *src* property of the component is performed with the *getSource()* method.

After saving the schema and reloading the application page, the [WEB] tab will appear on the edit page of a section record.The tab will display a web page by the URL specified in the [Web] field (Fig. 1). If the [Web] field is empty then the tab will be empty too.

Fig. 1. Case result



Web-To-Object. Using landings and web-forms



Introduction

Web-to-Object is a mechanism for implementing simple one-way integrations with bpm’online. It enables you to create records in bpm'online sections (leads, cases, orders, etc.) simply by sending the necessary data to the Web-to-Object service.

The most common cases of using the Web-to-Object service are the following:

- Integrating bpm'online with custom landings and web forms. The service call is performed from a landing (a customized custom page with a web form), after the visitor submits the completed web form.
- Integrating with external systems to create bpm'online objects.

Using Web-to-Object will enable you to configure the registration of virtually any objects in bpm'online (e.g., a lead, an order or a case).

The [Landings and web-forms] section is used to work with landing in bpm’online. This section is available in all bpm’online products, however it might not be enabled in workplaces of certain products (e.g., bpm’online sales). Each record in the [Landing and web-forms] section corresponds to a landing page. The record edit page features a [Landing setup] tab.

Learn more about the process of creating landing pages based on the Web-to-Case mechanism in the “**Creating**

Web-to-Case landing pages” article.

Implementing the Web-to-Object service

The main functionality of the Web-To-Object mechanism is contained in the *WebForm* package and is common to all products. Depending on the product, this functionality may be extended by the Web-to-Lead (the *WebLeadForm* package), Web-to-Order (the *WebOrderForm* package), and **Web-to-Case** (the *WebCaseForm* package) mechanisms.

To process data received from a web-form of a lending, the *WebForm* package implements the *GeneratedObjectWebFormService* configuration service (the *Terrasoft.Configuration.GeneratedWebFormService* class). The data of the landing’s web-form is accepted as the argument of the *public string SaveWebFormObjectData(FormData formData)* method. They are then passed to the *public void HandleForm(FormData formData)* method of the *Terrasoft.Configuration.WebFormHandler* class, in which the corresponding system object is created.

External API of the Web-to-Object service

To use the service, send a POST request to:

```
[Bpm'online application
path]/0/ServiceModel/GeneratedObjectWebFormService.svc/SaveWebFormObjectData
```

For example:

```
http://mybpmonline.com/0/ServiceModel/GeneratedObjectWebFormService.svc/SaveWebFormObjectData
```

The content type of the request is application/json. In addition to the required cookies, the JSON object containing the data of the web-form must be added to the content of the request. JSON object example:

```
{
    "formData":{
        "formId":"d66ebbf6-f588-4130-9f0b-0ba3414dafb8",
        "formFieldsData":[
            {"name":"Name","value":"John Smith"},
            {"name":"Email","value":"j.smith@bpmoline.com"},
            {"name":"Zip","value":"00000"},
            {"name":"MobilePhone","value":"0123456789"},
            {"name":"Company","value":"bpmonline"},
            {"name":"Industry","value":""},
            {"name":"FullJobTitle","value":"Sales Manager"},
            {"name":"UseEmail","value":""},
            {"name":"City","value":"Boston"},
            {"name":"Country","value":"USA"},
            {"name":"Commentary","value":""},
            {"name":"BpmHref","value":"http://localhost/Landing/"},
            {"name":"BpmSessionId","value":"0ca32d6d-5d60-9444-ec34-5591514b27a3"}
        ]
    }
}
```

Using the Web-to-Object service

Integrating with custom landings and web-forms

Integrating bpm'online with custom landings and web-forms is covered in the following articles:

- The [Landing and web-forms] section
- **Creating Web-to-Case landing pages**

Integrating with external systems

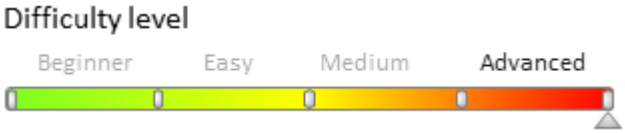
To integrate with external systems:

1. Create a new record in the [Landing and web-forms] section

2. Get the address to the service (*serviceUrl* property) and the identifier (the *landingId* property) from the configuration object of the created record.

3. Implement the process of sending a POST-request to the Web-to-Object service (at the received address) in the external system. Add the necessary data to the request in form of a JSON object. Set the value of the received identifier to the *formId* property of the JSON object.

The ProcessEngineService.svc web service



Introduction

Running the business processes is one of the purpose of integration the external application with bpm’online. The *ProcessEngineService.svc* web service that allows to start business processes from outside is implemented for this. The *ProcessEngineService.svc* web service is available by following URL*:*

```
http[s]://<bpm'online_application_address>/0/ServiceModel/ProcessEngineService.svc
```

> Attention
>
> Before calling a web-service via external tools, authenticate the user, whose name will be used to execute queries. For this, use the AuthService.svc service (see. "**The AuthService.svc authentication service**" and "**Authenticating external requests to bpm'online services**" articles). This service will return the correspondent cookies, which you have to use when querying the ProcessEngineService.svc.
>
> Moreover, if the SCRF attacks defense is enabled in your application, add the BPMCSRF token to the query headers (see. "**Protection from CSRF attacks during integration with bpm'online**").
>
> See the examples of executing queries to bpm'online web-services after user authentication in the "**Executing OData queries using Fiddler**".

> NOTE
>
> A ProcessEngineService.svc use case is available in the "**How to run bpm'online processes via web service**" article. The full list of the web service methods can be found in the **.NET class libraries of platform core (on-line documentation)** documentation.

ProcessEngineService.svc methods

Business process launch

To start specific business process you will need to call the *Execute()* method of the service. You can call the *Execute()* method via GET and POST HTTP requests. General format of the *Execute()* method:

```
http[s]://<bpm'online_application_address>/0/ServiceModel/ProcessEngineService.svc/PR
OCESSSCHEMANAME/Execute[?<optional incoming parameters of the business process>]
```

Where PROCESSSCHEMANAME – the name of the business process schema in the bpm’online.

> NOTE
>
> The name of the business process schema can be found in the [Configuration] section.

For example, start the *UsrSomeProcess* process. The *procParam* parameter with the 15 as a value is passed to the process. The GET string will be as follows:

```
.../0/ServiceModel/ProcessEngineService.svc/UsrSomeProcess/Execute?procParam=15
```

The *ProcessEngineService.svc* process enables to start specific business process and get the result of execution of this process with the specific parameter. For this, call the *Execute()* method in the following format:

```
http[s]://<bpm'online_application_address>/0/ServiceModel/ProcessEngineService.svc/PR
OCESSSCHEMANAME/Execute?ResultParameterName=RESULTPARAMETERNAME[&<optional incoming
parameters of the business process>],
```

where

- PROCESSSCHEMANAME – the name of the business process schema which instance will be launched
- RESULTPARAMETERNAME – the name of the process parameter, that stores the result of the process execution. If this parameter is not specified, the web service will launch the specified business process without waiting its execution result.

> ⚠ ATTENTION
>
> If the RESULTPARAMETERNAME parameter is disabled in the called process, the web service will return *null*.

For example, start the *CustomProcess* process. The process result is stored in the *CustomProcessResult* outgoing process parameter. In addition, the *incomeParam* parameter with the *"IncomeParamValue"* value is passed to the *CustomProcess* process. The GET string will as follows:

```
.../0/ServiceModel/ProcessEngineService.svc/CustomProcess/Execute?
ResultParameterName=CustomProcessResult&incomeParam=IncomeParamValue
```

The result of executing the *Execute()* method is returned as a string containing the JSON object (it is possible to get the *null*). Deserialization of the JSON object and bringing the result to a specific type of data must be performed in the code that calls the web service.

Executing a separate element of the business process

To execute a separate element of the business process, call the *ExecProcElByUId()* web service method. This method accepts the Id of the executed process element as a parameter. The format of the *ExecProcElByUId()*method call:

```
http[s]://<bpm'online_application_address>/0/ServiceModel/ProcessEngineService.svc/Ex
ecProcElByUId/PROCELUID
```

where PROCELUID – id of the executed process element.

> ⚠ ATTENTION
>
> Only the element of the launched process can be executed.
>
> If the *ExecProcElByUId()* process element has been already completed at the moment of calling the method, this element will not be executed.

Platform description

Contents

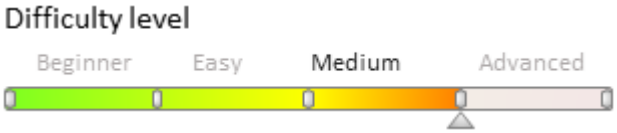
- **System Settings**
- **Working with data structure**
- **User interface**
- **Controls**
- **Dashboard widgets**
- **Scheduler setup**
- **Integration**
- **Self-service Portal**
- **ClientMessageBridge**
- **Sync Engine synchronization mechanism**
- **Data Enrichment and Prediction**
- **Bpm'online lending**
- **Bpm'online marketing**
- **Bpm'online service**
- **DataManager class description and use cases**
- **Feature Toggle. Mechanism of enabling and disabling functions**
- **The MoneyUtilsMixin mixin**
- **The DecimalUtils module**
- **Basic macros in the MS Word printables**
- **Web-to-Case**
- **Separate query pool**
- **Development recommendations for Right-To-Left mode**
- **Client static content in the file system**
- **Record deactivation**
- **Monitoring of private properties overriding. The Terrasoft.PrivateMemberWatcher class**
- **The [Timeline] tab**
- **Server content in the file system**
- **Logging in bpm’online. Log4net**

System Settings

Contents

- **Setting user session timeout**

Setting user session timeout



Introduction

The procedure for user session timeout setup in bpm’online 7.9.1 and up is different from the earlier versions. In the earlier versions, the user session timeout was specified by configuring application pool in the IIS and editing

bpm'online configuration files. Starting with version 7.9.1, session timeouts are set up in the system settings.

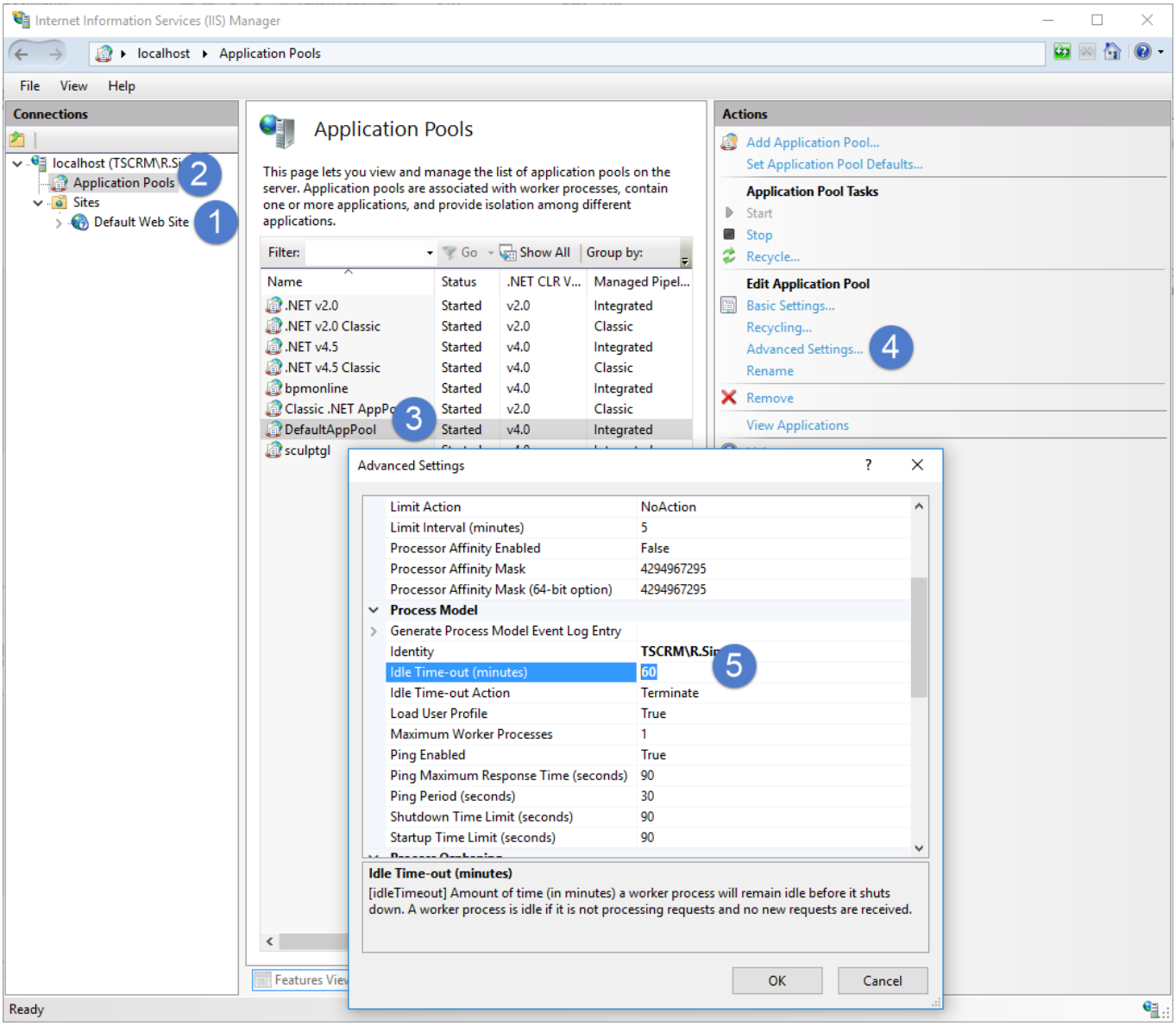
Setting user session timeouts for bpm'online 7.9.0 and earlier.

To set up the user session timeout:

1. Set the idle duration on the IIS in the application pool settings.

To do this, select an application (Fig. 1, 1), in the list of application pools (2), select the pool (3) where the application is published. Open advanced settings (4) and set the needed value for the [Idle Time-out (minutes)] parameter (5).

Fig. 1 Setting the idle duration in the IIS application pool settings.



2. Modifying the Web.config parameters

Modify the session and authentication parameters in the Web.config files used by the loader and the application (the "internal" and "external" Web.config files).

To modify the session parameter, assign the needed timeout value to the *timeout* attribute of the *sessionState*

element, which is a subordinate to the *system.web* element. This value must match the value set in the [Idle Time-out (minutes)] parameter (see item 1).

The authorization parameter is set up in the *forms* element of the *authentication* and *system.web* elements. To modify it, assign a corresponding timeout value in the *timeout* attribute. The authorization parameter value must be less than the value of the session parameter.

An example of the Web.config setup is available below:

```
<system.web>
...
  <authentication mode="Forms">
    <forms loginUrl="~/NuiLogin.aspx" protection="All" timeout="59" name=".ASPXAUTH"
path="/" requireSSL="false" slidingExpiration="true" defaultUrl="ViewPage.aspx?
Id=4e342d5e-bd89-4b79-98e2-22e433122403" cookieless="UseDeviceProfile"
enableCrossAppRedirects="true" />
  </authentication>
...
  <sessionState mode="InProc" timeout="60" cookieName="BPMLOADER" />
</system.web>
```

Setting user session timeouts for bpm'online 7.9.1 and up

Starting with version 7.9.1, session timeouts are set up using the *UserSessionTimeout* system setting. The value of this setting contains user session timeout in minutes.

By default the *UserSessionTimeout* system setting value is 60 minutes. The minimum session timeout is 10 minutes, and the maximum is 720 minutes.

If the system setting is deleted, empty, or otherwise unavailable, the application is still operational. In this case, the timeout value will be taken from the application Web.config session parameter.

Setting the value of the UserSessionTimeout system setting when updating from earlier versions

When updating bpm'online applications version 7.9.0 and earlier, the UserSessionTimeout system setting will be automatically added when the Base package is installed. Default value will be "60". If the timeout value has been modified earlier, you can edit it manually or use the WorkspaceConsole.

The required WorkspaceConsole parameters for updating the value are available in table 1. For more information on working with the WorkspaceConsole, please refer to the "**Working with WorkspaceConsole**" article.

Table 1. WorkspaceConsole settings

Parameter	Description
-operation	Name of the operation for system value actualization. Use the following value to update system settings: *ActualizeUserSessionTimeoutSettingsValue*.
-WebApplicationPath	Path to the deployed application. Example: "C:\bpmonline79"
-workspaceName	Name of the workspace. Default value: *Default*.
-logPath	Paths where the WorkspaceConsole will create log files. Example: "C:\Log".

After running the utility with the specified parameters, the value of session timeout of the *UserSessionTimeout* system setting will be taken from the application Web.config file. If the timeout specified in the Web.config file is not within the range of acceptable values (less than 10 or more than 720), the specified value will still be assigned to the system setting. At the same time, the SysSettings.config file in the loader will be modified to make the value acceptable. For example, "1000" is specified as the timeout value in the Web.config file:

```
<sessionState cookieName="BPMSESSIONID" timeout="1000" mode="Custom"
customProvider="RedisSessionStore">,
```

In this case, after WorkspaceConsole executes a command with the parameters specified in table 1, tha value of the system setting will become "1000" and the following will be specified in the SysSettings.config file:

```
<sysSetting key="UserSessionTimeout" valueType="int" operation="max" value="1000" />
```

Attention!

If the system setting cannot be found in the database, then it is possible that errors occurred during the update. In this case, WorkspaceConsole operation will result in error as well.

NOTE

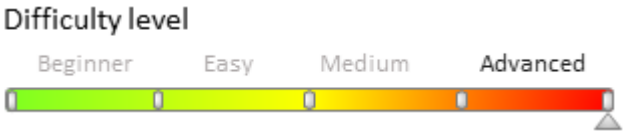
After updating the application and setting the UserSessionTimeout system setting value, no additional session timeout setup on the application pool level is required. There is no need to specify authentication timeout as well.

Working with data structure

Contents

- **Configuration localizable resources**
- **Localizable resource structure and use**
- **Localization tables**
- **Bound data structure**

Configuration localizable resources



Introduction

Configuration resources are localizable strings and images used by the application to display information to the user.

The application resources are places in the packages and bound to the base schema in the schema hierarchy. When resources of a certain schema are queried, all resources are gathered throughout the hierarchy, after which a flat list of the gathered resources is generated.

Resources displayed as a hierarchy

There are two modes of displaying schema resources: design mode (Design-time) and application runtime mode (Run-time).

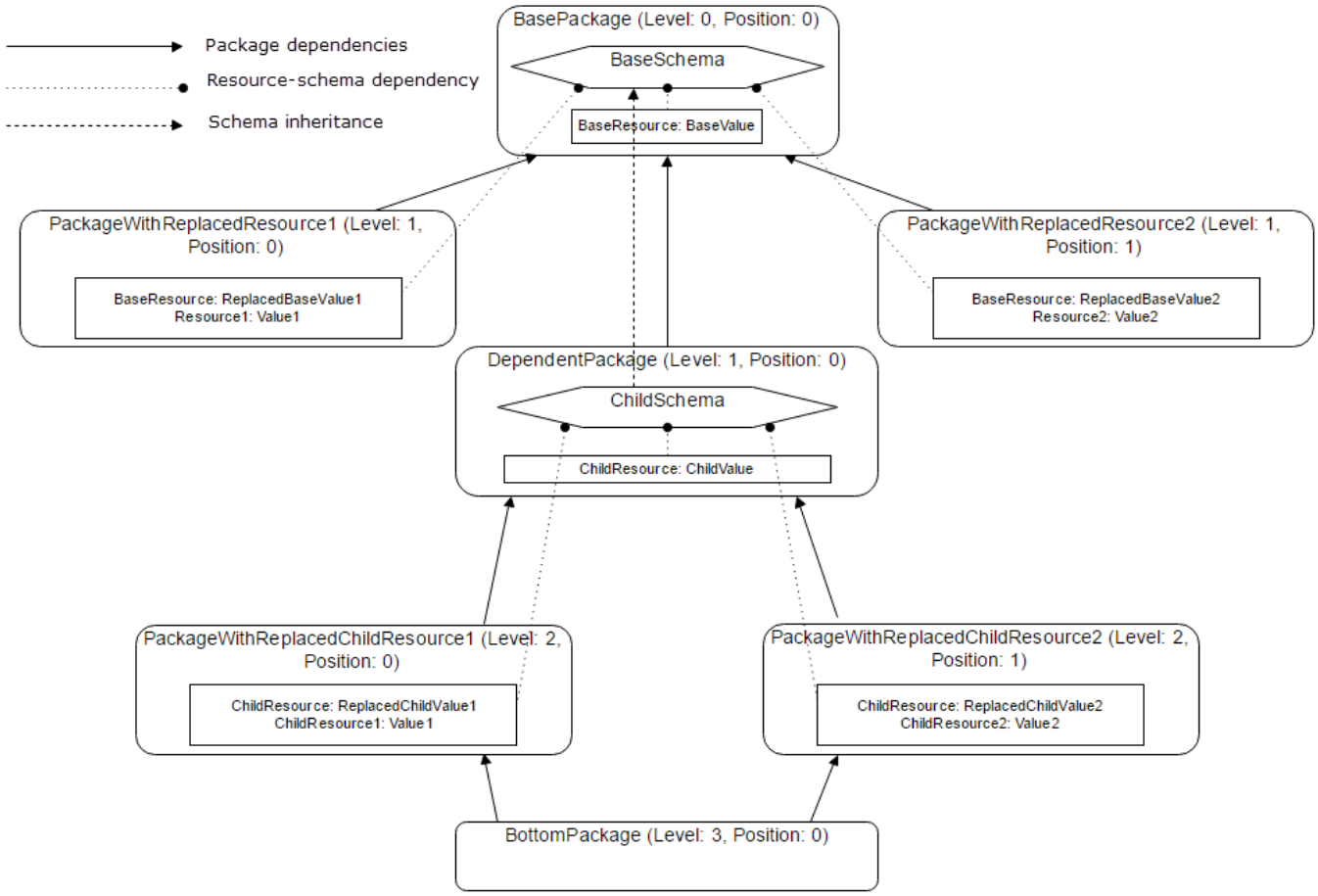
Design-time mode

This mode is used to display resources in designers and wizards. Schema resources are displayed only up to the package that contains the schema. Package resources that are not part of the hierarchy are not taken into account. For example, the *ChildSchema* schema (Fig. 1) will have the following resources:

- *BaseResource: BaseValue*;
- *ChildResource: ChildValue*.

The resources of the *PackageWithReplacedResource1* and *PackageWithReplacedResource2* packages that are not a part of the hierarchy are not taken into account. The resources in the *PackageWithReplacedChildResource1* and *PackageWithReplacedChildResource2* packages that are lower that the requested schema in the hierarchy are not taken into account as well.

Fig. 1 An example of the package hierarchy



If a schema is requested along with the package from which all resources must be obtained, then the resulting set of resources will be generated up to the level of the requested package. For example, *ChildSchema* schema resources up to the *BottomPackage* level will be as follows:

- *BaseResource: BaseValue;*
- *ChildResource: ReplacedChildValue2;*
- *ChildResource1: Value1;*
- *ChildResource2: Value2.*

The value of the *ChildResource* here has been changed to *ReplacedChildValue2*. This occurred because it has been replaced in the packages one level lower (Level 2). Packages with higher position value take precedence. If the position values are the same, the first package (if sorted by name) will take precedence.

Run-time mode

This mode displays resources in all system sections except for designers. The mechanism for obtaining resources is similar to the mechanism used in the Design-time mode. The main difference is that when a schema is requested, the resulting list will contain resources from packages that are not directly a part of the hierarchy. For example, if the *ChildSchema* resources are requested, the result will be as follows:

- *BaseResource: ReplacedBaseValue2;*
- *Resource1: Value1;*
- *Resource2: Value2;*
- *ChildResource: ReplacedChildValue2;*
- *ChildResource1: Value1;*
- *ChildResource2: Value2.*

Default resource view

If there are no resource values that can be displayed for users for a non-default culture, then the values are

“reverted” to the default culture values.

This mechanism is implemented in the *Terrasoft.Common.LocalizableString* (displays a localized string) and *Terrasoft.Common.LocalizableImage* (displays a localized image) classes. These classes contain the following properties and methods for obtaining localized values:

- *Value* – a property that returns the value of a localized object for the current culture or the default culture, if the former is not found.
- *HasValue* – a property that returns the flag indicating that the value of a localized object exists for the current culture or the default culture, if the former is not found.
- *GetCultureValue()* – a method that returns the value of a localized object for the current culture or the default culture, if the former is not found.
- *HasCultureValue()* – a method that returns he flag indicating that the value of a localized object exists for the current culture.

Storing resources

Resources needed for the application operation are stored in the database. Resources can be stored in a version control system for installing on a new application or transferring between applications.

Storing resources in the database

The resources for each string or image are stored in the *SysLocalizableValue* database table in the “key-value” format. The table structure is available in table 1. Each record in the *SysLocalizableValue* table is bound to a package and an Id of the base *Id* schema. The schema itself can be located in a different package.

Table 1. *SysLocalizableValue* table structure

Column name	Description
Id	Table record Id
CreatedOn	Table record creation date
CreatedById	Link to the *Contact* who created the record.
ModifiedOn	Table record last modification date
ModifiedById	Link to the last *Contact* who edited the record.
SysPackageId	Link to the package (*SysPackage*).
SysSchemaId	Link to the base schema (*SysSchema*). This column is filled in for configuration resources only.
ResourceManager	Name of the resource manager. This column is filled in for core resources only.
SysCultureId	Link to the culture (*SysCulture*).
ResourceType	The type of resource.
IsChanged	Indicates whether the resource has been modified by the user.
Key	Resource key.
Value	Value of the string resource.
ImageData	Value of the image resource.

Default resource saving

If a resource is created by a user whose culture is not default, a record matching the user’s culture will be created for the resource. Newly created resources are automatically duplicated in the default culture. As a result, a similar resource record will be created with a link to the default culture. The new value of the resource will be displayed in all cultures if they don’t have a native value for that resource (please see the “Default resource view” section of this article).

Storing resources in the version control system and file system

The resource structure in the version control system and file system is covered in the "Resource storage structure” of the "**Localizable resource structure and use**” article.

Localizable resource structure and use

Difficulty level

Introduction

Starting with version 7.8.3, the storage location of localized package resources has changed. In previous versions, the resources were stored in the *SysSchemaResource* table as BLOB data. Now, the localized resources are stored in the *SysLocalizableValues* table in text form.

For each set of schema resources in the package-schema-culture bundle, the checksum is stored in the *SysPackageResourceChecksum* table, which allows you to quickly determine if there are any changes to the package resources when updating the package. The checksum allowed for resources to be separated from schemas, enabling users to create translation packages.

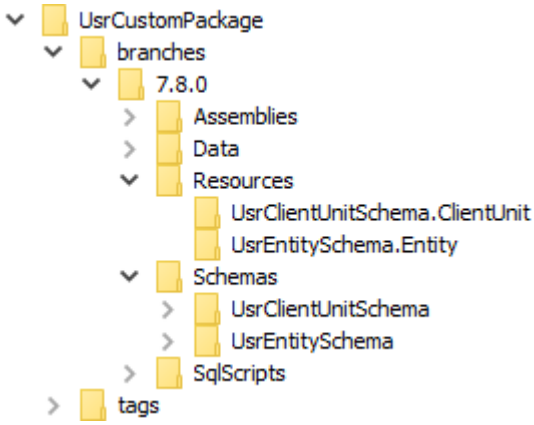
> ⚠ ATTENTION
>
> The approach to working with language cultures has changed in bpm’online 7.11.1. Now, the application uses only the cultures that have [Active] checkbox selected. This improves performance of different types of tasks, such as logging in, opening a record page, etc. However, all installed language cultures will be used when working with section and detail wizards, process and case designers and the [Translation] section.

Storing resources

Resource storage structure

The resources have been moved from schemas to a package, enabling users to create translation packages. To store schema resources with the same name, but with different managers (e.g. *Entity* and *ClientUnitSchema*), the names of the schema managers with the prefix "SchemaManager" clipping were added to the resource package names.

Fig. 1. Storing resources in a package



A package can contain resources for a schema that is defined in another package. In addition, the package can contain resources without containing schemas ("translation package").

The SysLocalizableValue table

The resources are stored in the *SysLocalizableValue* table for every localizable string or image. Each record is bound to the package and the base schema identifier. The main fields of the *SysLocalizableValue* table are listed in Table 1.

Table 1. The main fields of the SysLocalizableValue table

Column name	Description
Id	Record identifier.
ImageData	Graphic resource value.
IsChanged	A checkbox for specifying if the resource has been changed.
Key	Resource key.
ResourceManager	Resource manager name. Populated only for core resources.
ResourceType	Resource type.
SysCultureId	Culture identifier.
SysPackageId	Package identifier.
SysSchemaId	Base schema identifier. Populated only for configuration resources.
Value	String resource value.

Schema import and export

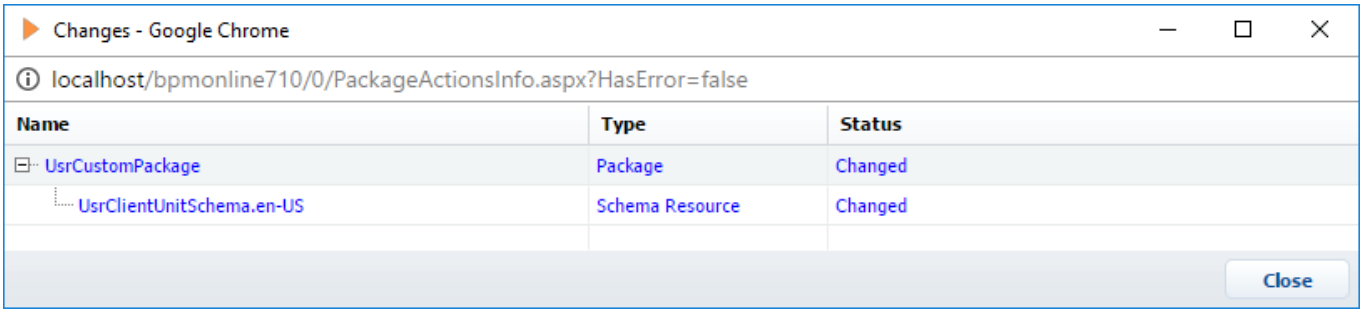
The format for storing resources in exported schemas has also changed. Now the resources in the exported schemas are stored in XML format.

Working with localizable resources

Updating the package from the repository

By using the new resource storage mechanism, all changes to localized resources are displayed when the package is updated (Fig. 2).

Fig. 2. Displaying resources when updating a package



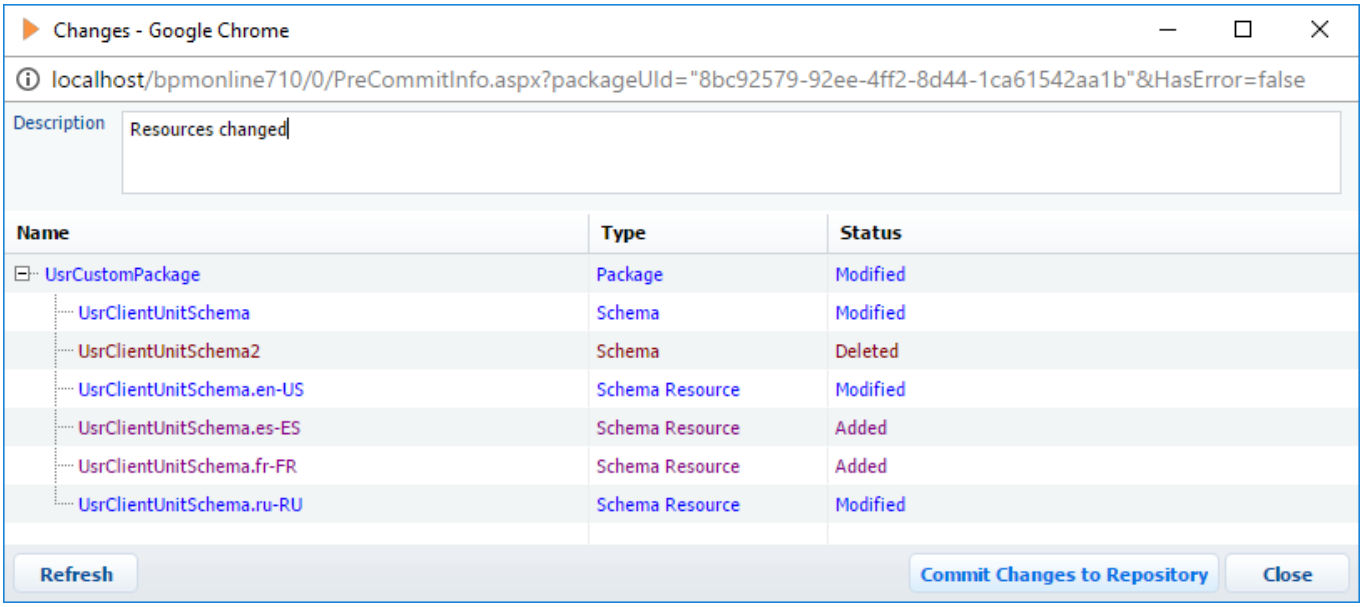
Possible resource states:

- [Modified] – the resource was changed.
- [Added] – a new resource has been added.
- [Deleted] – a resource has been deleted.
- [Conflict] – a resource was modified and fixed in SVN when another developer was working on it.

Committing a package to storage

When the package is committed, all changes to the localized resources are also displayed in the repository (Fig. 3).

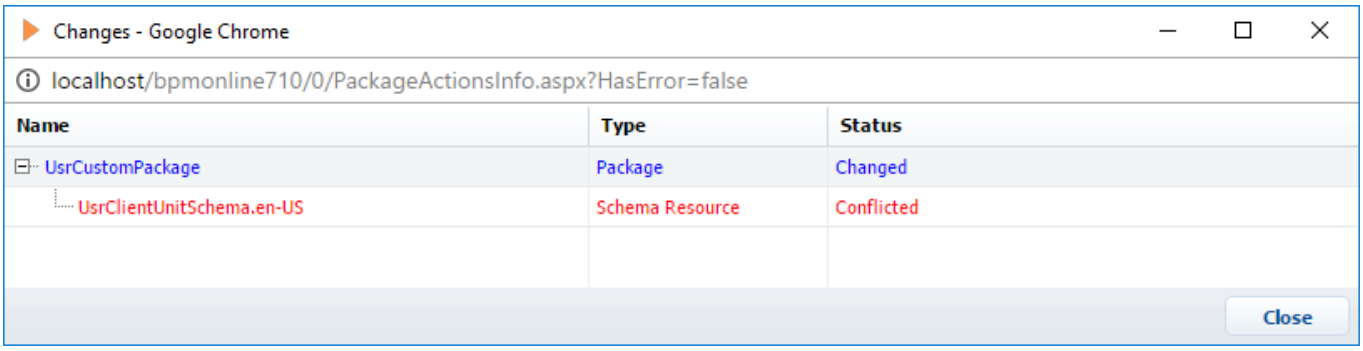
Fig. 3. Displaying resources when a package is committed



Conflicts when a package is committed and updated

It is not currently possible to block resources in the application. If the developer modifies the schema resources, and the package has already been modified and the same resources have been modified in it, they will see a list of those resources with the [Conflict] state. This means that the version and contents of the resources modified by the developer do not match the version and contents committed in SVN. When the developer commits again, their modifications will block the modifications committed in SVN. Such conflicts must be resolved manually in SVN clients (e.g. Tortoise).

Fig. 4. Displaying conflicts when updating a package



Editing resources in the file system.

Editing resources directly is available starting with version 7.8.3. To do this, you need to upload them to the file system (e.g. with Tortoise), and then make changes and commit to SVN.

> ⚠ ATTENTION
>
> For each resource value in the *SysLocalizableValue* table, there is only one record with the corresponding references to *SysPackageId*, *SysSchemaId*, *SysCultureId*, and a specific *Key* value. Therefore, when you commit a resource with the [Conflict] state, the table will write the last value.

Updating resources with a direct SQL query to the database

If you change the *SysLocalizableValue* table value with an SQL query, you must also change the value of the *IsChanged* column in the *SysPackageResourceChecksum* table for the corresponding schema. Otherwise, when the package is updated, the application will not detect a conflict.

You can not add data to the *SysLocalizableValue* table with a direct SQL query, because there is no information about the added resources in the corresponding schema metadata.

Localization tables

Difficulty level

Beginner Easy Medium Advanced

Localization tables are created for objects with at least one localizable column. These tables store localizable data for all languages (cultures) except for the default one.

> ⚠ ATTENTION
>
> The approach to working with language cultures has changed in bpm’online 7.11.1. Now, the application uses only the cultures that have [Active] checkbox selected. This improves performance of different types of tasks, such as logging in, opening a record page, etc. However, all installed language cultures will be used when working with section and detail wizards, process and case designers and the [Translation] section.

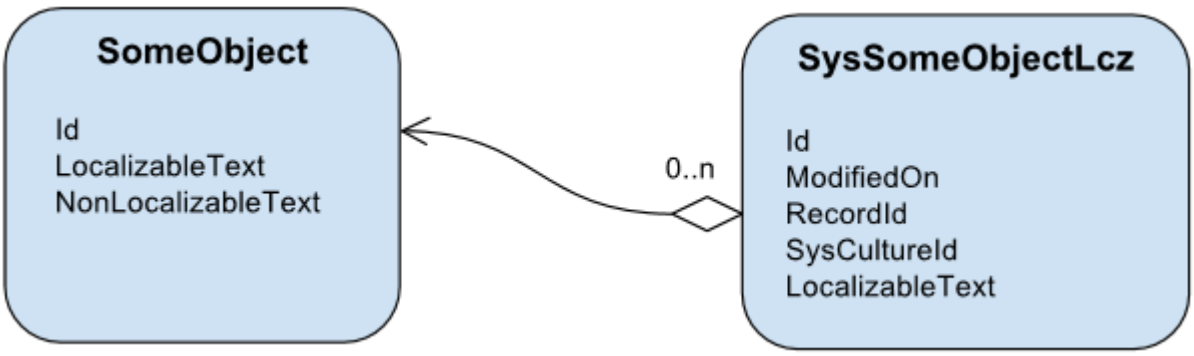
The default localization table name is Sys[schema_name]Lcz, where [schema_name] is the object schema with the localizable columns. General localization table structure:

Table 1. – General localization table structure

Column name	Description
Id	Record identifier.
ModifiedOn	Modification date.
RecordId	A link to the localized record in the main object table.
SysCultureId	Culture link.
LczColumn1	Columns corresponding to the object's localizable columns.
LczColumn2	
...	
LczColumnN	

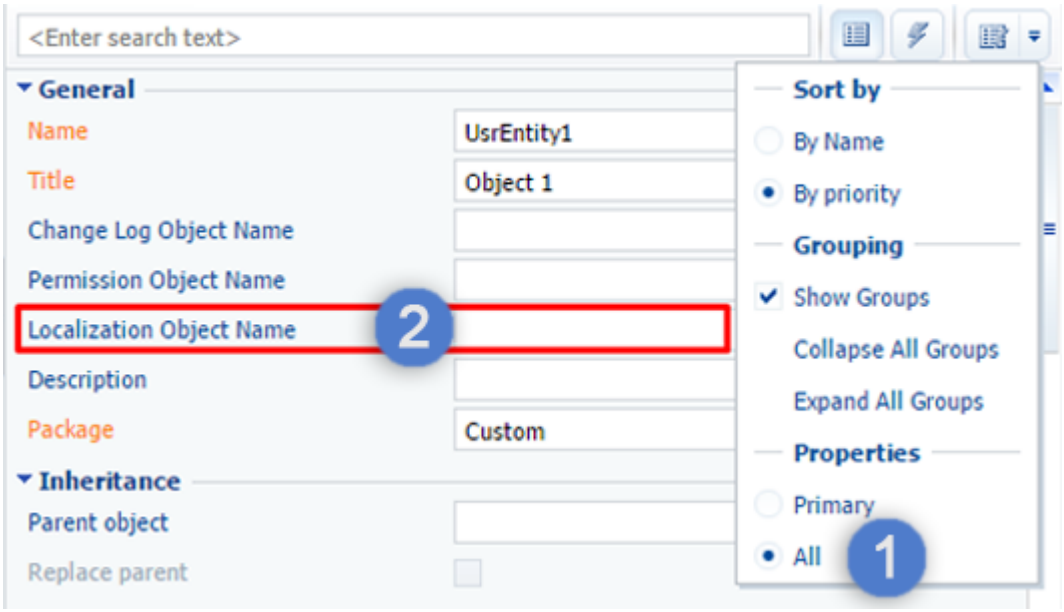
The localization table structure of the *Random* object with the *LocalizableText* column:

Fig. 1. The link between the main table and the localization table.

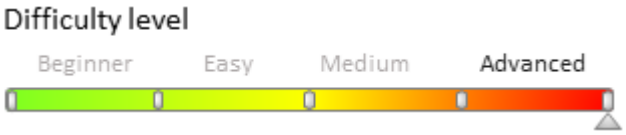


Use the object designer to specify / change the name of the table. Go to the advanced schema object properties and specify the name of the localization object.

Fig. 2. Localization table name in the object designer



Bound data structure



Introduction

Bpm’online comes with full multilanguage support since version 7.8. As a result, the storage structure of **resources** and bound data has been reworked.

Data binding specifics (compared to version 7.8):

- A new *SysPackageDataLcz* table now contains localized bindings data.
- A new mechanism for creating and installing bindings.
- A new SVN data storing structure.

The SysPackageDataLcz localization table

An additional table is used for storing localized bound data:

Table 1. SysPackageDataLcz table columns

Column name	Description
Id	Unique identifier.
SysPackageSchemaDataId	A reference to the unique binding identifier in the *SysPackageSchemaData* table.
SysCultureId	Unique culture identifier reference.
Data	Localized data.

ATTENTION

The binding mechanism interface is identical to that of the previous versions.

Creating a binding

If a schema does not contain localized columns, the bound data for this schema is still stored in the *SysPackageSchemaData* table. Data binding for a schema with localized columns:

- The bound data is still stored in the *SysPackageSchemaData* table.
- The *SysPackageDataLcz* table contains localized bindings data.
- Every record in *SysPackageDataLcz* corresponds with a record in *SysPackageSchemaData*, with a reference to the unique *SysCultureId* identifier. For example, if two cultures (English and Spanish) are installed in the system, each entry in the *SysPackageSchemaData* table will correspond to entries in the *SysPackageDataLcz* table, with a reference to the corresponding record identifier in the *SysPackageSchemaData* table, and the culture identifier in the *SysCulture* table.

Installing bound data

Installing data for a schema without localized columns is done in the corresponding schema table. If the data includes any localized values (i.e. there are corresponding records in the *SysPackageDataLcz* table), the installation occurs not only in the corresponding schema table, but also in its localized *Sys[schema_name]Lcz* table.

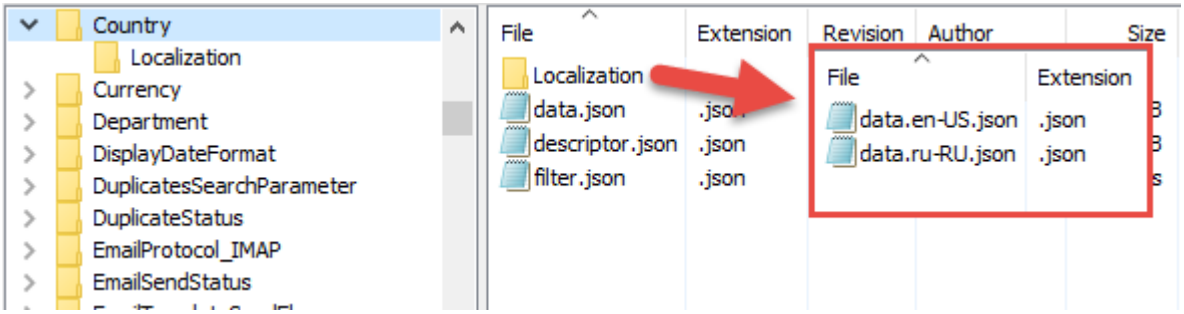
For example, the bound data for the *ContactType* schema is installed. Non-localized data is installed in the *ContactType* table, and the localized data is installed in the *ContactType* table (default culture values), and the *SysContactTypeLcz* table (the values of all other cultures included in the binding and in the system).

> ATTENTION
>
> If you are working with a system schema (the name begins with the “Sys” prefix), then the “Sys” prefix is not re-added to the localization table. For example, if the schema name is *SysTest1*, the localized data table name will be *SysTest1Lcz*, and not *SysSysTest1Lcz*.

Storing data in SVN

The structure of storing the bound schema data in SVN for bpm'online version 7.8 and higher:

Fig. 1. SVN data storing structure



Non-localized data is stored in the *data.json* file. All localized data is located in the corresponding files in the *Localization* subdirectory. For example, for the *Country* schema of the *Base* package, the data is localized for only two languages and stored in the corresponding files - *data.en-US.json* and *data.es-ES.json* (Fig. 1).

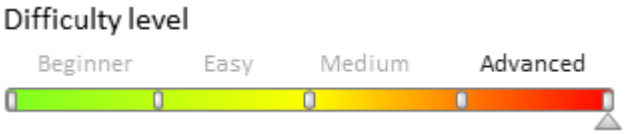
User interface

Contents

- **AMD concept Modules**
- **Modular development principles in bpm'online**
- **Client Modules**
- **Client view model schemas**
- **Sandbox. Module message exchange**

- **Sandbox. Bidirectional messages**
- **Sandbox. Loading and unloading modules**
- **New bindTo format at setting connection between view and viewModel**

AMD concept Modules



Introduction

Starting from version 7.0, the client part of the bpm'online application has a modular structure: it is designed as a set of functional blocks, implemented in separate modules. While the application is running, the modules and their dependencies are loaded in accordance with the Asynchronous Module Definition (AMD) approach.

The AMD approach declares the mechanism for defining and asynchronous loading of modules and their dependencies, which allows you to load only the data needed to work with the system at the moment. The AMD concept is supported by various JavaScript frameworks. In bpm'online, the RequireJS loader is used to work with modules.

Modules

A *module* is a code fragment encapsulated in a separate block that can be downloaded and executed independently.

In JavaScript, modules are created in accordance with the "Module" programming pattern. A classic implementation of this pattern is using anonymous functions that return specific values (object, function, etc.) associated with the module. The module value is exported to the global object. Example:

```
// Immediately-invoked functional expression (IIFE). Anonymous function,
// which initializes the "myGlobalModule" property of the global object with a
function,
// that returns module value. Thus, the module actually loads,
// which can later be accessed through the "myGlobalModule" global property.
(function () {
    // Access to a module on which the current module depends.
    // This module already should be loaded to the
    // "SomeModuleDependency" global variable at the moment of access.
    // "this" context in this case is a global object.
    var moduleDependency = this.SomeModuleDependency;
    // The declaration in the property of the global function object that returns the
module value.
        this.myGlobalModule = function () { return someModuleValue; };
}());
```

When interpreter finds a functional expression like this, it immediately resolves it. As a result, a function that will return the module value will be placed in the *myGlobalModule* property of the global object.

The main disadvantages of this approach are the complexity of declaration and use of the dependency modules for the modules of such type. In particular, the disadvantages are:

1. All module dependencies must already be loaded at the moment of anonymous function execution.
2. The dependency modules are loaded via the *<script><script/>* HTML element at the page header. Global variable names are then used to access the modules. At the same time, the developer must clearly understand and implement the order in which all dependency modules are loaded.
3. As a result, the modules are loaded before the page is rendered, therefore the modules cannot access the page controls to implement custom logic.

This means that the modules cannot be loaded dynamically; no additional logic can be applied at page loading, etc. In large projects like bpm'online, the complexity of managing a large number of modules with many dependencies

that can overlap each other is a problem.

The "RequireJS" loader

RequireJS is an AMD-based module declaring and loading mechanism that helps avoid the disadvantages of working with large numbers of modules. Basic principles of the RequireJS loader operation:

1. Modules are declared in a special *define()* function, which registers a factory function to instantiate a module. At the same time, it does not load the module immediately when function is called.
2. The module dependencies are passed as a string array and not through the properties of the global object.
3. The loader executes the loading of all dependency modules passed as arguments to *define()*. The modules are loaded asynchronously, and the loader determines their loading order arbitrarily.
4. After the loader completes loading of all specified module dependencies, it will call the factory function that will return the module value. The downloaded dependency modules will be passed to the factory function as arguments.

Module declaration. The "define()" function

For the loader to work with an asynchronous module, this module must be declared in the source code by the *define()* function in the following way:

```
define(
        ModuleName,
        [dependencies],
        function (dependencies) {
        }
);
```

The parameters of the *define()* function are listed in Table 1.

Table 1. - The parameters of the *define()* function

Argument	Value
ModuleName	Module name string. Optional parameter. If the parameter is not specified, the loader will assign the module name, based on its location in the application script tree. However, to access the module from other parts of the application (including the cases when this module must be asynchronously loaded as a dependency of another module), the parameter must be specified.
dependencies	An array of module names that this module depends on. Optional parameter. RequireJS loads all dependencies passed in the array. Note that the order of dependencies in the *dependencies* array enumeration must correspond to the order of parameters in the enumeration passed to the factory function. The factory function will be called only after all dependencies listed in the *dependencies* parameter have been loaded. The loading of dependency modules is asynchronous.
function(dependencies)	Anonymous factory function that instantiates the module. Required parameter. The objects that are associated by the loader with the dependency modules listed in the *dependencies* argument are passed to the function as arguments. Access to the properties and methods of the dependency modules within the created module is carried out through these arguments. The order of modules in the *dependencies* enumeration must correspond to the order of the factory function arguments. The factory function will be called only after all dependency modules of the current module (listed in the *dependencies* parameter) have been loaded. The factory must return a value that the loader will associate as the exported value of created module. The return value can be: • An *object, which is the module for the system*. After this module is initially download by the client, it is saved in the browser cache. If the module

declaration has been modified after it was downloaded to the client (for example, during the configuration logic implementation), then the cache needs to be cleared and the module must be loaded again. An example of module declaration that returns the declared module as an object is provided below.

- *The module constructor function*. The context object in which the module will be created is passed as an argument to the constructor. Loading this module will result in creating of the module instance (instantiated module) on the client. Reloading of this module to the client with the *require()* function will create another instance of the module. These two instances of the same module will be treated by the system as two different independent modules. An example of declaring an instantiated module is the *CardModule* module from the *NUI* package.

An example of using the *define()* function to declare a *SumModule*, which adds two numbers.

```
// The "SumModule" module has no dependencies.
// So, an empty array is passed as the second argument, and
// parameters are not passed to the anonymous factory function.

define("SumModule", [], function () {
    // The body of the anonymous function contains internal functionality
implementation of the module.
    var calculate = function (a, b) { return a + b; };
    // The value returned by the function is an object, which is the module for the
system.
    return {
        // Object description. In this case, the module is an object with the "summ"
property.
        // The value of this property is a function with two arguments, returning
the sum of these arguments.

  summ: calculate
    };
});
```

The factory function returns the object as the module value, which the module will be for the system.

Modular development principles in bpm'online

Difficulty level

Types of bpm’online modules

All client functions in bpm’online can be broken down into the following groups:

- Base libraries
- Core
- Sandbox
- Client modules

Base libraries

Base libraries are third-party JavaScript libraries used in the application. The RequireJS library is used as the module loader. The ExtJS framework functions are used in the configuration logic for working with interface controls. JQuery, Backbone and other frameworks are used as well. All third-party JavaScript libraries are placed in the *Terrasoft.WebApp\Resources\ui* folder of the application.

Core

The main purpose of the bpm’online client core is to provide a unified interface for interaction of all other client parts of the system. The core provides API for accessing base client libraries, defines the sandbox contents for modules, provides access to system enumerations and constants, implements client mechanism for working with data, etc. The core does not work directly with the system modules. It is only aware of the primary application module (*ViewModule*), which loads all remaining modules.

To access the core functions used in the custom client logic, a module must import the *terrasoft* module as a dependency.

Sandbox

A module is an isolated programming unit. It is not aware of other system modules except for the names of the modules from which it depends. A special object called the *sandbox* is designed for interaction between the modules.

The sandbox provides the two key mechanisms for interaction between the modules in the system:

1. A mechanism for message exchange between the modules. Modules can communicate with each other only through messages. If module needs to inform other modules that its status has changed, it publishes a message using the *sandbox.publish()* method of the sandbox. If a module needs to receive messages about status changes of other modules, it must subscribe to those messages. The subscription is done through calling the *sandbox.subscribe()* sandbox method.
2. Loading modules “on-demand” into the specified area of the application (for visual modules). In the process of implementing custom business logic, you many need to load dynamically the modules that have not been declared as dependencies. These modules can be loaded in the process of the module declaration, in the *define()* function. The *sandbox.load()* sandbox method is used for this.

To enable interaction with other modules, a module must import the *sandbox* module as dependency.

Client modules

Client modules are separate functional blocks that are loaded and run on-demand, according to the AMD technology. All custom functions are implemented in client modules. Despite several functional differences, all bpm'online client modules have similar structure that matches the module description format in AMD. For more information about client modules, please see the “**Client Modules**” article.

The “ext-base”, “terrasoft” and “sandbox” modules

Bpm’online contains modules that are used in most client modules of a configuration. These are the *ext-base* module of the ExtJs framework functions, the *terrasoft* module of the *Terrasoft* objects and name spaces, and the *sandbox* module that implements the mechanism for message exchange between modules. These modules can be accessed in the following way:

```
// Module definition and getting dependency module links.
define("ExampleModule", ["ext-base", "terrasoft", "sandbox"],
    // Ext — link to the object that grants access to the ExtJs features.
    // Terrasoft — link to the object that grants access to the system variables,
core variables, etc.
    // sanbox — used for message exchange between modules.
    function (Ext, Terrasoft, sandbox) {
});
```

Specifying base modules in the ["ext-base", "terrasoft", "sandbox"] dependencies is not required. After creating module’s class object, the *Ext*, *Terrasoft* and *sanbox* modules will be available as object properties: *this.Ext*, *this.Terrasoft*, *this.sanbox*.

Declaring module class The *Ext.define()* method

One of the more important ExtJs javascript framework functions in bpm’online is class declaration. The *define()* method of the global *Ext* object is used for this. An example of declaring a class with this method:

```
// Terrasoft.configuration.ExampleClass — class name with
```

```
// name space compliance.
Ext.define("Terrasoft.configuration.ExampleClass", {
    // Shortened class name.
    alternateClassName: "Terrasoft.ExampleClass",
    // Name of the class from which inheritance is made.
    extend: "Ext.String",
    // Block for declaring static properties and methods.
    static: {
        // Example of a static property.
        myStaticProperty: true,

        // Example of a static method.
        getMyStaticProperty: function () {
            // Example of access to a static property.
            return Terrasoft.ExampleClass.myStaticProperty;
        }
    },
    // Example of a dynamic property.
    myProperty: 12,
    // Example of a class dynamic method.
    getMyProperty: function () {
        return this.myProperty;
    }
});
```

Examples of various options for creating class instances:

```
// Creating a class instance by full name.
var exampleObject = Ext.create("Terrasoft.configuration.ExampleClass");
// Creating a class instance by a shortened name (alias).
var exampleObject = Ext.create("Terrasoft.ExampleClass");
// Creating a class instance with the specified properties.
var exampleObject = Ext.create("Terrasoft.ExampleClass", {
    // Overriding object property from 12 to 20.
    myProperty: 20,
    // Defining a new method for the current class instance.
    myNewMethod: function () {
        return this.getMyProperty() * 2;
    }
});
```

Inheriting a module class

In a simple implementation of a module, its contents is either a simple object with a set of methods and properties, or a constructor function that the module must return to a function that is called after its loading.

```
define("ModuleExample", [], function () {
    // Example of a module that returns a simple object.
    return {
        init: function () {
            // The method will be called on module initialization,
            // but the module contents will not get into the DOM.
        }
    }
});
define("ModuleExample", [], function () {
    // Example of a module that returns a constructor function.
    return function () {
        this.init = function () {
            // The method will be called on module initialization,
            // but the module contents will not get into the DOM.
        }
```

```
    }
});
```

Such simple module cannot add its view to the Document Object Model (DOM), unless you explicitly implement the *render()* method, which would return a view instance and insert it to the DOM. The logic for calling the *render()* method in a module object is covered on the application core level. The *destroy()* method is not implemented in such module as well. If the module is visual, i.e., it contains the *render()* method, then it will be impossible to unload the view from the DOM, unless the unloading logic is implemented in the *destroy()* method.

In most cases, the module class should be inherited from *Terrasoft.configuration.BaseModule* or *Terrasoft.configuration.BaseSchemaModule*, where the following methods are already implemented:

- *Init()* – a method for module initialization. Initializes the properties of class object and subscribes to messages.
- *render()* – a method for rendering the module view in the DOM. Returns a view. Accepts a single *renderTo* argument, which is the element where the module object view will be inserted.
- *Destroy()* – a method that deletes a module view, view model, unsubscribes from messages and destroys the module class object.

Below is an example of a simple module class inherited from "Terrasoft.BaseModule". This module adds a button to the DOM. Clicking the button will display a text message and then the button is deleted from the DOM.

```
define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        // Short class name.
        alternateClassName: "Terrasoft.ModuleExample",
        // The class from which the inheritance is done.
        extend: "Terrasoft.BaseModule",
        // Reguired property. If it is not defined, an error will be generated on the
        // "Terrasoft.core.BaseObject" level, since the class is inherited from
"Terrasoft.BaseModule".
        Ext: null,
        // Reguired property. If it is not defined, an error will be generated on the
        // "Terrasoft.core.BaseObject"level, since the class is inherited from
"Terrasoft.BaseModule".
        sandbox: null,
        // Reguired property. If it is not defined, an error will be generated on the
        // "Terrasoft.core.BaseObject"level, since the class is inherited from
"Terrasoft.BaseModule".
        Terrasoft: null,
        // View model.
        viewModel: null,
        // View. A button is used as an example.
        view: null,
        // If the init() method is not implemented in this class,
        // then, when an instance of the current class is created,
        // the init() method of the parent class Terrasoft.BaseModule will be called.
        init: function () {
            // Executes the logic of the init() method of the parent class.
            this.callParent(arguments);
            this.initViewModel();
        },
        // Initializes a view model.
        initViewModel: function () {
            // Saving module class context
            // for accessing it from the view model.
            var self = this;
            // Creating a view model.
            this.viewModel = Ext.create("Terrasoft.BaseViewModel", {
                values: {
                    // Button caption.
                    captionBtn: "Click Me"
                },
```

```
                methods: {
                    // Button click handler.
                    onClickBtn: function () {
                        var captionBtn = this.get("captionBtn");
                        alert(captionBtn + " button was pressed");
                        // Calls a method for unloading the view and view model,
                        // which results in deleting the button from the DOM.
                        self.destroy();
                    }
                }
            });
        },
        // Creates a view (button),
        // binds it to the view model and inserts in the DOM.
        render: function (renderTo) {
            // A button is created as a view.
            this.view = this.Ext.create("Terrasoft.Button", {
                // Container where the button will be placed.
                renderTo: renderTo,
                // The id HTML attribute.
                id: "example-btn",
                // Class name.
                className: "Terrasoft.Button",
                // Button caption.
                caption: {
                    // Binds the button caption
                    // with the captionBtn property of the view model.
                    bindTo: "captionBtn"
                },
                // Handler method for the button click event.
                click: {
                    // Binds the button click handler
                    // to the onClickBtn() method of the view model.
                    bindTo: "onClickBtn"
                },
                // Button style. Available styles are defined in the enumeration.
                // Terrasoft.controls.ButtonEnums.style.
                style: this.Terrasoft.controls.ButtonEnums.style.GREEN
            });
            // Binds the view and the view model.
            this.view.bind(this.viewModel);
            // Gets the view that will be inserted in the DOM.
            return this.view;
        },


        // Deletes unused objects.
        destroy: function () {
            // Destroys the view, which results in deleting the button from the DOM.
            this.view.destroy();
            // Deletes the unused view model.
            this.viewModel.destroy();
        }
    });
    // Gets module object.
    return Terrasoft.ModuleExample;
});
```

⚠ Notes

Adding the button using the ViewModel schema is described in the "**How to add a button to a section**", "**How to add a button to an edit page in the new record add mode**" и "**How to add the button on**

the edit page in the combined mode" articles.

Synchronous and asynchronous module initialization

There are two ways to initialize a module class instance: synchronously and asynchronously.

Synchronous initialization

A module is initialized synchronously if the *isAsync: true* property (of the configuration object that is passed as a parameter of the *loadModule()* method) is not specified at its loading. For example, if the following is executed:

```
this.sandbox.loadModule([moduleName])
```

...Then the module class methods will be loaded synchronously. The *init()* method will be called first, then the *render()* method will be immediately executed.

Below is an example of a synchronously initialized module.

```
define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        alternateClassName: "Terrasoft.ModuleExample",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
        init: function () {
            // This is executed first upon module initialization.
        },
        render: function (renderTo) {
            // This is executed on the module initialization, right after the init
method.
        }
    });
});
```

Asynchronous initialization

A module is initialized asynchronously if the *isAsync: true* property (of the configuration object that is passed as a parameter of the *loadModule()* method) is specified at its loading. For example, if the following is executed:

```
this.sandbox.loadModule([moduleName], { isAsync: true })
```

In this case, a single parameter will be passed to the *init()* method: a callback function with the current module context. When calling the callback function, the render() method of the loaded module is called. The view will be added to the DOM only after the render() method is executed.

Below is an example of an asynchronously initialized module.

```
define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        alternateClassName: "Terrasoft.ModuleExample",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
        // This is executed first upon module initialization.
        init: function (callback) {
            setTimeout(callback, 2000);
        },
        render: function (renderTo) {
            // The method is executed after a 2 second delay.
            // The delay is specified in the setTimeout() function argument, in the
init() method.
        }
    });
```

```
});
});
```

Chain of modules

Sometimes a model must be shown in the view of other model. For example, the *SelectData* page for selecting a lookup value must be displayed to set a value in a certain field on the current page. In this case, the current page module must not be unloaded, and the lookup selection page view must be displayed in its container. To implement this, use module chains.

To start building a chain, add the *keepAlive* property in the configuration object of the loaded module. For example, a lookup selection module *selectDataModule* must be called from the current page module *CardModule*. To do this, the following code must be executed:

```
sandbox.loadModule("selectDataModule", {
    // Id of the loaded module view.
    id: "selectDataModule_id",
    // The view that will be added to the current page container.
    renderTo: "cardModuleContainer",
    // Specifies that the current module must not be unloaded.
    keepAlive: true
});
```

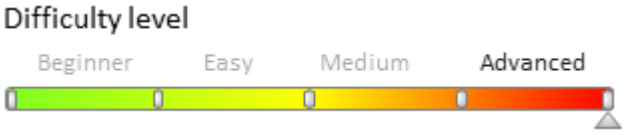
After the code is executed, a module chain will be created, consisting of the current page module and the lookup selection page module. Clicking the [Add new record] button from the current *selectData* page module will open a new page and add another module to the chain. This way you can add any number of module instances to a chain. Active module (the one that is currently displayed on the page) is always the last element in the chain. If an intermediate element in the chain is set as active, then all elements that are located after it will be destroyed. Use the *loadModule* function to activate a chain element and pass the module Id as its parameter:

```
sandbox.loadModule("someModule", {
    id: "someModuleId"
});
```

The core will destroy all chain elements after the specified one and will execute standard module loading logic (call the *init()* and *render()* methods). The *render()* method will be passed to the container where the previous active module was placed. All modules in the chain can work (receive and send messages, save data, etc.) as before.

If the *keepAlive* is not added to the configuration object (or added with the *keepAlive: false* value) on loading of the *loadModule()* method, then the module chain will be destroyed.

Client Modules



Introduction

Client Modules are separate functional blocks, downloaded and run on demand in accordance with the AMD technology. System functions are implemented via client modules. All client modules in bpm’online share description structures that correspond with AMD module description format.

Client module types

The following client module types are available in bpm’online:

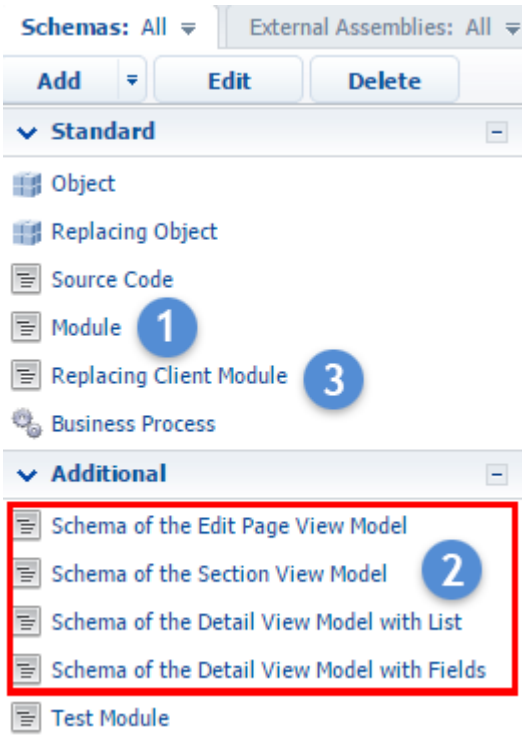
- non-visual modules (module schema)
- visual modules (view model schema)
- expanding modules and replacing client modules

Non-visual modules (module schema)

Non-visual modules represent system functionality that is not associated with data binding or data display in the UI. Examples of non-visual modules in the system are business rule modules (*BuisnessRuleModule*) and utility modules that implement service functions.

Go to the [Configuration] section, click [Add] and select [Module] to create a non-visual module (Fig. 1, 1).

(Fig. 1, 1). Creating non-visual modules



Visual module (view model schema);

Visual modules are used to implement *ViewModel* presentation models in the system, according to the MVVM pattern. Visual modules encapsulate both the data used in the GUI controls and methods for working with that data. Examples of visual modules are the section, detail and page modules.

Go to the [Configuration] section, click [Add] and select [Schema of the View Model] (Fig. 1, 2) to create a visual module. (Fig. 1, 2).

Replacing client modules

Use replacing client modules if you need to modify or expand the functionality of base modules.

Go to the [Configuration] section, click [Add] and select [Replacing client module] (Fig. 1, 3) to create a replacing client module.

Client module features

The "init()" and "render()" methods

A default bpm'online client module can contain two methods:

- The *init()* method implements the logic that is executed when the module is loaded. This method is called first by the client core if it's been detected upon module loading. The *init()* method usually implements subscriptions to events of other modules and initializes the module values.
- *The render(renderTo)* method implements the module visualization logic. The client core will automatically call this method (if it is available) upon module loading. Before data visualization, the mechanism for binding the view (*View*) and the view model (*ViewModel*) must be triggered for correct

data processing. As a rule, this mechanism is initiated in the *render()* method: the *bind()* method is called in the view object. If the module is loaded into a container, a reference to this container will be passed to the *render()* method as an argument. The visual modules must implement the *render()* method.

Case

Create a module with the *init()* and *render()* methods. Both methods must display a message. The client kernel will first call the *init ()* method and then the *render()* method when the module is loaded. A message must alert you each time a method is called.

NOTE

You can test any visual module, perform client downloads and generate visualization in the base version of bpm'online. To do this, generate the following address string:

[Application URL]/[Configuration Number]/NUI/ViewModule.aspx#[Module name]

Example: http://myserver.com/BPMonlineWebApp/0/Nui/ViewModule.aspx#CustomModule

The *CustomModule* module will be returned to the client, and its visual representation will be displayed in the central area of the application.

Case implementation:

1. Create a client module schema: Go to the [Configuration] section, click [Add] and select [Module] (Fig. 1, 1) to create a non-visual module.

2. Set the [Title] property to "Standard module example" and the [Name] property to "ExampleStandartModule". Select the name of the schema package in the [Package] property.

3. Add the following code to the [Source code] tab:

```
// Declaring the "ExampleStandartModule" module. The module does not have any
dependencies,
// so an empty array is passed as the second parameter.
define("ExampleStandartModule", [],
    // The factory function returns a module object with two methods.
    function () {
        return {
            // The method will be called first by the core upon loading to the
client.
            init: function () {
                alert("Calling the init() method of the "ExampleStandartModule"
module.");
            },
            The method will be called by the kernel when the module is loaded into
the container. The link to the container is passed to the method
            // as the renderTo parameter. A message will display a page control id
element,
            // which has to display the visual data of the module. centerPanel by
default.
            render: function (renderTo) {
                alert("Calling the render() method of the "ExampleStandartModule"
module. The module is uploaded to the container " + renderTo.id);
            }
        };
    });
```

4. Save and publish the schema.

You can run the example by executing the following query: *[Application URL]/[Workplace number]/NUI/ViewModule.aspx#ExampleStandartModule*

Calling a function of a module from another module. Utility modules

Although a module is essentially an isolated software unit, the functions of other modules can be used in its logic. The module with the intended functionality needs to be imported as a dependency for that to occur. Access to the dependency module instance is granted through the factory function argument.

You can group auxiliary and service methods into separate *utility modules* and import them into modules that require this functionality.

Case

Create a standard module with the *init()* and *render()* methods. The method for displaying a message window must be taken out to a separate utility module.

Case implementation:

1. Create a schema for the client module with the following properties:

- Assign "Utility module example" to the [Title] property.
- Assign "ExampleUtilsModule" to the [Name] property.

Select the name of the schema package in the [Package] property.

2. Add the following code to the [Source code] tab:

```
// Declaring a utility module. The module does not have any dependencies and only
contains one method
// for displaying a message.
define("ExampleUtilsModule", [],
    function () {
        return {
            // The method for displaying a message. The message displayed in the
window
            // is passed to the method as the "information" argument.
            showInformation: function (information) {
                alert(information);
            }
        };
    });
```

3. Save and publish the utility module schema.

4. Create a client schema with the following properties:

- [Title]: "Utility module use example".
- [Name]: "UseExampleUtilsStandartModule".

5. Add the following code to the [Source code] tab:

```
// The ExampleUtilsModule dependency module is imported to the module for access to
the utility method.
// The factory function argument - a link to a loaded utility module.
define("UseExampleUtilsStandartModule", ["ExampleUtilsModule"],
    function (ExampleUtilsModule) {
        return {
            // The utility method for displaying a message window is called in the
init() and render() functions
            // with a message which is passed to the utility method as an argument.
            init: function () {
                ExampleUtilsModule.showInformation ("Calling the init() method of the
UseExampleResourceStandartModule module".);
            },
            render: function (renderTo) {
                ExampleUtilsModule.showInformation("Calling the render() method of
the UseExampleUtilsStandartModule module. The module is uploaded to the container " +
renderTo.id);
            }
```

```
        };
    });
```

6. Save and publish the schema.

You can run the example by executing the following query: *[Application URL]/[Workplace number]/NUI/ViewModule.aspx#UseExampleUtilsStandartModule*

Working with resources

Localized strings and images are the resources of the client schema that are most often used in the implementation logic of the module.

Add resources to the client schema in the [Structure] tab of the client schema designer. The application core automatically generates a special [Client module name]Resources module, which contains resources of the client module. The *localizableStrings* property stores schema's localized strings. The *images* property stores image resources.

In order to access a resource module from a client module, you need to import the resource module as a dependency into the client module.

> ⚠ *ATTENTION*
>
> We recommend using localized resources rather than string literals or constants in the module code.

Case

Similar to previous cases, create an *ExampleResourceModule* module with the *init()* and *render()* methods. Use the *ExampleUtilsModule* method of the utility module to display the message window. Contents displayed in message windows must be specified by the values of localized strings in a client schema, rather than string literals.

Case implementation:

1. Create a client module with the following properties:

- [Title]: “Resource module use example”.
- [Name]: “ExampleResourceModule”.

Select the name of the schema package in the [Package] property.

2. In the created schema, add two localizable strings that will be displayed in the messages. To add a localizable string in the [Structure] tab, right-click the [LocalizableStrings] element and select [Add].

Assign the following properties for the message string of the *init()* method:

- [Name]: “InitMessage”.
- [Value]: “Calling the init() method of the UseExampleResourceStandartModule module”.

Assign the following properties for the message string of the *render()* method:

- [Name]: “RenderMessage”.
- [Value]: “Calling the render() method of the UseExampleResourceStandartModule module”.

3. Add the following code to the [Source code] tab:

```
Two dependency modules are loaded into the module: the “ExampleUtilsModule” utility
module, created earlier, and the
// ExampleResourceModuleResources resource module. The resource module is not
explicitly created – it is generated by the core on the basis of
// resources added to the client schema.
define("ExampleResourceModule",
    ["ExampleUtilsModule", "ExampleResourceModuleResources"],
    // Now, the messages in init() and render() are not specified by
    // constant values, but localized strings.
    function (utils, resources) {
        return {
```

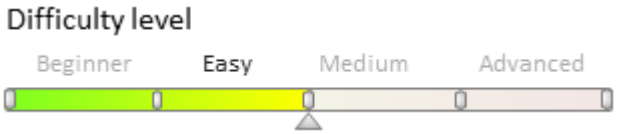
```
            init: function () {
                utils.showInformation(
                    // Access to the localized InitMessage line, in which the message
for the init() method is stored.
                    resources.localizableStrings.InitMessage);
            },
            render: function () {
                utils.showInformation(
                    // Access to the localized RenderMessage line, in which the
message for the render() method is stored.
                    resources.localizableStrings.RenderMessage);
            }
        }
    });
```

4. Save and publish the schema.

Using replacing client modules

The extension modules of the basic functionality do not support inheritance in its traditional sense. You must completely transfer (or copy) the program code of the original module when creating extension modules, and then make your changes in the extension module. Although you do not need to transfer the code of the original module while creating replacing client modules, you still can not use its resources. All resources (localized strings, images) must be duplicated in the replacement schema.

Client view model schemas



Introduction

A *custom view model schema* is a visual module schema that implements client part of the application. Custom view model schema is a kind of configuration object for generating views and view models by the *ViewGenerator* and *ViewModelGenerator* generators of bmp'online. Custom module types are described in the "**Client Modules**" article.

Source code structure of custom schema

All schemas have a common structure. Schema source code example:

```
define("ExampleSchema", [], function() {
    return {
        entitySchemaName: "ExampleEntity",
        mixins: {},
        attributes: {},
        messages: {},
        methods: {},
        rules: {},
        businessRules: /**SCHEMA_BUSINESS_RULES*/{}/**SCHEMA_BUSINESS_RULES*/,
        modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/,
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    };
});
```

A Schema configuration object is returned by anonymous factory function that is called after loading the module. The object can have following properties:

entitySchemaName – object (model) schema name that will be used by this client schema.

mixins – configuration object that contains mixin declaration. More information about the mixins you can find in the "**Mixins. The "mixins" property**" article.

attributes – configuration object that contains schema attributes. More information about the attributes you can find in the "**Attributes. The "attributes" property**" article.

messages – configuration object that contains schema messages. More information you can find in the "**Messages. The "messages" property**" article.

methods – configuration object that contains schema methods. More information about this property you can find in the "**Methods. The "methods" property**" article.

rules – configuration object that contains schema business rules. More information about this property you can find in the "**Rules. The "rules" property**" article.

businessRules — configuration object that contains schema business rules, which are created or edited via the section wizard or detail wizard. Marker comments /**SCHEMA_BUSINESS_RULES*/ are used by the wizards and therefore are required. More information about this property you can find in the "**Business rules. The businessRules property**" article.

modules – configuration object that contains schema modules. Marker comments / ** SCHEMA_MODULES * / are used by the wizards and therefore are required. More information about this property you can find in the "**Modules. The "modules" property**" article.

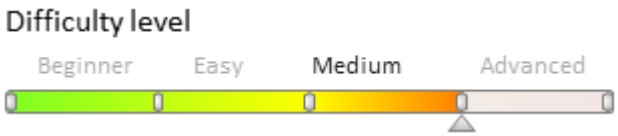
> NOTE
>
> To load a detail on a page the *details* property is used. But the detail is a module and it is appropriate to use the *modules* property.

diff – configuration object array that contains schema view description. Marker comments / ** SCHEMA_DIFF * / are used by the wizards and therefore are required. For more information about the *diff* array configuration, please refer to the "**The "diff" array**" article.

properties — configuration object which contains the view model properties. Detailed information about this property is available in the "**Properties. The "properties" property**" article.

$-properties — automatically generated properties for the view model attributes. More information can be found in the "**Automatically generated view model properties**" article.

Mixins. The "mixins" property



Introduction

A mixin is a class designed to extend the functions of other classes. Mixins are separately created classes with additional functionality. Mixins expand the functionality of schemas, allowing to avoid duplication of commonly used logic in schema methods. Mixins are different from other modules added to the dependency list in a way that their methods can be addressed directly, much like those of a schema.

Using mixins

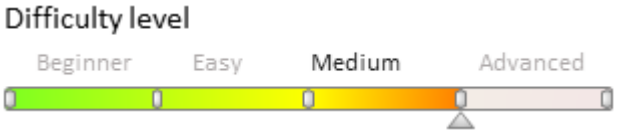
```
// WizardUtilities - a module in which the mixin class is implemented.
define("ExampleSchema", ["WizardUtilities"], function () {
    return {
        entitySchemaName: "Contact",
        mixins: {
            // Connecting mixins.
            WizardUtilities: "Terrasoft.WizardUtilities"
        },
```

```
        attributes: {},
        messages: {},
        methods: {},
        rules: {},
        modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/,
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    };
});
```

The mixin functionality will be available in a schema it was added to.

> ⚠ ATTENTION
>
> Mixins are designed in form of modules that must be connected to the schema dependencies list when it is declared by the define function.

Attributes. The "attributes" property



Introduction

Attributes is a configuration object property of the view model schema. It contains configuration objects that describe the model attributes. A model column is the attribute. All object schema columns are included in the *attributes* collection automatically upon generation.

Attribute base properties

The schema attributes have the following base properties:

- *dataValueType* – attribute data type. This property is used for generation of views. The available data types are represented by the *Terrasoft.DataValueType* enumeration.
- *type* – column type. Optional parameter used in the *BaseViewModel* internal work. The available column types are represented by the *Terrasoft.ViewModelColumnType* enumeration.
- *value* – the attribute value. The value of this parameter will be set in the view model at its creation.

> ⚠ Attention!
>
> You can specify numeric, string and Boolean values in the *value* attribute.
>
> If the attribute type involves the use of a reference type value (array, object, collection, etc.), its initial value must be initialized using methods.

An example of using attribute base properties:

```
attributes: {
  // Attribute name.
  "NameAttribute": {
    // Data type.
    "dataValueType": this.Terrasoft.DataValueType.TEXT,
    // Column type.
    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
    // Default value.
    "value": "NameValue"
  }
}
```

Attributes additional properties

Attributes can have the following properties:

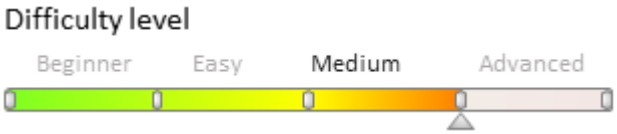
- *Caption* – attribute title.
- *isRequired* – indicates whether the attribute is required.
- *Dependencies* – dependency from another model attribute. For example, setting dependencies of an attribute on the value of another attribute. The property is used to create calculated fields. More information about the calculated fields and the uses of this parameter can be found in the "**Adding calculated fields**” article.
- *lookupListConfig* – property that configures lookup field features. More information about this parameter can be found in the "**Using filtration for lookup fields. Examples**” article. This is a configuration object that can contain the following optional properties:
  - columns – an array of column names that will be added to the query with the Id column and the primary display column.
  - orders – an array of configuration objects that determine the sorting of displayed data.
  - filter – a method that returns an object of *Terrasoft.BaseFilter* class or its inheritor that will be applied to the query. Can not be used with the “filters” property.
  - filters – filters array (methods that return collections of the *Terrasoft.FilterGroup* class). Can not be used with the filter property.

An example of using attribute additional properties:

```
attributes: {
  // Attribute name.
  "Client": {
    // Attribute header.
    "caption": { "bindTo": "Resources.Strings.Client" },
    // Attribute  is required.
    "isRequired": true
  },

  // Attribute name.
  "ResponsibleDepartment": {
    lookupListConfig: {
      // Additional columns.
      columns: [ "SalesDirector" ],
      // Sort column.
      orders: [ { columnPath: "FromBaseCurrency" } ],
      // Filter definition function.
      filter: function()
      {
        // Returns filter of Type column, which is equal the "Competitor" constant.
        return this.Terrasoft.createColumnFilterWithParameter(
        this.Terrasoft.ComparisonType.EQUAL,
        "Type",
        ConfigurationConstants.AccountType.Competitor);
      }
    }
  },
  // Attribute name.
  "Probability": {
    // Determination of the column dependency.
    "dependencies": [
      {
        // Depends on the "Stage" column.
        "columns": [ "Stage" ],
        // The name of the handler method for the "Stage" column change.
        // setProbabilityByStage() method is defined  in methods property
        // of schema object.
        "methodName": "setProbabilityByStage"
```

```
      }
    ]
  }
},
methods: {

  // "Stage" column modification handler method
  setProbabilityByStage: function()
  {
    // Getting the Stage column value.
    var stage = this.get("Stage");
    // The condition for the "Probability" column modification.
    if (stage.value && stage.value ===
        ConfigurationConstants.Opportunity.Stage.RejectedByUs)
    {
      // Setting the "Probability" column value.
      this.set("Probability", 0);
    }
  }
}
```

Messages. The "messages" property



Introduction

Data exchange between modules is organized through messages.

There are two message modes:

- *Address*. Address messages are only received by the last subscriber. To switch to address mode, set the *mode* property to *this.Terrasoft.MessageMode.PTP*.
- *Broadcasting*. Broadcasting messages are received by all subscribers. To switch to broadcasting mode, set the *mode* property to *this.Terrasoft.MessageMode.BROADCAST*.

The list of available message modes is represented by the *Terrasoft.MessageMode* enumeration.

There are two message directions:

- *Publication* – a message that can only be published (outbound). To set the direction for message publishing, set the *direction* property to *this.Terrasoft.MessageDirectionType.PUBLISH*.
- *Subscription* – a message that can only be subscribed to (inbound). To set the direction for message subscription, set the *direction* property to *this.Terrasoft.MessageDirectionType.SUBSCRIBE*.

> ⚠ ATTENTION
>
> The same message can not be announced with different directions in the schema inheritance hierarchy.

Message use examples

Message publishing

Declare a message with the "publishing" direction in the schema you want to publish the message in.

```
messages: {
    // Message name.
    "GetColumnsValues": {
```

```
        // Message type - address.
        mode: this.Terrasoft.MessageMode.PTP,
        // Message direction - publication
        direction: this.Terrasoft.MessageDirectionType.PUBLISH
    }
}
```

Publishing is done through calling the publish method from the *sandbox* class instance.

```
// The GetColumnsValues method for obtaining message publication resuls.
getColumnsValues: function(argument) {
    // Message publishing.
    return this.sandbox.publish("GetColumnsValues", argument, ["key"]);
}
```

In this code:

- "GetColumnsValues" – message name.
- *Argument* – the argument passed to the handler function of the subscriber. An object with message parameters.
- ["Key"] – an array of tags for filtering messages.

The *sandbox* property is declared in all schemas.

> ⚠ ATTENTION
>
> Message publishing can return the handler function result only in the "address" mode.

Message subscription

A message with the "subscription" direction should be declared in the subscription schema.

```
messages: {
    // Message name.
    "GetColumnsValues": {
        // Message type - address.
        mode: this.Terrasoft.MessageMode.PTP,
        // Message direction - subscription.
        direction: this.Terrasoft.MessageDirectionType.SUBSCRIBE
    }
}
```

A subscription is carried out by calling the *subscribe* method in the *sandbox* class instance.

```
this.sandbox.subscribe("GetColumnsValues", messageHandler, context, ["key"]);
```

In this code:

- "GetColumnsValues" – message name.
- *messageHandler* – message handler function.
- *Context* – handler function execution context.
- ["Key"] – an array of tags for filtering messages.

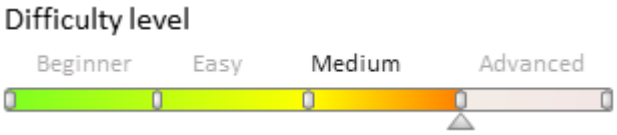
In the "address" mode, the *messageHandler* method should return the object, which is processed as the result of message publication.

```
methods: {
    messageHandler: function(args) {
    // Returning the object that is being processed as a result of message
publishing.
    return { };
    }
}
```

In broadcast mode, the *messageHandler* method returns nothing.

```
methods: {
    messageHandler: function(args) {
    }
}
```

Methods. The "methods" property



Introduction

The methods property of the view model schema contains a collection of methods that form the business logic of the schema and affect the view model. Create new methods and override (replace) base methods of parent schemas in this property. By default, the scope of methods is the view model scope.

Examples of method declaration

An example of a replaced method

Add the [Email] column completion requirement logic to the *setValidationConfig* method logic of the *Terrasoft.configuration.BaseSchemaViewModel* class.

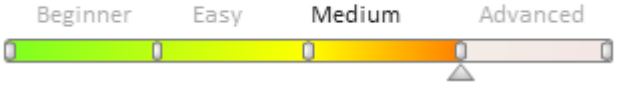
```
methods: {
    // Method name.
    setValidationConfig: function() {
        // Calling the logic of the setValidationConfig parent schema method.
        this.callParent(arguments);
        // Setting up the validation for the [Email] column.
        this.addColumnValidator("Email", EmailHelper.getEmailValidator);
    }
}
```

An example of a new method

```
methods: {
    // Method name.
    getBlankSlateHeaderCaption: function() {
        // Accessing the MasterColumnInfo column values.
        var masterColumnInfo = this.get("MasterColumnInfo");
        // Returning method work results.
        return masterColumnInfo ? masterColumnInfo.caption : "";
    },
    // Method name.
    getBlankSlateIcon: function() {
        // Returning method work results.
        return
this.Terrasoft.ImageUrlBuilder.getUrl(this.get("Resources.Images.BlankSlateIcon"));
    }
}
```

Rules. The "rules" property

Difficulty level



Introduction

Rules is a standard system mechanism, which enables the developer to add an implementation of typical functions by configuring view model columns.

The functions of rules are implemented in the *BusinessRuleModule* client module. Add the *BusinessRuleModule* module to the list of schema dependencies to use these functions.

```
define("CustomPageModule", ["BusinessRuleModule"],
    function(BusinessRuleModule) {
        return {
            // Client module implementation
        };
    });
```

Rule types are defined in the *RuleType* enumeration of the *BusinessRuleModule* module.

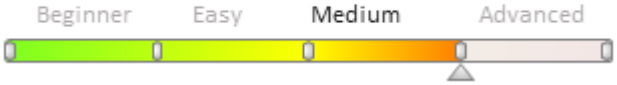
General procedure for declaring the rules:

- All rules are described in the *rules* property of the schema.
- The rules are applied to view model columns, not to controls.
- Rules have names.
- Rule parameters are set in its configuration object.

To learn more about business rules and to see the examples of their use, please refer to the “**Setting the edit page fields using business rules**” chapter.

Business rules. The businessRules property

Difficulty level



Introduction

In bpm’online, the behavior configuration of page / detail fields is done through business rules. Using business rules, you can configure the following field behavior:

- Hiding and displaying fields
- Enable or disable editing
- Compulsory or optional
- Filtering lookup fields depending on other field values

Unlike the business rules defined in the *rules* property of the page view model schema (see “ **Rules. The "rules" property**”), the business rules defined in the *businessRules* property are generated by the detail or the section wizard and have a higher execution priority. The *BusinessRuleModule* enumeration is not used when describing the generated business rule**.**

When creating a new business rule, the wizard generates a name for it and adds it to the custom schema of the edit page view model.

If the business rule is disabled, the *enabled* property of its configuration object is set to *false*.

When you delete a business rule, its configuration object remains in the custom schema of the edit page view model, but the *removed* property is set to *true*.

ATTENTION

We do not recommend editing the *businessRules* property manually!

Editing an existing business rule

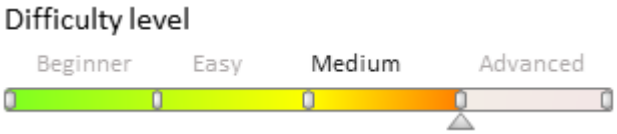
After editing the custom business rule in the wizard, the business rule configuration object remains unchanged in the *rules* property of the edit page view modelю This creates a new version of the business rule configuration object with the same name in the *businessRules* property.

The business rule defined in the *businessRules* property has a higher execution priority when processing a business rule at runtime. Therefore, subsequent changes to this rule in the *rules* property will not affect the system in any way.

NOTE

When you delete or disable the business rule, the changes made in the configuration object of the *businessRules* property have a higher priority.

Modules. The "modules" property



Introduction

The *modules* property contains a configuration object responsible for declaration and configuration of modules and details loaded to a page. The / ** SCHEMA_MODULES * / marker comments are required, since they are necessary for the work of the wizards.

NOTE

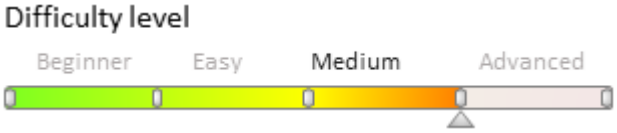
To load a detail to a page, use the *details* property. However, since details are essentially modules, we recommend using the *modules* property instead.

An example of using the modules property

```
modules: /**SCHEMA_MODULES*/{
    // Loading the module
    // Module title. Must be the same as the name property in the diff massive.
    "TestModule": {
        // Optional Loaded module id Will be generated by the system if not
specified.
        "moduleId": "myModuleId",.
        // If the parameter is not specified, BaseSchemaModuleV2 will be used for
loading.
        "moduleName": "MyTestModule",
        // Configuration object. When the module is loaded, it is passed as
instanceConfig. It stores a set of initial parameter values for the module.
        "config": {
            "isSchemaConfigInitialized": true,
            "schemaName": "MyTestSchema",
            "useHistoryState": false,
            // Additional module parameters.
            "parameters": {
                // Parameters added to a schema during its initialization.
```

```
                "viewModelConfig": {
                    masterColumnName: "PrimaryContact"
                }
            }
        }
    },

    // Loading a detail.
    // Detail name.
    "Project": {
            // The name of a schema detail.
            "schemaName": "ProjectDetailV2",
            "filter": {
            // Section object schema column.
            "masterColumn": "Id",
            // Detail object schema column.
            "detailColumn": "Opportunity"
        }
    }
}/**SCHEMA_MODULES*/
```

The "diff" array



Introduction

The “diff” array is an array of modifications described in the “*diff*” property of a schema. The array is used for generating module views in the system UI. Each array element is a metadata. The UI controls are generated based on these metadata.

The *diff* property contains an array of configuration objects that are responsible for schema display. The *diff* array contains objects that configure display of containers, controls, modules, fields and other visual components.

The *diff* array object properties

The *diff* array elements have the following properties:

- *operation* – can have the following values:
  - *set* – schema element value is set by the *values* parameter.
  - *merge* – the values from the parent, replacing and replacement schemas are merged. The properties from *values* parameter have the highest priority.
  - *remove* – the element is removed from the schema.
  - *move* – the element is moved to another parent element.
  - *insert* – the element is inserted in the schema.
- *name* – the name of schema element that the operation is applied to.
- *parentName* – the name of schema parent element where the element is placed as a result of the *insert* or *move* operation;
- *propertyName* – the name of parent element parameter in the *insert* operation. Also used in the *remove* operation if it is needed to remove specific element parameters and not the element itself;
- *index* – the index in which the parameter is being moved or inserted. The parameter is used in the *insert* and *move* operations. If the parameter is not specified, the element will be inserted as the last element in the array.
- *values* – the object whose properties will be set or merged with schema element properties. It is used in

the *set*, *merge* and *insert* operations.

Bpm'online has a set of basic elements that can be displayed on a page. They are specified in the *Terrasoft.ViewItemType* list (Table. 1).

Table 1. – Element type

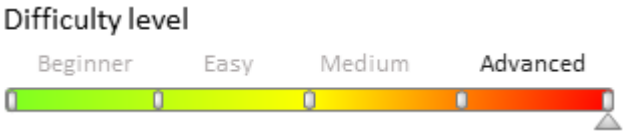
Name	Description
GRID_LAYOUT	Grid element that contains placements of other elements
TAB_PANEL	Set of tabs.
DETAIL	Detail.
MODEL_ITEM	View model element.
MODULE	Module.
BUTTON	Button.
LABEL	Label.
CONTAINER	Container.
MENU	Drop-down list.
MENU_ITEM	Drop-down list element.
MENU_SEPARATOR	Drop-down list separator.
SECTION_VIEWS	Section views.
SECTION_VIEW	Section view.
GRID	List.
SCHEDULE_EDIT	Scheduler.
CONTROL_GROUP	Group of controls.
RADIO_GROUP	Group of radio buttons.
DESIGN_VIEW	Customizable view.
COLOR_BUTTON	Color.
IMAGE_TAB_PANEL	Set of tabs with icons.
HYPERLINK	Hyperlink.
INFORMATION_BUTTON	Information button with tooltip.
TIP	Tooltip.

An example of using the “diff” property

```
diff: /**SCHEMA_DIFF*/[
  {
      "operation": "insert",
      "name": "CardContentWrapper",
      "values": {
          "id": "CardContentWrapper",
          "itemType": Terrasoft.ViewItemType.CONTAINER,
          "wrapClass": ["card-content-container"],
          "items": []
      }
  },
  {
      "operation": "insert",
      "name": "CardContentContainer",
      "parentName": "CardContentWrapper",
```

```
        "propertyName": "items",
        "values": {
            "itemType": Terrasoft.ViewItemType.CONTAINER,
            "items": []
        }
    },
    {
        "operation": "insert",
        "name": "HeaderContainer",
        "parentName": "CardContentContainer",
        "propertyName": "items",
        "values": {
            "itemType": Terrasoft.ViewItemType.CONTAINER,
            "wrapClass": ["header-container-margin-bottom"],
            "items": []
        }
    },
    {
        "operation": "insert",
        "name": "Header",
        "parentName": "HeaderContainer",
        "propertyName": "items",
        "values": {
            "itemType": Terrasoft.ViewItemType.GRID_LAYOUT,
            "items": [],
            "collapseEmptyRow": true
        }
    }
]/**SCHEMA_DIFF*/
```

Alias mechanism



General Information

The *Alias* mechanism – provides partial backward compatibility when user interface is changed in the new versions of the product. In the process of developing new versions, sometimes it becomes necessary to move page elements to new areas. In case the users have customized the page, the changes could lead to unpredictable consequences. The *Alias* mechanism helps to avoid this by interacting with the json-applier class which is *diff* array builder. This class merges all the parameters of the base and custom replacing schemas.

Diff – an array of objects, responsible for displaying schema elements. It can have containers, controls, modules and fields. For more information about the *diff* array, see the "**The "diff" array**” article.

Details

The *alias* property contains information about the previous element name. This information is taken into account when building a *diff* array of modifications, and informs that both the elements with a new name, and the elements with the name specified in *alias* must be considered. The *alias* is a configuration object that links two different elements – the new one and the old one. When building a *diff* array the *alias* configuration object can be used to exclude application of certain properties and operations to the element in which the alias is declared. The *alias* object can be added to any element in *diff* array.

Alias object structure

The *alias* object contains three custom properties:

- *name* – the name associated with the new element. This name will be used to locate the elements in the replaced schemas and connect them with the new element.

> ⚠ ATTENTION
>
> The value of the *name* element of the *diff* array should not be equal to the *alias.name* property.

- *excludeProperties* – array of properties of the *values* object of the *diff* modification array element. These properties will not be used when generating *diff*.
- *excludeOperations* – array of operations that should not be applied to this element when the *diff* modification array is generated.

Usage example of the *alias* object :

```
// diff. array
diff: /**SCHEMA_DIFF*/ [
  {
    // The operation with the element.
    "operation": "insert",
    // Element new name.
    "name": "NewElementName",
    // Element values.
    "values": {
      // ...
    },
    // Alias configuration object.
    "alias": {
      //  Element previous name.
      "name": "OldElementName",
      // Exclude properties array.
      "excludeProperties": [ "layout", "visible", "bindTo" ],
      // Exclude operations array.
      "excludeOperations": [ "remove", "move", "merge" ]
    }
  },
  ///...
]
```

An example of the Alias mechanism usage for multiple schema replacement

There is an initial element of the *diff* array with the name "Name" and a set of properties. The element is located in the *Header* container. This schema was replaced several times and each time the "Name" element is moved and modified in every possible way.

Diff property of the base schema

```
diff: /**SCHEMA_DIFF*/ [
  {
    // Insert operation.
    "operation": "insert",
    // The name of the parent element to insert into.
    "parentName": "Header",
    // The name of the parent element with which operation is performed.
    "propertyName": "items",
    // Element name.
    "name": "Name",
    // Element property values object.
    "values": {
      // Layout.
```

```
      "layout": {
        // Column number.
        "column": 0,
        // Row number.
        "row": 1,
        // Number of joined columns.
        "colSpan": 24
      }
    }
  }
] /**SCHEMA_DIFF*/
```

Diff property after first replacement of the base schema:

```
diff: /**SCHEMA_DIFF*/ [
  {
    // Merging properties of the two elements.
    "operation": "merge",
    "name": "Name",
    "values": {
      "layout": {
        "column": 0,
        // Row number. The element is moved.
        "row": 8,
        "colSpan": 24
      }
    }
  }
] /**SCHEMA_DIFF*/
```

Diff property after second replacement of the base schema:

```
diff: /**SCHEMA_DIFF*/ [
  {
    //Moving the element.
    "operation": "move",
    "name": "Name",
    //The name of the parent element where the element is moved.
    "parentName": "SomeContainer"
  }
] /**SCHEMA_DIFF*/
```

In the new version, the "Name" element was moved from the *SomeContainer* element to the *ProfileContainer* element and must remain there regardless of the client customization. For this, the element gets a new name "NewName" and an *alias* configuration object is added to it.

```
diff: /**SCHEMA_DIFF*/ [
  {
    // Insert operation.
    "operation": "insert",
    // The name of the parent element in which insert is carried out.
    "parentName": "ProfileContainer",
    // The name of the parent element property with which operation is performed.
    "propertyName": "items",
    // Element new name.
    "name": "NewName",
    // Object with element property values.
    "values": {
      // Binding to a property value or a function
      "bindTo": "Name",
      // Layout.
      "layout": {
```

```
        // Column number.
        "column": 0,
        // Row number.
        "row": 0,
        // Number of joined columns.
        "colSpan": 12
      }
    },

    // Alias configuration object.
    "alias": {
      // Element previous name.
      "name": "Name",
      // Array of excluded properties.
      "excludeProperties": [ "layout" ],
      // Array of ignored operations.
      "excludeOperations": [ "remove", "move" ]
    }
  }
] /**SCHEMA_DIFF*/
```

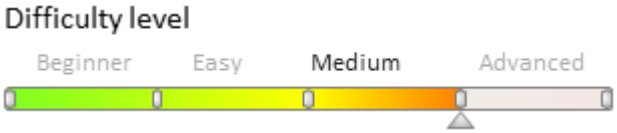
Alias has been added in the new element. The parent element and its location on the edit page also has been changed. The *excludeProperties* property stores a set of properties that will be ignored when the difference is applied. The *excludeOperations* property stores a set of operations that will not be applied to the element from replacements.

In this example *layout* properties of all “Name” element inheritors are excluded and *remove* and *move* operations are not allowed. This indicates that the "NewName” element will only contain a root *layout* property and all of the "Name" element properties from replacements except *Layout*. Same applies to operations.

The result for the *diff* array builder will be:

```
diff: /**SCHEMA_DIFF*/ [
  {
    // Insert operation.
    "operation": "insert",
    // The name of the parent element in which insert is carried out.
    "parentName": "ProfileContainer",
    // The name of the parent element property with which operation is performed.
    "propertyName": "items",
    // Element new name.
    "name": "NewName",
    //Object with element property values.
    "values": {
      // Bind to a property value or a function
      "bindTo": "Name",
      // Layout.
      "layout": {
        // Column number.
        "column": 0,
        // Row number.
        "row": 0,
        //  Number of joined columns.
        "colSpan": 12
      },
    }
  },
] /**SCHEMA_DIFF*/
```

Schema formatting requirements for compatibility with wizards



General information

In general, a client schema has three components:

1. Automatically generated code that contains a schema description, its dependencies, localized resources, and messages.
2. Visualization styles (may not be present in certain types of client schemas).
3. Schema code – syntactically correct JavaScript code that defines the module.

Changes made to client schemas using wizards (adding a field, changing the tab position, adding a detail or a module to the edit page layout, etc.) are saved by modifying the *diff*, *modules*, *details* and *businessRules* properties of the schema structure. For more information about the schema structure, please see the "**Client view model schemas**” article. Due to technical limitations, marker comments for these properties are used to identify them uniquely in the schema code.

Marker comments

Marker comments identify the *diff*, *modules* and *details* properties of the schema structure if it is edited with the help of wizards.

In addition to the basic validation, the checking procedure for client schemas will indicate the schemas without the necessary comments when a wizard is run. Schema validation rules are given in Table 1.

Table 1. Schema validation rules

Schema type	Required marker comments
View model schema of the *EditViewModelSchema* edit page	details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/, modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/, diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/ businessRules: /**SCHEMA_BUSINESS_RULES*/{}/**SCHEMA_BUSINESS_RULES*/
View model schema of *ModuleViewModelSchema* section	modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/, diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
View model schema of *EditControlsDetailViewModelSchema* detail with edit fields	modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/, diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
View model schema of *DetailViewModelSchema* detail	modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/, diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
View model schema of *GridDetailViewModelSchema* detail with editable list	modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/, diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/

The schema types are determined by the *ClientUnitSchemaType* enumeration.

The declaration rules of the *diff* property

The *diff* property contains an array of configuration objects that are responsible for schema display. The *diff* array may contain objects that configure display of containers, controls, modules, fields and other visual components. For more information about the diff array, see the "**The "diff" array**” article.

Proper use of converters

The *converter* is a function executed in the *viewModel* environment that receives *viewModel,* properties and returns a result of the corresponding type. For the wizards to work correctly, the value of the *diff* property must be in JSON format. Therefore, the converter value must be the name of the view model method, and not the *inline* function.

An example the converter improper use:

```
diff: /**SCHEMA_DIFF*/[
    {
        //...
        "bindConfig": {
            converter: function(val) {
            // ...
            }
        }
    }
]/**SCHEMA_DIFF*/
```

An example the converter proper use:

```
methods: {
    someFunction: function(val) {
        //...
    }
},

diff: /**SCHEMA_DIFF*/[
  {
      //...
      "bindConfig": {
          "converter": "someFunction"
      }
      //...
  }
]/**SCHEMA_DIFF*/
```

Parent element (container)

Parent element is a DOM element into which the module draws its view. For correct work of the wizard, it is necessary that the parent container have only one child element.

An example of incorrect view placement in parent element:

```
<div id="OpportunityPageV2Container" class="schema-wrap one-el" data-item-
marker="OpportunityPageV2Container">
    <div id="CardContentWrapper" class="card-content-container page-with-left-el"
data-item-marker="EntityLoaded"></div>
    <div id="DuplicateContainer" class="DuplicateContainer"></div>
</div>
```

An example of correct view placement in parent element:

```
<div id="OpportunityPageV2Container" class="schema-wrap one-el" data-item-
marker="OpportunityPageV2Container">
    <div id="CardContentWrapper" class="card-content-container page-with-left-el"
data-item-marker="EntityLoaded"></div>
</div>
```

When adding, changing, moving an element in the *diff* (the *insert*, *merge*, *move* operations), the *parentName* property (the parent element name) is required.

An example of incorrect view element specification in the *diff* property:

```
{
  "operation": "insert",
  "name": "SomeName",
  "propertyName": "items",
  "values": {}
}
```

An example of correct view element specification in the *diff* property:

```
{
  "operation": "insert",
  "name": "SomeName",
  "propertyName": "items",
  "parentName": "SomeContainer",
  "values": {}
}
```

In case if *parentName* property is missing, at the wizard launch, an error will be displayed, indicating that the page cannot be set up by the wizard.

The *parentName* property value must match the name of the parent element in the corresponding base page schema. For example, for edit pages, it is "CardContentContainer".

The Name uniqueness

Each new *diff* array element must have a unique name.

An example of incorrect adding of elements to the *diff* array:

```
{
  "operation": "insert",
  "name": "SomeName",
  "values": { }
},
{
  "operation": "insert",
  "name": "SomeName",
  "values": { }
}
```

An example of correct adding of elements to the *diff* array:

```
{
  "operation": "insert",
  "name": "SomeName",
  "values": { }
},
{
  "operation": "insert",
  "name": "SomeSecondName",
  "values": { }
}
```

The non-existing parent element

If you specify the name of a non-existing container element as the parent element in the *parentName* property, the "Schema cannot have more than one root object" error will occur, since the added element will be placed in the root container.

The placement of view elements

In order to be able to customize and modify the view elements, they must be located on the markup grid. In the bpm'online, each grid row has 24 cells (columns). The *layout* property is used to place elements on the grid.

The grid element properties:

column – left column index

row – upper row index

colSpan – the number of columns occupied

rowSpan – the number of rows occupied

An example of element placement:

```
{
  "operation": "insert",
  "parentName": "ParentContainerName",
  "propertyName": "items",
  "name": "ItemName",
  "values": {
      // Element location.
     "layout": {
         // Start with a "0" column.
        "column": 0,
        // Place in the 5th row of the grid.
        "row": 5,
        // Take 12 columns wide.
        "colSpan": 12,
        // Take 1 row height.
        "rowSpan": 1
     },
     "contentType": Terrasoft.ContentType.ENUM
  }
}
```

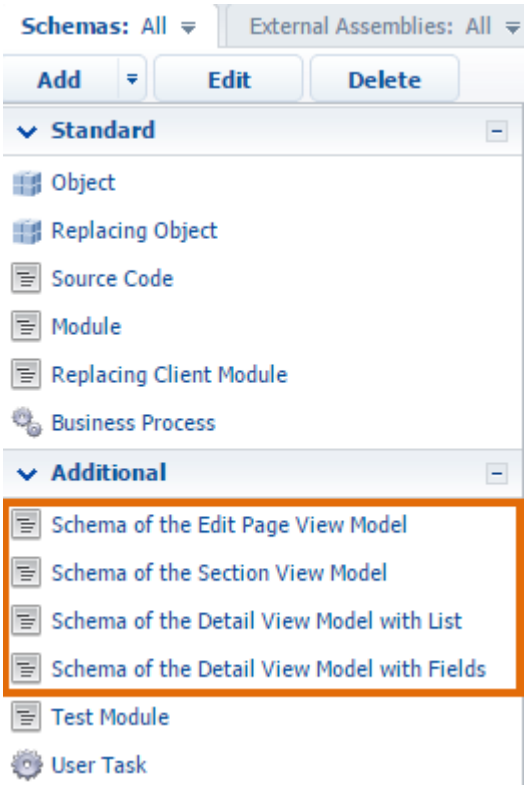
Number of operations

If the client schema is changed without using a wizard, it is recommended to add no more than one operation for one element in the edited schema for the correct operation of the wizard.

Inheritance rules

It is obligatory for the client schema to be a descendant of the *BaseModulePageV2* base schema. It is recommended to create client schemas using the menu commands in the [Configuration] section (Figure 1) or with the help of the wizards.

Fig. 1. The commands for client schemas creation that are compatible with wizards



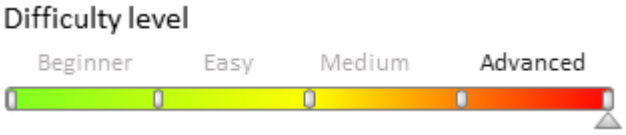
Specifying an object schema for a client schema

In the client schema, you must fill in the *entitySchemaName* property in which the object (model) schema name must be specified. It is sufficient to specify it in one of the inheritance hierarchy schemas.

An example of the *entitySchemaName* property declaration:

```
define("ClientSchemaName", [], function () {
    return {
        // Object schema (model).
        entitySchemaName: "EntityName",
        //...
    };
});
```

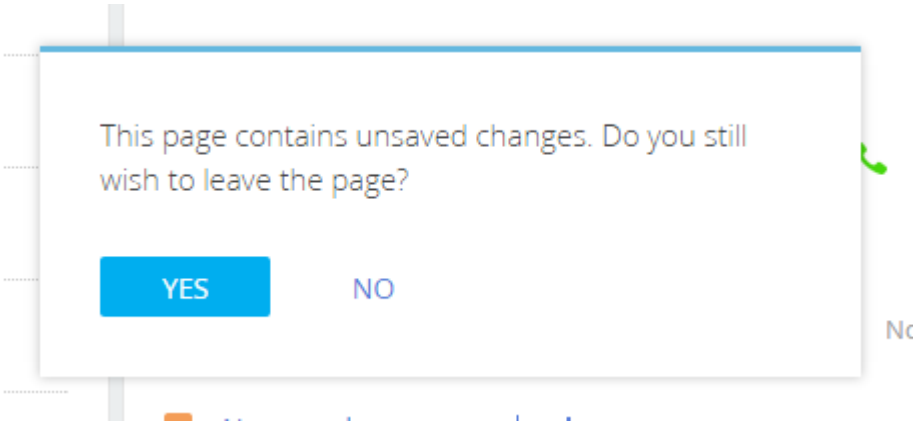
Handling a data context loss



Introduction

The mechanism that tracks data context loss on a record edit page enables you to detect changes (for example, when you go from a page to another section) automatically. If any changes were made on the page, the user will be notified about the unsaved changes and will be able to return to editing or leave the page without saving (Fig. 1).

Fig. 1. Potential changes loss warning



The structure of the context data loss tracking mechanism on a page

The structure of the context data loss tracking mechanism on a page includes:

- New methods and events of the *Terrasoft.Component* base component
- The *CheckModuleDestroyMixin* mixin.
- New and modified methods of the *BasePageV2* base editing page.

New methods and events of the Terrasoft.Component base component.

The “canExecute()” method

The *canExecute()* method calls the *canExecute* event, and adds the *callback* function to its parameters – a link to the *onClick()* method, in which the *canExecute()* method was called. The *canExecute* event connects to the view model and calls the *canBeDestroyed()* method of the *CheckModuleDestroyMixin* mixin. This method may return *false* if there are any unsaved changes on the editing page. If all changes are saved, the method always returns *true*, and the execution of *onClick()* is not interrupted. The *canExecute()* method implementation:

```
// Generates the canExecute event.
// Returns true, if the onClick() method execution can continue.
canExecute: function(config) {
    var args = config.args;
    var event = args[args.length - 1];
    // If the onClick() method was called from the CheckModuleDestroyMixin mixin as a
callback function,
        // then the last parameter signaling the interruption is passed to the event.
    if (event && event.isComeBack) {
        return true;
    }
    // Adding the event of method interruption to the arguments.
    // Required for stopping the execution of the method when onClick is called from
the callback function.
    Array.prototype.push.call(config.args, {
        isComeBack: true
    });
    // Applying the current context.
    Ext.apply(config, { scope: this });
    // Generating the canExecute event.
    var canExecute = this.fireEvent("canExecute", config);
    return canExecute;
},
```

The process of calling the *canExecute()* method occurs in the *onClick()* method of certain control elements (or in all other elements, called in the *onClick()* method). These control elements include the *Terrasoft.Button* base button, the *Terrasoft.Grid* list, the *Terrasoft.BaseMenuItem* menu elements, etc. If necessary, the process of calling this

method can be added to other components as well. An example of the *canExecute()* method calling process in the *Terrasoft.Grid* component:

```
//  The method that indicates the active list record.
setActiveRow: function(newId) {
    var oldId = this.activeRow;
    if (!oldId && !newId) {
        return;
    }
    if (newId !== oldId) {
        var canExecute = oldId && this.canExecute({
            method: this.setActiveRow,
            args: arguments
        });
        // If CheckModuleDestroyMixin returns false, the method execution is
interrupted.
        if (canExecute === false) {
            return;
        }
        ...
    }
},
```

The *OnGridClick()* method handler is called upon clicking the *Terrasoft.Grid* class instance. *OnGridClick()* calls the *setActiveRow()* method in its turn. If there's an existing active string, the *canExecute()* method is called in the *setActiveRow()* method. The configuration object is passed to the *canExecute()* method as an argument. The link to the *setActiveRow()* method and the arguments of *setActiveRow() (which were used to call the method) are added to the properties of the configuration object*. If the called event *canExecute* returns *false*, the work of the *setActiveRow()* method will be interrupted. However, the calling process of the setActiveRow() method can be restored since it was passed as a callback function.

The "canExecute" event

The canExecute event is called in the *Terrasoft.Component* base component constructor, and its generation occurs in the *canExecute()* method.

```
// Class constructor.
constructor: function(config) {
    this.addEvents(
        ...
        "canExecute"
    );
    ...
}
```

In order to control component behavior in the configuration schema, you must associate the view model with the *canExecute* event. To do this, in the *diff* modification array of the *BaseSectionV2* base section schema, define the binding of the *canExecute* event to the *canBeDestroyed* method in the *DataGrid* element. The *canBeDestroyed* method is defined in the *CheckModuleDestroyMixin* mixin.

```
// Modifications array of the basic list schema.
diff: [
    // Adding a list element.
    {
        "operation": "insert",
        "name": "DataGrid",
        "parentName": "DataGridContainer",
        "propertyName": "items",
        "values": {
            ...
            "canExecute": {"bindTo": "canBeDestroyed"},
            ...
```

```
        }
    }...
```

The "CheckModuleDestroyMixin" mixin

The *CheckModuleDestroyMixin* mixin provides message exchange with open edit pages (including those that are open in a chain). Additionally, it is used to interrupt the execution of the method that called *canExecute()* (for example, the *onClick()* button method). This mixin is "mixed" into the view model of the configuration schema.

The message exchange with short editing pages is established by the *updateCanBeDestroyedConfig()* method:

```
// Updates the information about the ability to delete editing page context.
updateCanBeDestroyedConfig: function() {
    // Creating a key.
    var cacheKey = this.Ext.String.format("{0}-cache", this.sandbox.id);
    // Saving to client cache by object key, used by the cards.
    this.Terrasoft.ClientPageSessionCache.setItem(cacheKey, { canBeDestroyed: true
});
    // Publishing a system broadcast message.
    this.sandbox.publish("CanBeDestroyed", cacheKey);
    // Updating work results of message subscribers.
    this.canBeDestroyedConfig =
this.Terrasoft.ClientPageSessionCache.getItem(cacheKey);
}
```

The display of the edit page message dialog and the result processing of user selection are both implemented in the *showCanBeDestroyed()* method:

```
// Displays a message in a dialog.
showCanBeDestroyed: function(resumeConfig) {
    // A message from the page, displayed in the confirmation dialog.
    var message = this.getDestroyedMessage();
    // Display the confirmation dialog.
    this.Terrasoft.showConfirmation(message, function(returnCode) {
        // If the user selects "Yes", the callback function is called.
        // This will continue execution of the interrupted handler for the component
click event.
        if (returnCode === this.Terrasoft.MessageBoxButtons.YES.returnCode) {
            this.Ext.callback(resumeConfig.method, resumeConfig.scope || this,
resumeConfig.args);
        }
    }, ["yes", "no"], this);
}
```

New and modified methods of the BasePageV2 base editing page

The subscription to the *CanBeDestroyed* message (sent out by the *CheckModuleDestroyMixin* mixin) is implemented in the *subscribeSandboxEvents()* method:

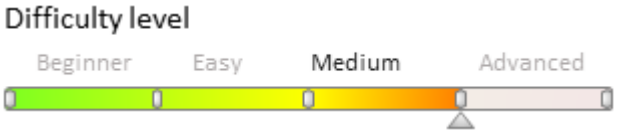
```
// Message subscription.
subscribeSandboxEvents: function() {
   ...
   this.sandbox.subscribe("CanBeDestroyed", this.onCanBeDestroyed, this);
   ...
}
```

The *onCanBeDestroyed()* and the *setNotBeDestroyedConfig()* methods check if unsaved data exists:

```
// Checking unsaved data.
onCanBeDestroyed: function(cacheKey) {
    // Receiving and checking the cache object.
    var config = this.Terrasoft.ClientPageSessionCache.getItem(cacheKey);
    if (!this.Ext.isObject(config)) {
```

```
            return;
        }
        var isChanged = this.isChanged();
        // If the edit page has unsaved data, a message is generated and the object is
changed in the cache.
        if (isChanged) {
            this.setNotBeDestroyedConfig(config);
        }
    }, ...
    // Forms a configuration object for displaying the message to the user.
    setNotBeDestroyedConfig: function(config) {
        // Determining the message shown to the user.
        var message = this.get("Resources.Strings.PageContainsUnsavedChanges");
        Ext.apply(config, {
            // Unsaved data attribute.
            canBeDestroyed: false,
            // The message shown to the user.
            errorInfo: {
                message: message
            }
        });
    }
```

Properties. The "properties" property



Introduction

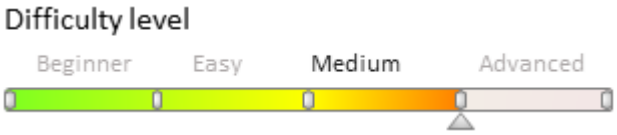
The "*properties*" property of the configuration object of the view model schema contains a JavaScript object that describes the properties of the view model.

An example of using the "properties" property in the *SectionTabsSchema* schema of the *NUI* package:

```
define("SectionTabsSchema", [],
        function() {
            return {
                ...
                // Declaring the "properties" property.
                properties: {
                    // The "parameters" property. Array.
                    parameters: [],
                    // The "modulesContainer" property. Object.
                    modulesContainer: {}
                },
                methods: {
                    ...
                    // Initialization method. Always executed first.
                    init: function(callback, scope) {
                        ...
                        // Calling a method that uses the properties of the view
model.
                        this.initContainers();
                        ...
                    },
                    ...
                    // A method that uses the properties.
                    initContainers: function() {
```

```
                        // Using the "modulesContainer" propery.
                        this.modulesContainer.items = [];
                        ...
                        // Usting the "parameters" property.
                        this.Terrasoft.each(this.parameters, function(config) {
                            config = this.applyConfigs(config);
                            var moduleConfig = this.getModuleContainerConfig(config);
                            var initConfig = this.getInitConfig();
                            var container =
viewGenerator.generatePartial(moduleConfig, initConfig)[0];
                            this.modulesContainer.items.push(container);
                        }, this);
                    },
                    ...
                },
            ...
            }
        });
```

Automatically generated view model properties



Introduction

In the bpm'online version 7.11.3 or higher for all attributes of the view model the properties are automatically generated with the “$” as prefix. Example:

```
//Traditional approach
var value = this.get("Attribute1"); /// Getting the value of the attribute.
this.set("Attribute1", 1) //Assigning the value to the attribute.

//Use of automatically generated properties.
this.$Attribute1; // Getting the value of the attribute.
this.$Attribute1 = 1 // Assigning the value to the attribute.
```

> ⚠ ATTENTION
>
> In the bpm'online version 7.11.3 such properties are not generated for the attributes that contain points in their names. For example, for the "Resources.Strings.TracingSaveException” attribute the automatically generated property will not be created.

Advantages of using automatically generated properties:

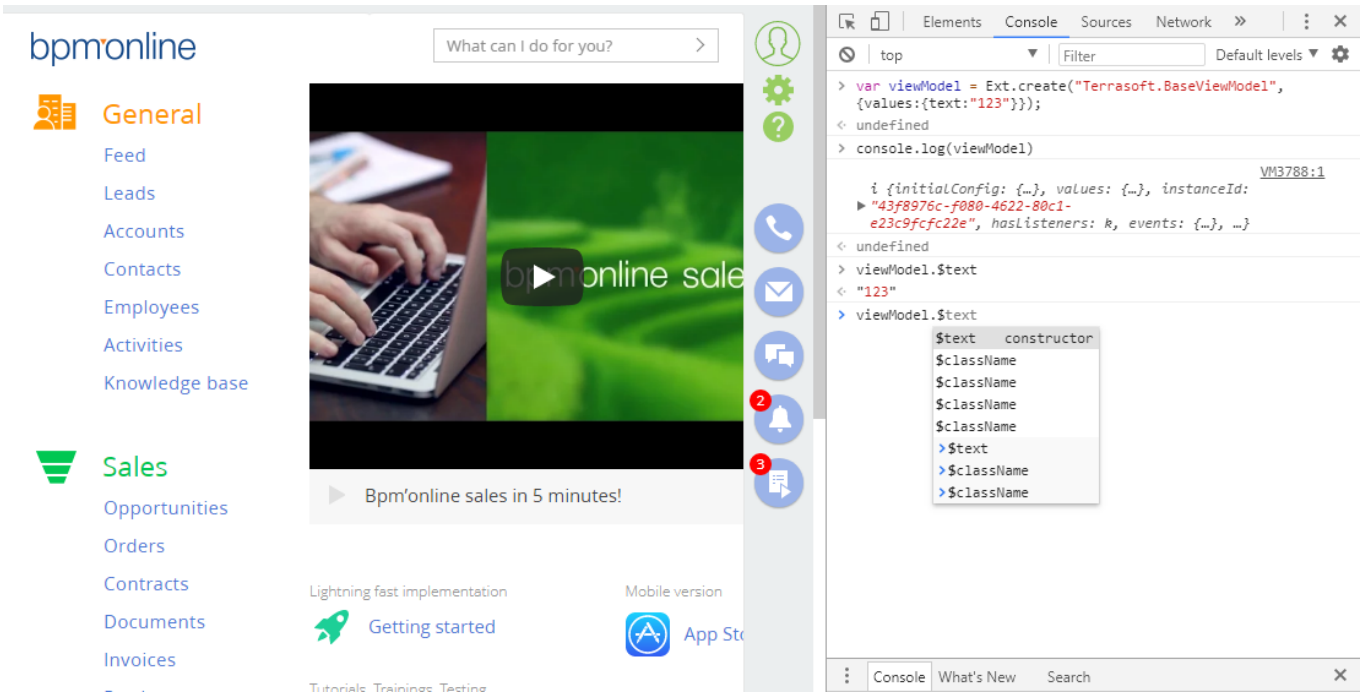
1. Reducing the amount of source code. No need to store attribute values in the variables, you can work with properties directly. For example, in the *ContactPageV2* view model schema, you can rewrite the *jobChanged()* method as follows:

```
//Traditional approach.
jobChanged: function() {
    var job = this.get("Job");
    var jobTitle = this.get("JobTitle");
    if (this.isNotEmpty(job) && this.isEmpty(jobTitle)) {
        this.set("JobTitle", job.displayValue);
    }
```

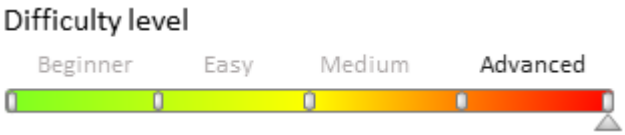
```
}

//Using auto-generated properties.
jobChanged: function() {
    if (this.isNotEmpty(this.$Job) && this.isEmpty(this.$JobTitle)) {
        this.$JobTitle = this.$Job.displayValue;
    }
}
```

2. Using the features of auto-tip (IntelliSense) in the browser console (Fig. 1).

Fig. 1. Using auto-tip for automatically generated properties



Sandbox. Module message exchange



Introduction

A bpm’online module is an isolated software unit. It has no information about other bpm’online modules apart from the module name list from which is depends. See “**Modular development principles in bpm'online**” for more information about bpm’online modules.

Sandbox object is used for interaction between the isolated modules. One of the key *sandbox* mechanisms is module message exchange.

Modules can only communicate via messages. A module shall publish a message to communicate its status change to other bpm’online modules. If the module needs to receive messages about status change in other modules, it must be subscribed to these messages.

To interact with other bpm’online modules, the module must import the *sandbox* module as a dependency.

> ⚠ NOTE
>
> It is not necessary to specify ["ext-base", "terrasoft", "sandbox"] base modules in dependencies if the module exports class constructor. *Ext*, *Terrasoft* and *sandbox* objects will be available as object properties after

creating module class object: *this.Ext*, *this.Terrasoft*, *this.sandbox*.

Message registration

You need to register messages to implement module message exchange.

> ⚠ NOTE
>
> Message registration is executed automatically if messages are declared in the *messages* module property.

sandbox.registerMessages(messageConfig) method is used to register module messages, where *messageConfig* is a module message configuration object.

Configuration object is a "key-value" collection, where every element is as follows:

```
"MessageName": {
    mode: [Message operation mode],
    direction: [Message direction]
}
```

"MessageName" is a collection element key that contains the message name. The value is a configuration object that contains the following properties:

- *mode* – message operation mode. Must contain *Terrasoft.MessageMode (Terrasoft.core.enums.MessageMode)* enumeration value.
- *direction* – message direction. Must contain *Terrasoft.MessageDirectionType (Terrasoft.core.enums.MessageDirectionType)* enumeration value.

Message exchange modes (*mode* property):

- Broadcast – message operation mode with a predefined number of subscribers. Corresponds to *Terrasoft.MessageMode.BROADCAST* enumeration value.
- Address – message operation mode when a message can only be processed by one subscriber. Corresponds to *Terrasoft.MessageMode.PTP*. enumeration value.

> ⚠ ATTENTION
>
> There can be several subscribers in the address mode, but only one can process messages, usually it is the last registered subscriber.

Message direction (*direction* property):

- Publication (publish) – the module can only publish a message in *sandbox*. Corresponds to *Terrasoft.MessageDirectionType.PUBLISH*. enumeration value.
- Subscription (follow) – the module can only subscribe to a message, published from another module. Corresponds to *Terrasoft.MessageDirectionType.SUBSCRIBE*. enumeration value.
- Bidirectional – allows to publish and subscribe to the same message in different instances of the same class or within the same schema inheritance hierarchy. Corresponds to *Terrasoft.MessageDirectionType.BIDIRECTIONAL*. enumeration value.

Module message registration:

```
// Message configuration object collection.
var messages = {
    "MessageToSubscribe": {
        mode: Terrasoft.MessageMode.PTP,
        direction: Terrasoft.MessageDirectionType.SUBSCRIBE
    },
    "MessageToPublish": {
        mode: Terrasoft.MessageMode.BROADCAST,
        direction: Terrasoft.MessageDirectionType.PUBLISH
    }
};
```

```
// Message registration.
this.sandbox.registerMessages(messages);
```

> ATTENTION
>
> It is not necessary to register messages via the *sandbox.registerMessages()* method in the view model schemas. Declare the message configuration object in *messages* property (see "**Messages. The "messages" property**").

To reject message registration in a module, use *sandbox.unRegisterMessages(messages)* method, where *messages* – is a message name or a message name array. Message registration rejection:

```
// Single message registration rejection.
this.sandbox.unRegisterMessages("MessageToSubscribe");
// Message array registration rejection.
this.sandbox.unRegisterMessages(["MessageToSubscribe", "MessageToPublish"]);
```

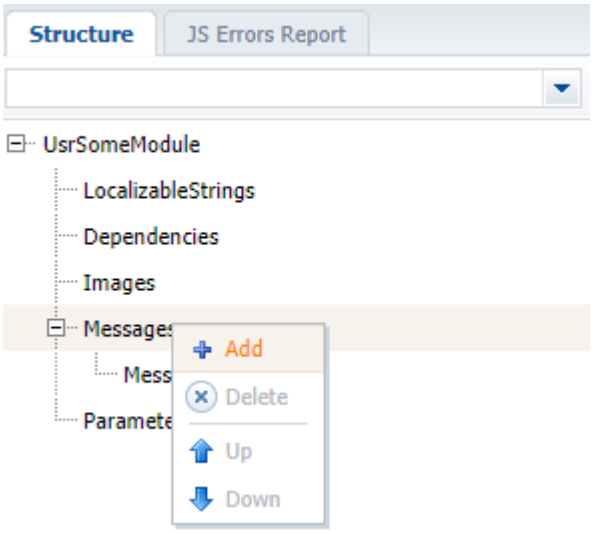
Adding messages to module schema

You can also register messages by adding them to a module schema or via a designer (see “**Module designer**”).

To add messages to module schema:

1. On the [Structure] tab of the module schema designer select the [Messages] node, right-click and execute the [Add] command (Fig.1).

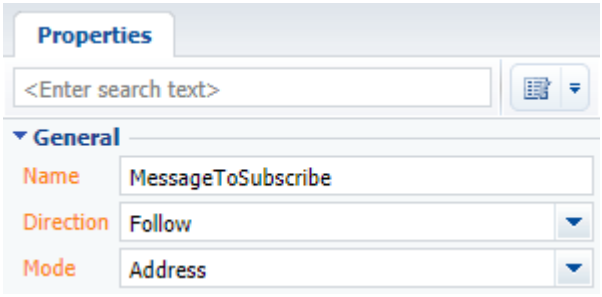
Fig. 1. Adding messages to the module schema structure



2. Set the necessary properties for the added message (Fig.2):

- [Name] – the message name that corresponds to the module configuration object key.
- [Direction] – message direction. Possible values: “Follow” (subscribe) and “Publish” (publish).
- [Mode] – message operation mode. Possible values: “Broadcast” and “Address”.

Fig. 2. Message properties



ATTENTION

It is not necessary to add messages to schema structure in **view model schemas** .

Message publication

sandbox.publish(messageName , messageArgs, tags) method is used to publish messages.

Method parameters:

- *messageName* – the string that contains the message name, for instance, "MessageToSubscribe".
- *messageArgs* – the object, passed as an argument to the message handler method in the subscription module. If there are no input parameters in the handler method, assign null value to messageArgs parameter.
- *tags* – tag array that allows to uniquely identify the message sending module. Usually, the *[this.sandbox.id]* value is used. *Sandbox* defines the message subscribers and publishers according to the tag array.

NOTE

Only the handlers that meet at least one tag will be run for the message published with the tag array. Messages, published without tags, will only be processed by subscribers without tags.

Message publication method:

```
// Message publication without tags or argument.
this.sandbox.publish("MessageWithoutArgsAndTags");
// Message publication without a handler method argument.
this.sandbox.publish("MessageWithoutArgs", null, [this.sandbox.id]);
// Message publication with a handler method argument.
this.sandbox.publish("MessageWithArgs", {arg1: 5, arg2: "arg2"}, ["moduleName"]);
// Message publication with an arbitrary tag array.
this.sandbox.publish("MessageWithCustomIds", null, ["moduleName","otherTag"]);
```

When you publish a message in the address mode, you can receive the result of its processing by the subscriber. To do this, the message handler method in the subscription module must return the corresponding result (see "Message subscription"). Message publication:

```
// Message declaring and registration.
var messages = {
    "MessageWithResult": {
        mode: Terrasoft.MessageMode.PTP,
        direction: Terrasoft.MessageDirectionType.PUBLISH
    }
};
this.sandbox.registerMessages(messages);
// Message publication and receipt of the result of its processing by the
subscription module.
var result = this.sandbox.publish("MessageWithResult", {arg1:5, arg2:"arg2"},
["resultTag"]);
// Result display on the browser console.
console.log(result);
```

When you publish a message in the broadcast mode, you can receive the result of its processing via the object, passed as an argument to the handler method.

```
//  Message declaring and registration.
var messages = {
    "MessageWithResult": {
        mode: Terrasoft.MessageMode.BROADCAST,
        direction: Terrasoft.MessageDirectionType.PUBLISH
    }
```

```
};
this.sandbox.registerMessages(messages);
var arg = {};
// Message publication and receipt of the result of its processing by the
subscription module.
// Add result property into the object of the subscription module handler method and
populate it with the processing result.
this.sandbox.publish("MessageWithResult", arg, ["resultTag"]);
// Result display on the browser console.
console.log(arg.result);
```

Message subscription

You can subscribe to a message using the *sandbox.subscribe(messageName, messageHandler, scope, tags)* method.

Method parameters:

- *messageName* – the string that contains the message name, for instance, "MessageToSubscribe".
- *messageHandler* – the handler method, run upon the message receipt. It can be either an anonymous function or a module method. A parameter, whose value must be passed upon the message publishing via the *sandbox.publish()* method can be indicated in the method definition.
- *Scope* – *messageHandler* handler method execution context.
- *tags* – tag array that allows to uniquely identify the message sending module. *Sandbox* defines the message subscribers and publishers according to the tag array.

Message subscription method:

```
// Message subscription without handler method arguments.
// Handler method is an anonymous function. Execution context is the current module.
//The getsandboxid() method must return the tag that corresponds to the published
message tag.
this.sandbox.subscribe("MessageWithoutArgs", function(){console.log("Message without
arguments")}, this, [this.getSandBoxId()]);
// Message subscription with a handler method argument.
this.sandbox.subscribe("MessageWithArgs", function(args){console.log(args)}, this,
["moduleName"]);
// Message subscription with an arbitrary tag.
// It can be any tag out of the published message tag array.
// The myMsgHandler handler method must be implemented separately.
this.sandbox.subscribe("MessageWithCustomIds", this.myMsgHandler, this,
["otherTag"]);
```

The message handler method must return the corresponding result for a message in the address mode. Message subscription:

```
// Message declaring and registration.
var messages = {
    "MessageWithResult": {
        mode: Terrasoft.MessageMode.PTP,
        direction: Terrasoft.MessageDirectionType.SUBSCRIBE
    }
};
this.sandbox.registerMessages(messages);
// Message subscription.
this.sandbox.subscribe("MessageWithResult", this.onMessageSubscribe, this,
["resultTag"]);
...
// The handler method is implemented in the subscription module.
// args — object, passed upon message publication.
onMessageSubscribe: function(args) {
    // Parameter change.
    args.arg1 = 15;
```

```
    args.arg2 = "new arg2";
    // Obligatory return of result.
    return args;
},
```

Asynchronous message exchange

Use callback function approach if the message handler method in subscription module generates the result asynchronously.

Message publication and result:

```
// Message publication without tags or argument.
this.sandbox.publish("MessageWithoutArgsAndTags");
// Message publication without a handler method argument.
this.sandbox.publish("MessageWithoutArgs", null, [this.sandbox.id]);
// Message publication with a handler method argument.
this.sandbox.publish("MessageWithArgs", {arg1: 5, arg2: "arg2"}, ["moduleName"]);
// Message publication with an arbitrary tag array.
this.sandbox.publish("MessageWithCustomIds", null, ["moduleName","otherTag"]);
```

Message subscription:

```
// Message subscription without handler method arguments.
// Handler method is an anonymous function. Execution context is the current module.
//The getsandboxid() method must return the tag that corresponds to the published
message tag.
this.sandbox.subscribe("MessageWithoutArgs", function(){console.log("Message without
arguments")}, this, [this.getSandBoxId()]);
// Message subscription with a handler method argument.
this.sandbox.subscribe("MessageWithArgs", function(args){console.log(args)}, this,
["moduleName"]);
// Message subscription with an arbitrary tag.
// It can be any tag out of the published message tag array.
// The myMsgHandler handler method must be implemented separately.
this.sandbox.subscribe("MessageWithCustomIds", this.myMsgHandler, this,
["otherTag"]);
```

Module with a message:

Below is a module with message publication and subscription:

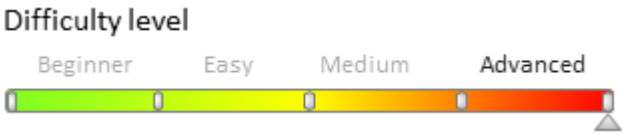
```
define("UsrSomeModule", [], function() {
    Ext.define("Terrasoft.configuration.UsrSomeModule", {
        alternateClassName: "Terrasoft.UsrSomeModule",
        extend: "Terrasoft.BaseModule",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
        messages: {
            "MessageToSubscribe": {
                mode: Terrasoft.MessageMode.PTP,
                direction: Terrasoft.MessageDirectionType.SUBSCRIBE
            },
            "MessageToPublish": {
                mode: Terrasoft.MessageMode.BROADCAST,
                direction: Terrasoft.MessageDirectionType.PUBLISH
            }
        },
        init: function() {
            this.callParent(arguments);
            this.sandbox.registerMessages(this.messages);
```

```
            this.processMessages();
        },
        processMessages: function() {
            this.sandbox.subscribe("MessageToSubscribe", this.onMessageSubscribe,
this);
            this.sandbox.publish("MessageToPublish", null, [this.sandbox.id]);
        },
        onMessageSubscribe: function() {
            console.log("'MessageToSubscribe' received");
        },
        destroy: function() {
            if (this.messages) {
                var messages = this.Terrasoft.keys(this.messages);
                this.sandbox.unRegisterMessages(messages);
            }
            this.callParent(arguments);
        }
    });
    return Terrasoft.UsrSomeModule;
});
```

For more information

- **Modular development principles in bpm'online**
- **Client Modules**
- **Client view model schemas**
- **Messages. The "messages" property**
- **Sandbox. Bidirectional messages**

Sandbox. Bidirectional messages



Introduction

One of the key *sandbox* mechanisms is module message exchange (see “**Sandbox. Module message exchange**”).

It often becomes necessary to publish and subscribe to the same message in different instances of the same class (module) or within the same schema inheritance hierarchy. To perform this6 the *sandbox* object has bidirectional messages that correspond to the value of the *Terrasoft.MessageDirectionType.BIDIRECTIONAL* enumeration.

Registration of bidirectional messages

To register bidirectional messages in the *messages* property of the schema, use the following confrontation object:

```
messages: {
    "MessageName": {
        mode: [Режим работы сообщения],
        direction: Terrasoft.MessageDirectionType.BIDIRECTIONAL
    }
}
```

The purpose and possible values of the elements of configuration object used in message registration are described in the "**Sandbox. Module message exchange**” article.

Use case

The following case demonstrates how bidirectional messages work.

The *CardModuleResponse* message is registered in the *BaseEntityPage* schema, which is a base schema for all view model schemas of the record edit pages.

```
define("BaseEntityPage", [...], function(...) {
    return {
        messages: {
            ...
            "CardModuleResponse": {
                "mode": this.Terrasoft.MessageMode.PTP,
                "direction": this.Terrasoft.MessageDirectionType.BIDIRECTIONAL
            },
            ...
        },
        ...
    };
});
```

For example, the message is published after saving the modified record.

```
define("BasePageV2", [..., "LookupQuickAddMixin", ...],
    function(...) {
        return {
            ...
            methods: {
                ...
                onSaved: function(response, config) {
                    ...
                        this.sendSaveCardModuleResponse(response.success);
                    ...
                },
                ...
                sendSaveCardModuleResponse: function(success) {
                    var primaryColumnValue = this.getPrimaryColumnValue();
                    var infoObject = {
                        action: this.get("Operation"),
                        success: success,
                        primaryColumnValue: primaryColumnValue,
                        uId: primaryColumnValue,
                        primaryDisplayColumnValue:
this.get(this.primaryDisplayColumnName),
                        primaryDisplayColumnName: this.primaryDisplayColumnName,
                        isInChain: this.get("IsInChain")
                    };
                    return this.sandbox.publish("CardModuleResponse", infoObject,
[this.sandbox.id]);
                },
                ...
            },
            ...
        };
    });
```

This functionality is implemented in the *BasePageV2* child schema (i.e. the *BaseEntityPage* schema is parental for the *BasePageV2* schema). Also, the *LookupQuickAddMixin* mixin is specified as a dependency in the *BasePageV2*. The subscription for the *CardModuleResponse* message is performed in this mixin.

> **NOTE**
>
> *A mixin* is a class designed to extend the functions of other classes. Mixins expand the functionality of

schemas, allowing to avoid duplication of commonly used logic in schema methods. Mixins are different from other modules added to the dependency list in a way that their methods can be addressed directly, much like those of a schema (see “**Mixins. The "mixins" property**”).

```
define("LookupQuickAddMixin", [...],
        function(...) {
            Ext.define("Terrasoft.configuration.mixins.LookupQuickAddMixin", {
                alternateClassName: "Terrasoft.LookupQuickAddMixin",
                ...
                // Declaration of the message.
                _defaultMessages: {
                    "CardModuleResponse": {
                        "mode": this.Terrasoft.MessageMode.PTP,
                        "direction":
this.Terrasoft.MessageDirectionType.BIDIRECTIONAL
                    }
                },
                ...
                // Message registration method.
                _registerMessages: function() {
                    this.sandbox.registerMessages(this._defaultMessages);
                },
                ...
                // Initializing an instance of a class.
                init: function(callback, scope) {
                    ...
                    this._registerMessages();
                    ...
                },
                ...
                // Performed after adding a new record to the lookup.
                onLookupChange: function(newValue, columnName) {
                    ...
                    // Here, the method call chain is executed.
                    // As a result, the _subscribeNewEntityCardModuleResponse ()
method will be called.
                    ...
                },
                ...
                // The method in which the subscription to the "CardModuleResponse"
message is performed.
                // In the reference field, the adding of the value sent when the
message was published
                // is executed in the callback function.
                                _subscribeNewEntityCardModuleResponse:
function(columnName, config) {
                    this.sandbox.subscribe("CardModuleResponse", function(createdObj)
{
                        var rows = this._getResponseRowsConfig(createdObj);
                        this.onLookupResult({
                            columnName: columnName,
                            selectedRows: rows
                        });
                    }, this, [config.moduleId]);
                },
                ...
            });
            return Terrasoft.LookupQuickAddMixin;
        });
```

Adding a new address on the contact edit page is a good example of how bidirectional messages work.

1. After executing the command of adding a new record on the [Addresses] detail (Fig. When will the timeline be finished? 1), the *ContactAddressPageV2* module is loaded to the module chain and the edit page of the contact address opens (Fig. 2).

Fig. 1. Adding a new record on the [Addresses] detail

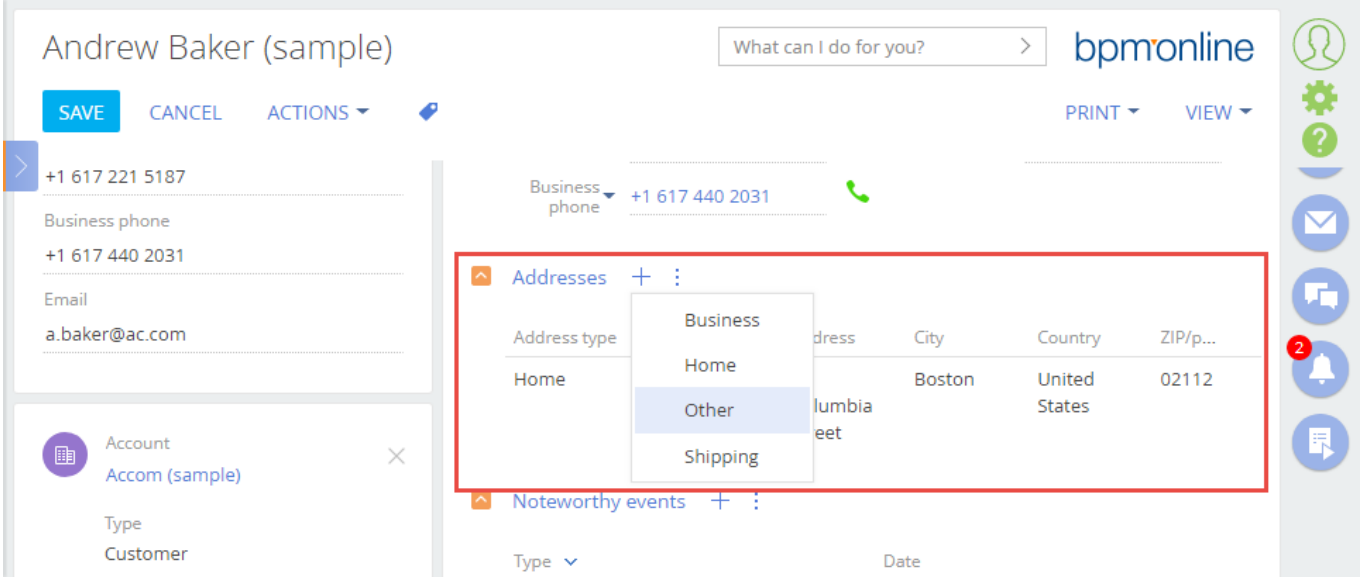
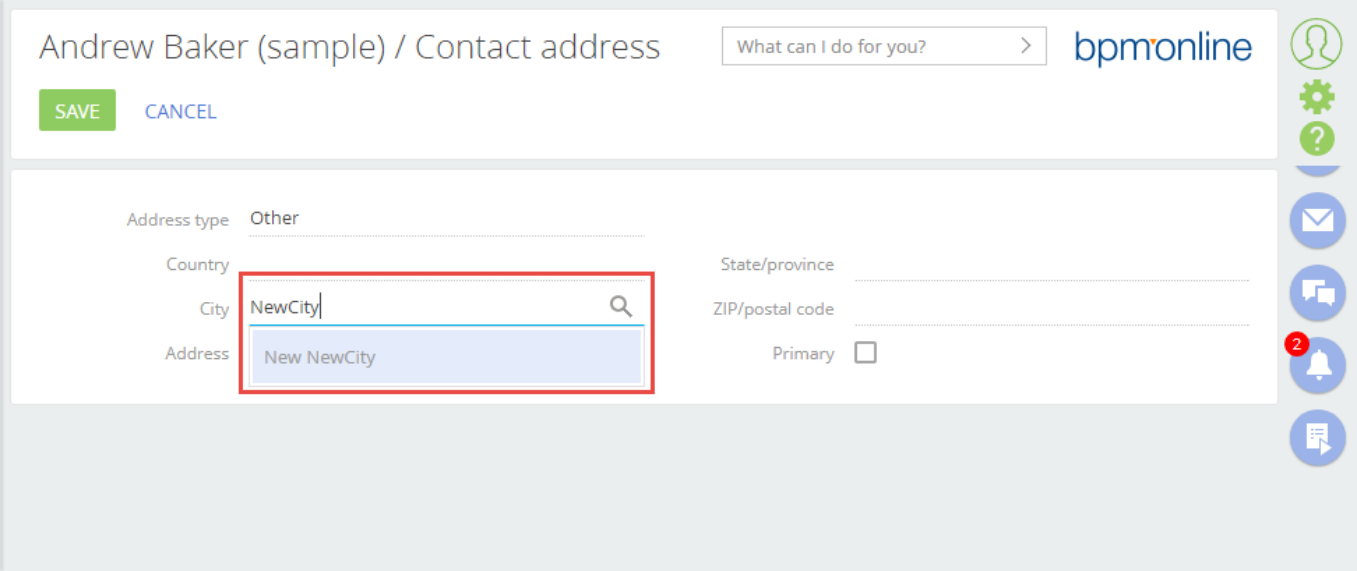


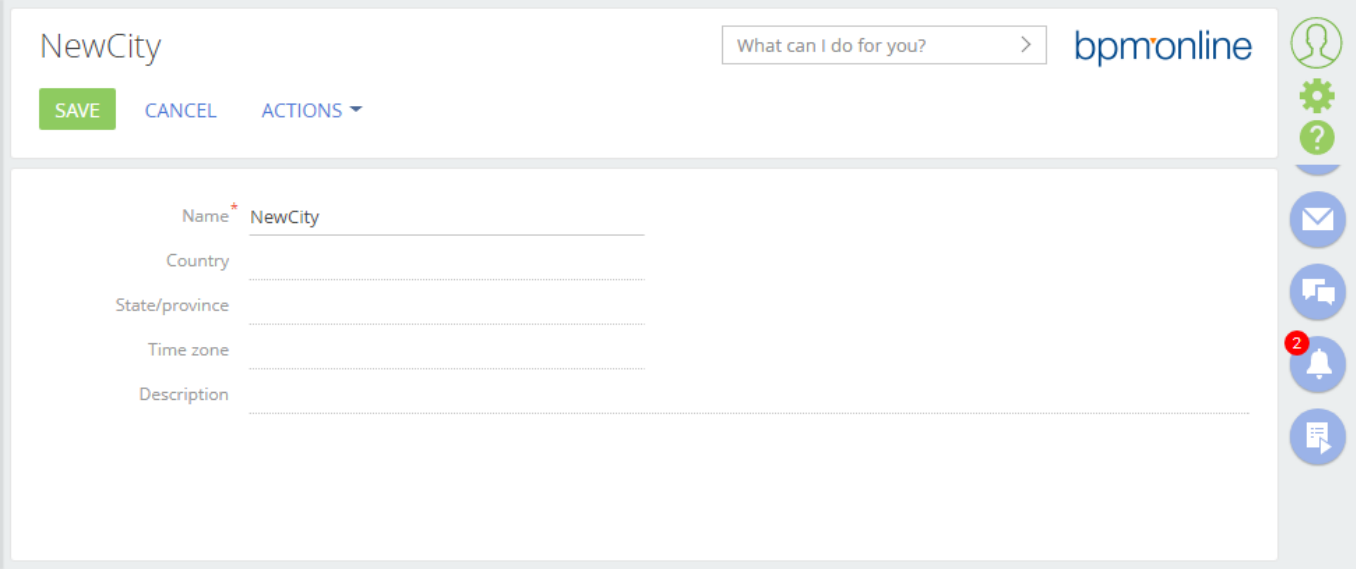
Fig. 2. Contact address edit page



The *CardModuleResponse* message has been already registered in the *ContactAddressPageV2* schema as it has the *BaseEntityPage* and the *BasePageV2* schemas in the inheritance hierarchy. This message is also registered in the*_registerMessages()* method of the *LookupQuickAddMixin* mixin at its initialization as the dependency module of the *BasePageV2*.

2. When adding the new value to the lookup fields of the *ContactAddressPageV2* page (for example, a new city (Fig.2)) the *onLookupChange()* method of the *LookupQuickAddMixin* mixin is called. In this method, in addition to loading the *CityPageV2* module to the module chain, the*_subscribeNewEntityCardModuleResponse()* methodis called, in which the subscription for the *CardModuleResponse* message is performed. After this, the city edit page is opened (the *CityPageV2*, Fig. 3).

Fig. 3. City edit page



3. As the *CityPageV2* schema also has the *BasePageV2* schema in the inheritance hierarchy, the *onSaved()* method implemented in the base schema will be executed after saving the record (the [Save] button, Fig. 3). This method calls the *sendSaveCardModuleResponse()* method where the message (*CardModuleResponse*) is published. At the same time, the object with the necessary saving results is passed.

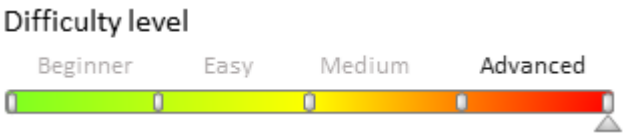
4. The execution of the *callback* subscriber function that process results of saving a new city in the lookup starts after publication of the message (see the *_subscribeNewEntityCardModuleResponse()* method of the *LookupQuickAddMixin* mixin).

The publication and subscription for the bidirectional message was performed in one inheritance hierarchy of the schemas with the *BasePageV2* base schema that contains all necessary functions.

See also

- **Sandbox. Module message exchange**
- **Messages. The "messages" property**

Sandbox. Loading and unloading modules



Introduction

A bpm’online module is an isolated software unit. It has no information about other bpm’online modules apart from the module name list from which is depends. See “**Modular development principles in bpm'online**” for more information about bpm’online modules.

In certain situations, you might need to load modules that were not declared as dependencies when working with bpm’onlie interface. To load and unload such modules, *sandbox.loadModule()* and *sandbox.unloadModule()* methods are designed.

Loading modules

Use the *sandbox.loadModule(moduleName, config)* method to load undeclared modules. Method parameters:

- *moduleName* – module name.
- *config* – configuration object that contains module parameters. This is a required parameter for visual modules.

Method *sandbox.loadModule()* call parameters:

```
// Loading a module without additional parameters.
this.sandbox.loadModule("ProcessListenerV2");
// Loading a module with additional parameters.
this.sandbox.loadModule("CardModuleV2", {
    renderTo: "centerPanel",
    keepAlive: true,
    id: moduleId
});
```

Module parameters

Additional module loading parameters are aggregated in the *config* configuration object. Most common properties of this object are as follows:

- *id* – module Id. If the Id is not specified, it will be generated automatically. Data type – string.
- *renderTo* – name of the container where visual module view will be displayed. Passed as an argument to the *render()* method of the loaded module. Required for visual modules. Data type – string.
- *keepAlive* – indicates whether the module is added to a module thread. Used for navigation between the module views. Data type – Boolean.
- *isAsync* – indicates asynchronous initialization of the module (see "**Modular development principles in bpm'online**"). Data type – Boolean.

Module class constructor parameters. The instanceConfig property.

There is an option to pass arguments to the class constructor of the instantiated module. To do this, add the *instanceConfig* property to the *config* configuration object and assign an object with the needed values to it.

> ⚠ NOTE
>
> The instantiated module is the one returning the a constructor function.

For example, the following module is declared:

```
// A module that returns a class instance.
define("CardModuleV2", [...], function(...) {
    // A class used for creating an edit page module.
    Ext.define("Terrasoft.configuration.CardModule", {
        // Class alias.
        alternateClassName: "Terrasoft.CardModule",
        // Parent class.
        extend: "Terrasoft.BaseSchemaModule",
        // Indicates that schema parameters have been set from without.
        isSchemaConfigInitialized: false,
        // Indicates that history state is used on module loading.
        useHistoryState: true,
        // Schema name of the displayed entity.
        schemaName: "",
        // Indicates that the common display mode (with the section list) is used.
        // If the value is false, SectionModule is present on the page.
        isSeparateMode: true,
        // Object schema name.
        entitySchemaName: "",
        // Primary column value.
        primaryColumnValue: Terrasoft.GUID_EMPTY,
        // Edit page mode. Possible values
        // ConfigurationEnums.CardStateV2.ADD|EDIT|COPY
        operation: ""
    });
    // Return class instance.
    return Terrasoft.CardModule;
```

```
}
```

Use the following code to pass the needed values to the module constructor on its loading:

```
// Object, whose properties contain values
// passed as constructor parameters
var configObj = {
    isSchemaConfigInitialized: true,
    useHistoryState: false,
    isSeparateMode: true,
    schemaName: "QueueItemEditPage",
    entitySchemaName: "QueueItem",
    operation: ConfigurationEnums.CardStateV2.EDIT,
    primaryColumnValue: "{3B58C589-28C1-4937-B681-2D40B312FBB6}"
};

// Loading module.
this.sandbox.loadModule("CardModuleV2", {
    renderTo: "DelayExecutionModuleContainer",
    id: this.getQueueItemEditModuleId(),
    keepAlive: true,
    // Specifying values passed to module constructor.
    instanceConfig: configObj
    }
});
```

As a result, the module will be loaded with pre-set property values and no additional messages will be needed to set them.

ATTENTION

The following types of properties can be passed to module instance:

- *string*;
- *boolean*;
- *number*;
- *date* (the value will be copied);
- *object* (Literal objects only. You cannot pass class instances, HTMLElement inheritors, etc.).

ATTENTION

When passing parameters to module constructor of a Terrasoft.BaseObject inheritor, the following limitations apply: if a parameter is not described in the module class or one of parent classes, it cannot be passed.

Additional module parameters. The "parameters" property.

The *parameters* property is designed to pass additional parameters to a module on its loading in the *config* configuration object. Same property must be defined in the module class (or one of its parent classes) as well.

NOTE

The *parameters* property is already defined in the *Terrasoft.BaseModule* class.

Thus, when instantiating the module, its *parameters* property will be initialized with values passed in the *parameters* property of the *config* object.

For example, the "parameters" property is defined in the *MiniPageModule* module:

```
define("MiniPageModule", ["BusinessRulesApplierV2", "BaseSchemaModuleV2",
"MiniPageViewGenerator"],
    function(BusinessRulesApplier) {
        Ext.define("Terrasoft.configuration.MiniPageModule", {
```

```
            extend: "Terrasoft.BaseSchemaModule",
            alternateClassName: "Terrasoft.MiniPageModule",
            ...
            parameters: null,
            ...
        });

        return Terrasoft.MiniPageModule;
    });
```

In this case, the pop-up summary module can be instantiated with additional parameters. Example:

```
define("MiniPageContainerViewModel", ["ConfigurationEnumsV2"], function() {
    Ext.define("Terrasoft.configuration.MiniPageContainerViewModel", {
        extend: "Terrasoft.BaseModel",
        alternateClassName: "Terrasoft.MiniPageContainerViewModel",
        ...
        loadModule: function() {
            // The "parameters" property is defined in the parent class
Terrasoft.BaseModel
            var parameters = this.get("parameters");
            ...
            this.sandbox.loadModule("MiniPageModule", {
                renderTo: Ext.get("MiniPageContainer"),
                id: moduleId,
                // Passing parameters to configuration object.
                parameters: parameters
            });
        }
        ...
    });
});
```

Unloading modules

Use the *sandbox.unloadModule(id, renderTo, keepAlive)* method to unload module. Method parameters:

- *id* – module Id. Data type – string.
- *renderTo* – name of the container where visual module view will be deleted. Required for visual modules. Data type – string.
- *keepAlive* – indicates if the module model is saved. On unloading the module, the core can save its model for using its properties, methods an messages. Data type – Boolean. Not recommended.

Method *sandbox.unloadModule()* call parameters:

```
...
// Method that obtains Id of unloaded module.
getModuleId: function() {
    return this.sandbox.id + "_ModuleName";
},
...
// Unloading a non-visual module.
this.sandbox.unloadModule(this.getModuleId());
...
// Unloading a visual module, previously loaded to the "ModuleContainer" container.
this.sandbox.unloadModule(this.getModuleId(), "ModuleContainer");
```

Module thread

Sometimes a model view must be shown in place of other model. For example, to set a field value on the current page, a *SelectData* lookup selection page must be displayed. In such cases, the current page module must not be unloaded, but the module view of the lookup selection page must be displayed in place of its container. For this, use

module threads:

To begin a new module thread, add the *keepAlive* property to the configuration object of the loaded module. For example, you need to display lookup selection module (*selectDataModule*) in the current page module (*CardModule*). To do this, execute the following code:

```
sandbox.loadModule("selectDataModule", {
    // Id of the loaded module view.
    id: "selectDataModule_id",
    // The view will be added to the current page container.
    renderTo: "cardModuleContainer",
    // Denies unloading of the current module.
    keepAlive: true
});
```

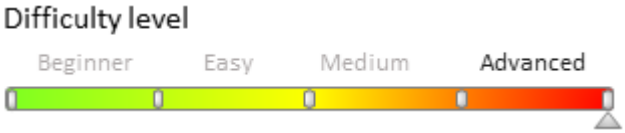
After the code is executed, a module thread consisting of the current page module and lookup selection module will be created. Clicking the [Add new record] button in the current page module (*selectData*) will open a new page and add another element to the thread. Thus, unlimited number of modules can be added to the module thread. Active module (the one displayed on the page) will always be the last element in the thread. If an element in a thread is set as active, all elements after it will be destroyed. To activate a chain element, call the *loadModule* function and pass module Id to its parameter:

```
sandbox.loadModule("someModule", {
    id: "someModuleId"
});
```

The core will destroy all elements in the thread after the specified one and execute standard module loading logic, calling the *init()* and *render()* methods. The container where the previous active module was placed will be passed to the *render()* method. All modules in the thread can work as before, receiving and sending messages, saving data, etc.

If the *keepAlive* property is not added to the configuration object when the *loadModule()* method is called, or if this property is added as *keepAlive: false*, the module thread will be destroyed.

New bindTo format at setting connection between view and viewModel



Introduction

You can use the *bindTo* property of the *Ext.create()* construction function configuration object to indicate the connection between a view model attribute and a view value when creating the *Terrasoft* namespace components. Example:

```
Ext.create("Terrasoft.BaseEdit", {
   value: {
      bindTo: "Value"
   }
});
```

Starting from version 7.12.0 you already have an opportunity to indicate this connection in a new format.

```
Ext.create("Terrasoft.BaseEdit", {
   value: "$Value"
});
```

> ⚠ ATTENTION
>
> The previous format also remains available for usage.

NOTE

The new format became possible due to usage of automatically generated properties (see "**Automatically generated view model properties**").

Usage of the new bindTo format in the diff array

In the diff array configuration object values property of the view model schemas the new *bindTo* format is implemented for tab titles (for objects, whose “propertyName” is populated with the “tabs” value).

```
...
{
    "operation": "insert",
    "name": "GeneralInfoTab",
    "parentName": "Tabs",
    "propertyName": "tabs",
    "index": 0,
    "values": {
        "caption": "$Resources.Strings.GeneralInfoTabCaption",
        "items": []
    }
},
...
```

For the rest of elements the previous format is valid.

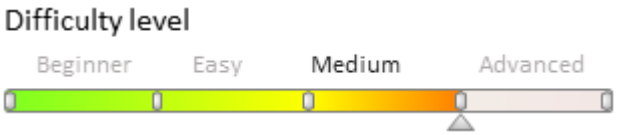
```
...
{
    "operation": "insert",
    "parentName": "ProfileContainer",
    "propertyName": "items",
    "name": "JobTitleProfile",
    "values": {
        "bindTo": "JobTitle",
        "layout": {...}
   }
},
...
```

Controls

Contents

- **Controls. Introduction**
- **Details**
- **The [Connected entity profile] control**
- **SourceCodeEditMixin class description and work examples.**
- **Blocking edit page fields**

Controls. Introduction



Controls are objects used to create an interface between the user and a bpm’online application. They are displayed in navigation panels, dialog boxes and toolbars. Controls include buttons, checkboxes, radio buttons, input fields etc.

All controls are inherited from the *Terrasoft.Component* class (*Terrasoft.controls.Component* in full), which in turn is inherited from *Terrasoft.BaseObject*. Because the *Bindable* mixin is declared in the (*Component*) parent class, it is possible to bind control properties to desired view model properties, methods, or attributes.

For correct operation, the necessary events are declared in the control element. Each element contains events inherited from the *Terrasoft.Component* class:

- *added* – triggered after the component is added to the container.
- *afterrender* – triggered after the component has been rendered and its HTML representation gets in the DOM.
- *afterrerender* – triggered after the component has been rendered and its HTML representation is updated in the DOM.
- *beforererender* – triggered before the component has been rendered and its HTML representation gets in the DOM.
- *destroy* – triggered before the control is deleted.
- *destroyed* – triggered after the control is deleted.
- *init* – triggered when component initialization is complete.

Learn more about *Terrasoft.Component* class events in the "**('JavaScript API for platform core' in the online documentation)**" article.

Controls may subscribe to browser events and determine their own events.

The control is defined in the *diff* modification array of the module where it must be located.

```
// The diff modification array of the module
diff: [{
    // Insert operation.
    "operation": "insert",
    // Control parent element.
    "parentName": "CardContentContainer",
    // Name of the parent element property with which the operation is performed.
    "propertyName": "items",
    // Control name.
    "name": "ExampleButton",
    // Control value.
    "values": {
        // Control type.
        "itemType": "Terrasoft.ViewItemType.BUTTON",
        // Control caption.
        "caption": "ExampleButton",
        // Binding the control event to a function.
        "click": {"bindTo": "onExampleButtonClick"},
        // Control style.
        "style": Terrasoft.controls.ButtonEnums.style.GREEN
    }
}]
```

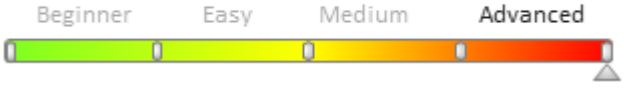
The appearance of the control is determined by the (*template <tpl>*) template. Element view is generated according to a specified template during the rendering of the control in the page view.

Controls do not have any business logic. The logic is determined in the module where the control is added.

The control has the *styles* and *selectors* attributes, which are determined in the *Terrasoft.Component* parent class. These attributes provide style customization flexibility.

Details

Difficulty level



Introduction

Details are used to display supplemental data for a primary section object. The section details are displayed on the section edit page tabs in the tabs area.

Depending on the method of entering and displaying data, there are following types of details.

- Details with edit fields.
- Details with adding page.
- Details with editable list.
- Details with selection from lookup.

More information about details of different types can be found in the "**Details**” article.

The detail creation

A custom detail must be registered so that the detail wizard could work with it. To register a detail, add a record with detail caption, detail schema identifier *DetailSchemaUid* (from the *UId* column in the *SysSchema* table) and detail object schema identifier *EntitySchemaUId* (from the *UId* column in the *SysSchema* table) to the *SysDetail* table.

More information about creating details of different types can be found in the "**Adding details**” article.

The base schema of the BaseDetailV2 detail

All detail schemas must be inherited from the *BaseDetailV2* base schema. The base logic of data initialization and communication with the edit page are implemented in the schema.

The base schema class has the following properties:

BaseDetailV2 messages

The massages are used for communication between the detail and the edit page. A full list of messages, their broadcast mode and direction are given in the table 1.

Table 1. The messages of the base detail

Name	Mode	Direction	Description
GetCardState	Address	Publication	Returns a state of the edit page.
SaveRecord	Address	Publication	Tells the edit page to save the data.
DetailChanged	Address	Publication	Tells the edit page about the changes of the detail data.
UpdateDetail	Address	Subscription	A subscription to the edit page update.
OpenCard	Address	Publication	Opens edit page.

The message modes are defined in the *Terrasoft.core.enums.MessageMode* enumeration and message direction is defined in the *Terrasoft.core.enums.MessageDirectionType*. More information about them can be found in the “**('JavaScript API for platform core' in the on-line documentation)**” article.

BaseDetailV2 attributes

The *attributes* property contains the attributes of detail view model. The attributes that are defined in base detail class are given in the table 2.

Table 2. The attributes of the base detail

Name	Type	Description
CanAdd	BOOLEAN	Indicates the possibility to add data.
CanEdit	BOOLEAN	Indicates the possibility to edit data.
CanDelete	BOOLEAN	Indicates the possibility to delete data.

Collection	COLLECTION	Detail data collection.
Filter	CUSTOM_OBJECT	Detail filter. Used for filtering detail data.
DetailColumnName	STRING	The column name where the filtering is performed.
MasterRecordId	GUID	The key value of the parent record.
IsDetailCollapsed	BOOLEAN	Indicates if the detail is collapsed.
DefaultValues	CUSTOM_OBJECT	The default value of the model columns.
Caption	STRING	The detail caption.

The available attribute data types are represented by the *Terrasoft.DataValueType* enumeration. More information about them can be found in the “**('JavaScript API for platform core' in the on-line documentation)**” article.

BaseDetailV2 methods

The methods defined in base detail class are given in the table 3.

Table 3. The methods of the base detail

Name	Parameters	Description
init	*{Function} callback* – callback function. *{Object} scope* – the context of the method execution.	Initializes the detail page.
initProfile	No	Initializes the schema profile. Default value is *Terrasoft.emptyFn*
initDefaultCaption	No	Sets the default caption of the detail.
initDetailOptions	No	Initializes the list view data collection.
subscribeSandboxEvents	No	It is subscribed to the messages necessary for the work of the detail.
getUpdateDetailSandboxTags	No	Generates the array of tags for the *UpdateDetail* message.
UpdateDetail	*{Object} config* – configuration object that contains the properties of the detail.	Updates the detail according to the parameters passed. Default value is *Terrasoft.emptyFn*
initData	*{Function} callback* – callback function. *{Object} scope* – the context of the method execution.	Initializes the list view data collection.
getEditPageName	No	Returns the name of the edit page depending on the record type at editing or on selected record type for adding.
onDetailCollapsedChanged	*{Boolean} isCollapsed* – the attribute of the collapsed/expanded detail.	The handler of collapsing or expanding of the detail.
getToolsVisible	No	Returns the collapse value of the detail.
getDetailInfo	No	Publishes a message to get information about the detail.

BaseDetailV2 array of modifications

In the *diff* modifications array of the base detail, only a base container for detail view is defined:

```
diff: /**SCHEMA_DIFF*/[
  // Base container for detail view.
  {
     "operation": "insert",
     "name": "Detail",
     "values": {...}
  }
]/**SCHEMA_DIFF*/
```

The “BaseGridDetailV2” base “detail with list” class

The class is a *BaseDetailV2* inheritor. All details with lists must be the *BaseGridDetailV2* inheritors. The list base logic (import, filtering, adding, deleting and editing the detail records) is implemented in the *BaseGridDetailV2* schema.

More information about creating custom details can be found in the" **Adding a detail with an editable list**” article.

BaseGridDetailV2 messages

Main *BaseGridDetailV2* messages are given in table 4.

Table 4. The messages of the detail with a list

Name	Mode	Direction	Description
getCardInfo	Address	Subscription	Returns information about the edit page: its default values, type column name and type column value.
CardSaved	Broadcasting	Subscription	Handles a message of saving the edit page.
UpdateFilter	Broadcasting	Subscription	Refreshes filters in the detail.
GetColumnsValues	Address	Publication	Receives the column values of the edit page model.

The message modes are defined in the “Terrasoft.core.enums.MessageMode” enumeration and message direction in the “Terrasoft.core.enums.MessageDirectionType” enumeration. More information about them can be found in the “**('JavaScript API for platform core' in the on-line documentation)**” article.

BaseGridDetailV2 attributes

Main *BaseGridDetailV2* attributes are given in table 5.

Table 5. The attributes of the detail with a list

Name	Type	Description
ActiveRow	GUID	The value of the primary column of the active record in the list.
IsGridEmpty	BOOLEAN	Indicates that the list is empty.
MultiSelect	BOOLEAN	Indicates if multiple selection is permitted.
SelectedRows	COLLECTION	An array of selected values.
RowCount	INTEGER	Number of rows in the list.
IsPageable	BOOLEAN	Indicates if the page-by-page loading is enabled.
SortColumnIndex	INTEGER	Index of the sorting column.
CardState	TEXT	Opening mode for the record edit page.

EditPageUId	GUID	A unique identifier of the edit page.
ToolsButtonMenu	COLLECTION	The collection of the functional button’s drop-down list.
DetailFilters	COLLECTION	Collection of the detail filters.
IsDetailWizardAvailable	BOOLEAN	Indicates if the detail wizard is available.

The available attribute data types are represented by the *Terrasoft.DataValueType* enumeration. More information about them can be found in the “**('JavaScript API for platform core' in the on-line documentation)**” article.

BaseGridDetailV2 mixins

Main *BaseGridDetailV2* mixins are given in the table 6.

Table 6. The attributes of the detail with a list

Name	Class	Description
GridUtilities	*Terrasoft.GridUtilities*	Mixin for the list.
WizardUtilities	*Terrasoft.WizardUtilities*	Mixin for the detail wizard.

More information about the *GridUtilities* mixin is given below.

BaseGridDetailV2 methods

Main *BaseGridDetailV2* methods are given in table 7.

Table 7. The methods of the base detail with a list

Name	Parameters	Description
init	*{Function} callback* – callback function. *{Object} scope* – the context of the method execution.	Overrides the *BaseDetailV2* method. Calls the parent’s *init* method logic, registers the messages, initializes the filters.
initData	*{Function} callback* – callback function. *{Object} scope* – the context of the method execution.	The override of the *BaseDetailV2* method. Calls the parent’s *initData* method logic, initializes the data collection of the list view.
loadGridData	No	Executes the load of the list data.
initGridData	No	Executes the initialization of the default values for working with the list.
getGridData	No	Returns list collection.
getFilters	No	Returns the detail filter collection.
getActiveRow	No	Returns the identifier of the selected record in the list.
addRecord	*{String} editPageUId* – identifier of typed edit page.	Adds the new record to the list. Saves the edit page, if needed.
copyRecord	*{String} editPageUId* – identifier of typed edit page.	Copies the record and opens the edit page.
editRecord	*{Object} record* – record model for editing.	Opens edit page of the selected record.
subscribeSandboxEvents	No	It is subscribed to the messages necessary for the detail operation.
UpdateDetail	*{Object} config* – configuration object	The override of the *BaseDetailV2* method.

	that contains the properties of the detail.	Calls the parent's *updateDetail* method logic, updates the detail.
OpenCard	*{String} operation* – operation type (creating/modifying) *{String} typeColumnValue* – the value of record typing column. *{String} recordId* – record identifier.	Opens edit page.
onCardSaved	No	Handles the save event of the edit page where the detail is located.
addToolsButtonMenuItems	*{Terrasoft.BaseViewModelCollection} toolsButtonMenu* – The collection of the functional button drop-down list.	Adds elements to the collection of the functional button drop-down list.
initDetailFilterCollection	No	Initializes the detail filter.
setFilter	*{String} key* – filter type. *{Object} value* – filter value.	Sets the detail filter value.
loadQuickFilter	*{Object} config* – parameters of the filters module load.	Loads the quick filter.
destroy	No	Clears the data, exports the detail.

BaseGridDetailV2 array of modifications

In the *diff* modifications array of the base detail, only a base container for detail view is defined:

```
diff: /**SCHEMA_DIFF*/ [
  {
    // Element for displaying the list.
    "operation": "insert",
    "name": "DataGrid",
    "parentName": "Detail",
    "propertyName": "items",
    "values": {
      "itemType": Terrasoft.ViewItemType.GRID,
      …
    }
  },
  {
    // List reloading button.
    "operation": "insert",
    "parentName": "Detail",
    "propertyName": "items",
    "name": "loadMore",
    "values": {
      "itemType": Terrasoft.ViewItemType.BUTTON,
      …
    }
  },
  {
    // Record adding button.
    "operation": "insert",
    "name": "AddRecordButton",
    "parentName": "Detail",
    "propertyName": "tools",
    "values": {
      "itemType": Terrasoft.ViewItemType.BUTTON,
      …
```

```
    }
  },
  {
    // Typed record adding button.
    "operation": "insert",
    "name": "AddTypedRecordButton",
    "parentName": "Detail",
    "propertyName": "tools",
    "values": {
      "itemType": Terrasoft.ViewItemType.BUTTON,
      …
    }
  },
  {
    // Detail menu.
    "operation": "insert",
    "name": "ToolsButton",
    "parentName": "Detail",
    "propertyName": "tools",
    "values": {
      "itemType": Terrasoft.ViewItemType.BUTTON,
      …
    }
  }
] /**SCHEMA_DIFF*/
```

The GridUtilitiesV2 mixin

GridUtilitiesV2 is a mixin that implements the logic of the "list" control. Features that are implemented in the *Terrasoft.configuration.mixins.GridUtilities* class:

1. Message subscription
2. Data load.
3. Working with the list:
   - selection of the records (the search of the active records)
   - adding, deleting and modifying the records
   - setting up the filters
   - sorting
   - exporting to the file
   - checking the access permissions to the list records

GridUtilitiesV2 methods

Main GridUtilitiesV2 methods are given in table 8.

Table 8. The methods of the base detail with a list

Name	Parameters	Description
init	No	Subscribes to the events.
destroy	No	Deletes event subscriptions.
loadGridData	No	Executes the load of the list data.
beforeLoadGridData	No	Prepares the view model to the data load.
afterLoadGridData	No	Prepares the view model after the data load.
onGridDataLoaded	*{Object} response* – the result of fetching the data from the database.	A handler of the data load event. Executes when the server returns the data.
addItemsToGridData	*{Object} dataCollection* –	Adds a collection of new elements to the list

	collection of new elements. *{Object} options* – adding parameters.	collection.
reloadGridData	No	Reloads the list.
initQueryOptions	*{Terrasoft.EntitySchemaQuery} esq* – in this query the necessary settings will be initialized.	Initializes the settings of the query instance, such as pagination and hierarchy.
initQuerySorting	*{Terrasoft.EntitySchemaQuery} esq* – in this query the necessary settings will be initialized.	Initializes the sorting column.
prepareResponseCollection	*{Object} collection* – list elements collection.	Modifies the data collection before loading it to the list.
getFilters	No	Returns filters that are applied to current schema. It is overridden in the inheritors.
exportToFile	No	Exports the contents of the list into a file.
sortGrid	*{String} tag* – a key that shows how to sort the list.	Performs list sorting.
deleteRecords	No	Initiates the deletion of the selected records.
checkCanDelete	*{Array} items* – the identifiers of the selected records. *{Function} callback* – callback function. *{Object} scope* – the context of the method execution.	Checks the ability to delete a record.
onDeleteAccept	No	Performs the deletion after the confirmation of the user.
getSelectedItems	No	Returns the selected records in the list.
removeGridRecords	*{Array} records* – deleted records.	Removes the deleted records from the list.

A detail with editable list

A detail with editable list enables editing records directly in the list without going to the record editing page. To make a detail list editable, you need to modify its schema the following way:

1. Add the dependencies from the *ConfigurationGrid*, *ConfigurationGridGenerator* and *ConfigurationGridUtilities* modules.
2. Connect the *ConfigurationGridUtilites* and *OrderUtilities* mixins.
3. Set the *IsEditable* attribute to “true”.
4. Add a configuration object in the modification array where the properties will be set and the methods that handle the detail list events will be bound.

The development case of creating a detail with an editable list can be found in the "**Adding a detail with an editable list**” article.

The “ConfigurationGrid” module

The *ConfigurationGrid* module contains the implementation of the “Configuration Grid” control. The *Terrasoft.ConfigurationGrid* class is the inheritor of the *Terrasoft.Grid* class. Main *Terrasoft.ConfigurationGrid* methods are given in table 9.

Table 9. Configuration grid methods

Name	Parameters	Description
init	No	Initializes a component. Subscribes to the events.
activateRow	*{String|Number}* id – the identifier of the list string.	Selects the string and adds edit elements.
deactivateRow	*{String|Number}* id – the identifier of the list string.	Removes selection of a string and removes edit elements.
formatCellContent	*{Object} cell* – the cell. *{Object} data* – the data. *{Object} column* – the cell configuration.	Formats the data of a string cell.
onUpdateItem	*{Terrasoft.BaseViewModel} item* – collection element.	The handler of the record update event.
onDestroy	No	Destroys the list and its components.

ConfigurationGridGenerator module

The *Terrasoft.ConfigurationGridGenerator* generates list configuration and is an inheritor of the *Terrasoft.ViewGenerator* class. The methods that are implemented in the *Terrasoft.ConfigurationGridGenerator* class are given in table 10.

Table 10. The methods of the configuration grid generator

Name	Parameters	Description
addLinks	No	Overridden method of the *Terrasoft.ViewGenerator* class. No links will be added to the editable list.
generateGridCellValue	*{Object} config* – column configuration.	Overridden method of the *Terrasoft.ViewGenerator* class. Generates a value configuration in the cell.

The ConfigurationGridUtilities module

The *Terrasoft.ConfigurationGridUtilities* class contains methods that initialize a view model of the list string, process the active record actions and handle "hotkeys".

The main properties of the *Terrasoft.ConfigurationGridUtilities* class are given in table 11 and its methods are provided in table 12.

Table 11. The *Terrasoft.ConfigurationGridUtilities* class properties

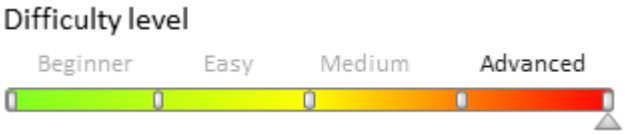
Name	Type	Description
currentActiveColumnName	*String*	The name of the currently selected column.
columnsConfig	*Object*	Column configuration.
systemColumns	*Array*	Collection of the system column names.

Table 12. The *Terrasoft.ConfigurationGridUtilities* class methods

Name	Parameters	Description
onActiveRowAction	*{String} buttonTag* – a tag of the selected action. *{String} primaryColumnValue* – active record identifier.	Handles clicking an action of the active record.
saveRowChanges	*{Object} row* – list string.	Saves the record.

	{Function} callback – callback function. *{Object} scope* – the callback function context.	
activeRowSaved	*{Object} row* – list string. *{Function} callback* – callback function. *{Object} scope* – the callback function context.	Handles the result of saving the record.
initActiveRowKeyMap	*{Array} keyMap* – events description.	Initializes the subscription to the button events in the active string.
getCellControlsConfig	*{Terrasoft.EntitySchemaColumn} entitySchemaColumn* – the column of the list cell.	Returns the configuration of the list cell edit items.
copyRow	*{String} recordId* – the identifier of a copying record	Copies and adds a record to the list.
initEditableGridRowViewModel	*{Function} callback* – callback function. *{Object} scope* – the callback function context.	Initializes the classes of the collection elements of the edited list.

The [Connected entity profile] control

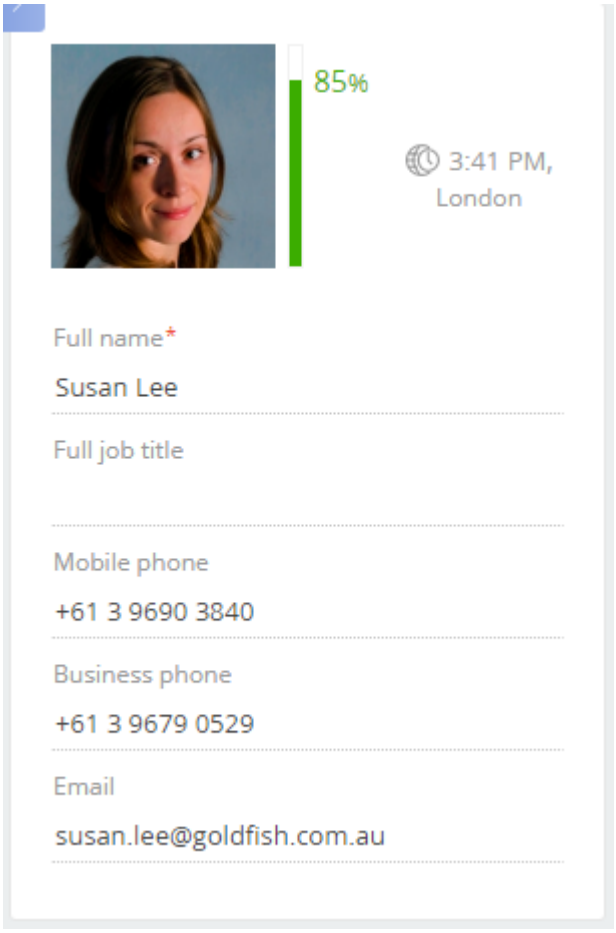


General Information

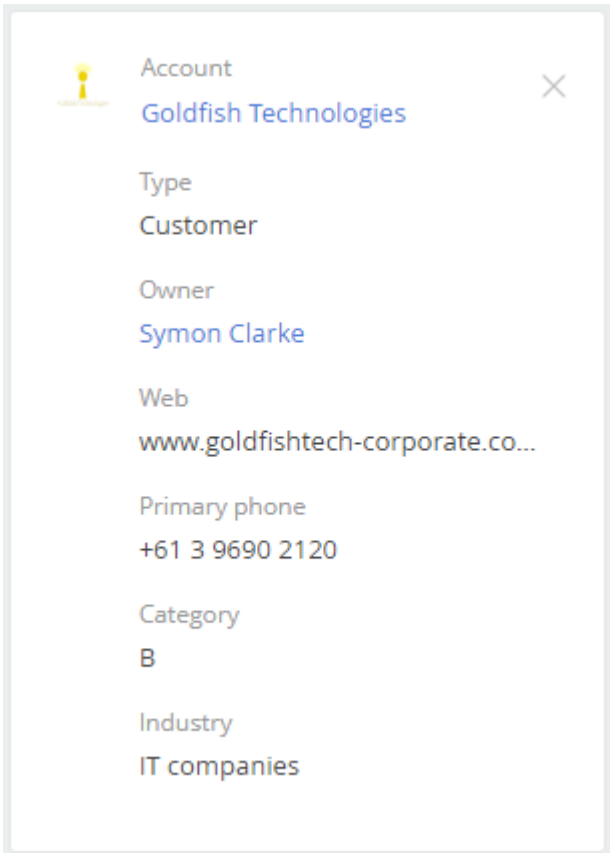
The [Connected entity profile] control (the *Profile* class) is a configuration module (information block) which is populated with information about the connected entity when the page is loading. This element is used in the system as a connected record profile on the section entry editing page. For example, if you open the [Contact] editing page in the [Profile] element (Fig. 1), the contact profile and the associated account information will be displayed (communication via the [Account] column of the [Contact] object, Fig. 2). Learn more about record profiles and connected records in the “Record pages”.

The parent class for *Profile* is *BaseProfileSchema* – a basic schema for creating any related record profiles in the system.

(Fig. 1). [Contact] object profile



(Fig. 2). Connected contact profile of the related [Account] entity



The parent class for *Profile* is *BaseProfileSchema* – a basic schema for creating any related record profiles in the system.

> ATTENTION
>
> All profiles are inherited from *BaseProfileSchema*.

The *BaseProfileSchema* schema implements the ability to display any set of fields of a related entity, as well as any number of different modules.

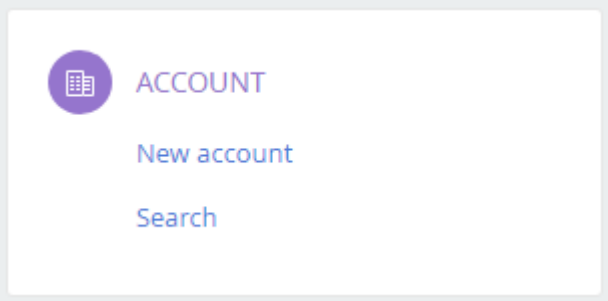
A view is described by the *diff* property (similar to the editing page description process). While embedding a module to the editing page, specify the parameter *masterColumnName* in the module attributes. The parameter stores the name of the column used to connect the profile to the main editing page diagram. *Profile* will download the data based on this column value.

At the initialization stage, the profile object sends a message to *GetColumnInfo* to obtain additional information about its connected column (filters, header, etc.). Then it requests the connected column value, and if it is full, the data for this record is initialized. When you clear the field or change the value, the data in the profile object is reinitialized.

If the profile is empty, i.e. the entry in the link field is not selected, then the name of the field through which the connection is made and the two actions are displayed in it (Fig. 3).

- [Add account] – create a new entry in the link field lookup.
- [Select] – select an existing entry from the list.

(Fig. 3). An empty connected entity profile



> NOTE
>
> When you select an existing related entity, all business logic (defined on the editing page and connected to the referring entity attribute) is superimposed on the lookup. Filtering, query column settings, etc. are retained.
>
> If the attribute is removed from the layout of the editing page, all logic will be lost along with it.

The BaseProfileSchema

The interaction interface is implemented by the standard messages mechanism. The following messages are used:

- *OpenCard* – opens a page for adding an entry using the standard mechanism.
- *UpdateCardProperty* – updates the value of the editing page attribute.
- *GetColumnInfo* – returns communication field information.
- *GetColumnsValues* – returns the values of the requested editing page columns.
- *GetEntityColumnChanges* – subscription to edit the editing page data.
- *CardModuleResponse* – the result of adding a new record through the profile.

The visual content of a profile (buttons, links, modules) is defined in the *diff* modification array.

Profile configuration case

```
define("ContactProfileSchema", ["ProfileSchemaMixin"],
function () {
    return {
```

```
        // Name of object schema.
        entitySchemaName: "Contact",
        // Mixins.
        mixins: {
            // Mixin with functions for obtaining icons and profile pictures.
            ProfileSchemaMixin: "Terrasoft.ProfileSchemaMixin"
        },
        // The diff modification array.
        diff: /**SCHEMA_DIFF*/[
            {
                // Insterting.
                "operation": "insert",
                // Entity name.
                "name": "Account",
                // The name of the parent element in which to insert.
                "parentName": "ProfileContentContainer",
                // The property of the parent element with which the operation is
performed.
                "propertyName": "items",
                // The values of the inserted item.
                "values": {
                    // Binding the Account property to the Contact object value.
                    "bindTo": "Account",
                    // Layout configuration. Element positioning.
                    "layout": {
                        "column": 5,
                        "row": 1,
                        "colSpan": 19
                    }
                }
            }
            // ...Other modification array configuration objects.
        ]/**SCHEMA_DIFF*/
    };
});
```

An example of embedding a profile to an editing page.

```
//Defining the editing page schema and its dependencies.
define("ContactPageV2", ["BaseFiltersGenerateModule", "BusinessRuleModule",
"ContactPageV2Resources",
            "ConfigurationConstants", "ContactCareer",
"DuplicatesSearchUtilitiesV2"],
function (BaseFiltersGenerateModule, BusinessRuleModule, resources,
ConfigurationConstants, ContactCareer) {
    return {
        entitySchemaName: "Contact",
        // Modules used .
        modules: /**SCHEMA_MODULES*/{
            // Account profile module.
            "AccountProfile": {
                // Profile configuration.
                "config": {
                    // Shema name.
                    "schemaName": "AccountProfileSchema",
                    // A characteristic indicating that circuit configuration is
initialized.
                    "isSchemaConfigInitialized": true,
                    // A characteristic indicating that HistoryState is not used.
                    "useHistoryState": false,
                    // Profile parameters.
                    "parameters": {
```

```
                        // View model configuration.
                        "viewModelConfig": {
                            // The name of the connected entity column.
                            masterColumnName: "Account"
                        }
                    }
                }
            }
        }/**SCHEMA_MODULES*/,
        // Modification array.
        diff: /**SCHEMA_DIFF*/[
            {
                "operation": "insert",
                "parentName": "LeftModulesContainer",
                "propertyName": "items",
                // Profile name.
                "name": "AccountProfile",
                // Values.
                "values": {
                    // Element type - module.
                    "itemType": Terrasoft.ViewItemType.MODULE
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

The BaseMultipleProfileSchema profile schema

In addition to the *BaseProfileSchema* base profile schema, there are additional schemas that implement specific functionality of user profiles.

The profile described by the *BaseMultipleProfileSchema* schema can contain any number of profiles and switch between them by using the logic of selected values from several directories. The main difference from the base profile is the ability to embed other profiles to the current profile. In this case, the built-in profiles can communicate with each other via messages. Otherwise, the way it works with the editing page is similar to that of the base profile.

ATTENTION

The *BaseMultipleProfileSchema* profiles must be inherited from the *BaseRelatedProfileSchema* base profile schema, which can be dependent or embedded to other profiles.

Example of the BaseMultipleProfileSchema client profile module.

```
// Defining a profile.
define("ClientProfileSchema", ["ProfileSchemaMixin"],
function () {
    return {
        // Mixins.
        mixins: {
            ProfileSchemaMixin: "Terrasoft.ProfileSchemaMixin"
        },
        // Attributes.
        attributes: {
            // Contact profile visibility.
            "IsVisibleContactProfile": {
                dataValueType: this.Terrasoft.DataValueType.BOOLEAN,
                value: true
            }
        },
        // Methods.
```

```
        methods: {
            // Date-marker. Needed for automatic tests.
            getProfileModuleContainerDataMarker: function () {
                return "client-profile-module-container";
            },
            // Returns the header.
            getBlankSlateHeaderCaption: function () {
                return this.get("Resources.Strings.Client");
            },
            // Returns the warning icon.
            getWarningIcon: function () {
                return this.getImageUrlByResourceKey("Resources.Images.WarningIcon");
            },
            // Checks a warning display.
            getIsVisibleWarning: function () {
                var masterColumnNames = this.get("MasterColumnNames");
                if (!masterColumnNames) {
                    return false;
                }
                var masterColumnValues = masterColumnNames.filter(function
(columnName) {
                    var value = this.get(columnName);
                    return !this.Ext.isEmpty(value);
                }, this);
                return masterColumnValues.length > 1;
            },
            // The event handler of the profile column change.
            onProfileColumnChanged: function () {
                this.set("IsVisibleContactProfile", !this.getIsVisibleWarning());
                return this.callParent(arguments);
            },
            // The column change event handler.
            onColumnChanged: function () {
                this.callParent(arguments);
                this.set("IsVisibleContactProfile", !this.getIsVisibleWarning());
            },
            // The [Clear] button hint.
            getClearButtonHint: function () {
                var clearActionCaption =
this.get("Resources.Strings.ClearButtonCaption");
                var masterColumnCaption = this.get("Resources.Strings.Client");
                return this.Ext.String.format("{0} {1}", clearActionCaption,
masterColumnCaption);
            }
        },
        // Schema modules.
        modules: /**SCHEMA_MODULES*/{
            // Built-in client profile of the account.
            "AccountClientProfile": {
                // Profile configuration.
                "config": {
                    "schemaName": "ClientAccountProfileSchema",
                    "isSchemaConfigInitialized": true,
                    "useHistoryState": false,
                    "parameters": {
                        "viewModelConfig": {
                            masterColumnName: "Account"
                        }
                    }
                }
            },
            // Client contact embedded profile.
```

```
    "ContactClientProfile": {
        "config": {
            "schemaName": "ClientContactProfileSchema",
            "isSchemaConfigInitialized": true,
            "useHistoryState": false,
            "parameters": {
                "viewModelConfig": {
                    masterColumnName: "Contact"
                }
            }
        }
    }
}/**SCHEMA_MODULES*/,
// Profile view modifications.
diff: /**SCHEMA_DIFF*/[
    {
        "operation": "remove",
        "name": "ProfileIcon"
    },
    {
        "operation": "remove",
        "name": "ProfileHeaderContainer"
    },
    {
        "operation": "insert",
        "name": "ClientProfilesContainer",
        "parentName": "ProfileContentContainer",
        "propertyName": "items",
        "values": {
            "itemType": this.Terrasoft.ViewItemType.CONTAINER,
            "items": [],
            "layout": {
                "column": 0,
                "row": 0,
                "colSpan": 24,
                "rowSpan": 24
            }
        }
    },
    {
        "operation": "insert",
        "name": "WarningIcon",
        "parentName": "ClientProfilesContainer",
        "propertyName": "items",
        index: 0,
        "values": {
            "getSrcMethod": "getWarningIcon",
            "readonly": true,
            "generator": "ImageCustomGeneratorV2.generateSimpleCustomImage",
            "visible": { "bindTo": "getIsVisibleWarning" },
            "classes": {
                "wrapClass": ["warning-icon"]
            },
            "hint": { "bindTo": "Resources.Strings.WarningMessage" }
        }
    },
    {
        "operation": "insert",
        "parentName": "ClientProfilesContainer",
        "propertyName": "items",
        "name": "AccountClientProfile",
        "values": {
```

```
                    "itemType": this.Terrasoft.ViewItemType.MODULE
                }
            },
            {
                "operation": "insert",
                "parentName": "ClientProfilesContainer",
                "propertyName": "items",
                "name": "ContactClientProfile",
                "values": {
                    "itemType": this.Terrasoft.ViewItemType.MODULE,
                    "visible": { "bindTo": "IsVisibleContactProfile" }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
}
);
```

Example of a built-in BaseRelatedProfileSchema client profile

```
define("ClientContactProfileSchema", ["ProfileSchemaMixin"],
function () {
    return {
        // Schema object name.
        entitySchemaName: "Contact",
        // Mixins.
        mixins: {
            ProfileSchemaMixin: "Terrasoft.ProfileSchemaMixin"
        },
        // Methods.
        methods: {
            getProfileHeaderCaption: function () {
                return this.get("Resources.Strings.ProfileHeaderCaption");
            }
        },
        // Modifications array.
        diff: /**SCHEMA_DIFF*/[
            {
                "operation": "insert",
                "name": "Account",
                "parentName": "ProfileContentContainer",
                "propertyName": "items",
                "values": {
                    "bindTo": "Account",
                    "enabled": false,
                    "layout": {
                        "column": 5,
                        "row": 1,
                        "colSpan": 19
                    }
                }
            },
            {
                "operation": "insert",
                "name": "Job",
                "parentName": "ProfileContentContainer",
                "propertyName": "items",
                "values": {
                    "bindTo": "Job",
                    "enabled": false,
                    "layout": {
                        "column": 5,
```

```
                "row": 2,
                "colSpan": 19
            }
        }
    },
    {
        "operation": "insert",
        "name": "Type",
        "parentName": "ProfileContentContainer",
        "propertyName": "items",
        "values": {
            "bindTo": "Type",
            "enabled": false,
            "layout": {
                "column": 5,
                "row": 3,
                "colSpan": 19
            }
        }
    },
    {
        "operation": "insert",
        "name": "MobilePhone",
        "parentName": "ProfileContentContainer",
        "propertyName": "items",
        "values": {
            "bindTo": "MobilePhone",
            "enabled": false,
            "layout": {
                "column": 5,
                "row": 4,
                "colSpan": 19
            }
        }
    },
    {
        "operation": "insert",
        "name": "Phone",
        "parentName": "ProfileContentContainer",
        "propertyName": "items",
        "values": {
            "bindTo": "Phone",
            "enabled": false,
            "layout": {
                "column": 5,
                "row": 5,
                "colSpan": 19
            }
        }
    },
    {
        "operation": "insert",
        "name": "Email",
        "parentName": "ProfileContentContainer",
        "propertyName": "items",
        "values": {
            "bindTo": "Email",
            "enabled": false,
            "layout": {
                "column": 5,
                "row": 6,
                "colSpan": 19
```

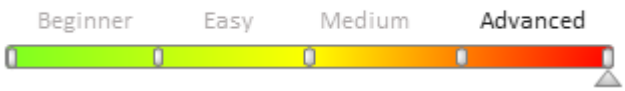
```
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

An example of embedding a client profile module to the editing page.

```
// Editing page modules.
modules: /**SCHEMA_MODULES*/{
    // Module name.
    "ClientProfile": {
        // Configuration.
        "config": {
                    // A characteristic indicating that circuit configuration is
initialized.
                    "isSchemaConfigInitialized": true,
                    // A characteristic indicating that HistoryState is not used.
                    "useHistoryState": false,
            // Schema name.
            "schemaName": "ClientProfileSchema",
            // Parameters.
            "parameters": {
                // View Model Configuration.
                "viewModelConfig": {
                    // Connected entity column name.
                    "masterColumnName": "Client"
                }
            }
        }
    }
}/**SCHEMA_MODULES*/,
// The diff modification array.
diff: /**SCHEMA_DIFF*/[
   {
       "operation": "insert",
       // Profile name.
       "name": "ClientProfile",
       "parentName": "LeftModulesContainer",
       "propertyName": "items",
       // Values.
       "values": {
           // Element type - module.
           "itemType": Terrasoft.ViewItemType.MODULE
       }
   }
]/**SCHEMA_DIFF*/
```

SourceCodeEditMixin class description and work examples.

Difficulty level



Introduction

When developing controls, it may be necessary to implement the string value editing functionality with the HTML, JavaScript, or LESS code. The *SourceCodeEditMixin* class was created for that.

SourceCodeEditMixin is a **mixin**, which provides a user-friendly string editing interface for class enrichment. Its concept resembles that of multiple inheritance.

SourceCodeEditMixin properties and methods.

The main *SourceCodeEditMixin* class properties are listed in table 1, and its methods are listed in table 2.

Table 1. SourceCodeEditMixin mixin proiperties

Name	Type	Details
sourceCodeEdit	*Terrasoft.SourceCodeEdit*	An instance of the source code editor control.
sourceCodeEditContainer	*Terrasoft.Container*	An instance of the container with the source code editor.

Table 2. Main SourceCodeEditMixin mixin methods

Name	Parameters	Details
openSourceCodeEditModalBox	*No*	A method that implements the opening of a window for editing source code.
loadSourceCodeValue	*No*	Abstract method. Must be implemented in the main class. Implements the logic of obtaining the value to edit.
saveSourceCodeValue	*value {String}*	Abstract method. Must be implemented in the main class. Implements the logic of saving editing results in the main class object.
destroySourceCodeEdit	*No*	A method that implements the purification of mixin resources.
getSourceCodeEditModalBoxStyleConfig	*No*	Returns a key-value object that describes styles in the modal window for editing source code.
getSourceCodeEditStyleConfig	*No*	Returns a key-value object that describes styles applied to source editor controls.
getSourceCodeEditConfig	*No*	Returns a key-value object that describes the properties that the created instance of the source code editor controls will have.

Main properties of the created instance of source code editor controls (see *getSourceCodeEditConfig* method) are listed in eable 3.

Table 3. Created source editor properties

Name	Type	Details
showWhitespaces	*Boolean*	Displaying invisible strings Default value: *false*.
language	*SourceCodeEditEnums.Language*	Language syntax. Selected from the *SourceCodeEditEnums.Language* enumeration (Table 4). Default value: *SourceCodeEditEnums.Language.JAVASCRIPT*.
theme	*SourceCodeEditEnums.Theme*	Editor theme Selected from the *SourceCodeEditEnums.Theme* enumeration (Table 5). Default value:

		SourceCodeEditEnums.Theme.CRIMSON_EDITOR.
showLineNumbers	*Boolean*	Display string numbers. Default value: *true*.
showGutter	*Boolean*	Set the gap between columns. Default value: *true*.
highlightActiveLine	*Boolean*	Highlighting the active line. Default value: *true*.
highlightGutterLine	*Boolean*	Highlighting of the inter-column space line. Default value: *true*.

Table 4. Language syntax of the source code editor (SourceCodeEditEnums.Language)

Enumeration member	Programming language
JAVASCRIPT	JavaScript
CSHARP	C#
LESS	LESS
CSS	CSS
SQL	SQL
HTML	HTML

Table 5. - Source code editor themes (SourceCodeEditEnums.Theme)

Enumeration member	Subject
SQLSERVER	SQL editor subject
CRIMSON_EDITOR	Crimson editor subject

Use case

Add a mixin to the *mixins* property to use it in a control:

```
// Connecting a mixin.
mixins: {
    SourceCodeEditMixin: "Terrasoft.SourceCodeEditMixin"
},
```

Mixin functionality gets a value by calling the *getSourceCodeValue()* abstract *getter* method. It returns the string for editing. The *getter* method should be implemented each time a mixin is used:

```
// Implementing a field string value.
getSourceCodeValue: function () {
    // The "getValue()" method is implemented in the "Terrasoft.BaseEdit" base class.
    return this.getValue();
},
```

After the editing is completed, the mixin will call the *setSourceCodeValue()* abstract *setter* method to save the result. The *setter* method should be implemented each time a mixin is used:

```
// Method implementation for setting up a result string.
setSourceCodeValue: function (value) {
    // The "setValue()" method is implemented in the "Terrasoft.BaseEdit" base class.
    this.setValue(value);
},
```

Call the *openSourceCodeBox()* method to open the source code editing window. The method is called in the main class instance context. For example, when the *onSourceButtonClick* method of the component is called.

```
// Implementing the process of calling a source editor window method.
onSourceButtonClick: function () {
    this.mixins.SourceCodeEditMixin
      .openSourceCodeBox.call(this);
},
```

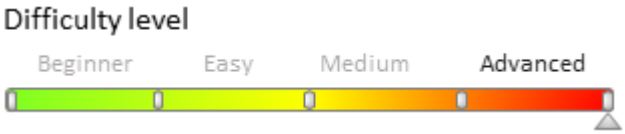
After the work with the primary class instance is finished, it is deleted from the memory. The *SourceCodeEditMixin* requires freeing certain resources. To do this, the *destroySourceCode* method is called in the main class instance context.

```
onDestroy: function () {
    this.mixins.SourceCodeEditMixin
      .destroySourceCode.apply(this, arguments);
    this.callParent(arguments);
}
```

Below is the complete source code:

```
// Adding a mixin module to dependencies.
define("SomeControl", ["SomeControlResources", "SourceCodeEditMixin"],
  function (resources) {
      Ext.define("Terrasoft.controls.SomeControl", {
          extend: "Terrasoft.BaseEdit",
          alternateClassName: "Terrasoft.SomeControl",

          // Connecting a mixin.
          mixins: {
              SourceCodeEditMixin: "Terrasoft.SourceCodeEditMixin"
          },

          // Method implementation for obtaining a string value.
          getSourceCodeValue: function () {
              // The "getValue()" method is implemented in the "Terrasoft.BaseEdit"
base class.
              return this.getValue();
          },

          // Method implementation for setting up the result string.
          setSourceCodeValue: function (value) {
              // The "setValue()" method is implemented in the "Terrasoft.BaseEdit"
base class.
              this.setValue(value);
          },

          // Implementing the process of calling a source editor window method.
          onSourceButtonClick: function () {
              this.mixins.SourceCodeEditMixin
                .openSourceCodeBox.call(this);
          },

          // Implementing a call to clear mixin resources.
          onDestroy: function () {
              this.mixins.SourceCodeEditMixin
                .destroySourceCode.apply(this, arguments);
              this.callParent(arguments);
          }
      });
  });
```

Blocking edit page fields



Introduction

During the development of the bpm’online custom functions you may need to block all fields and details on the page when specific condition is met. Mechanism of blocking of the edit page fields can simplify the process without creating a number of business rules.

> ⚠ ATTENTION
>
> Blocking mechanism is implemented in the bpm'online version 7.11.1 or higher.

> ⚠ ATTENTION
>
> You can disable the function for blocking edit page fields using the "CompleteCardLockout" option on the feature toggle page (see. "**Feature Toggle. Mechanism of enabling and disabling functions**"). Use the following URL to open the feature toggle page: ../0/Nui/ViewModule.aspx#BaseSchemaModuleV2/FeaturesPage. For example, https://mycompany.bpmonline.com/0/Nui/ViewModule.aspx#BaseSchemaModuleV2/FeaturesPage.

As a result of applying the clocking mechanism on the edit page, all fields and details will be blocked. If the field has binding for the *enabled* property in the *diff* array element or in the business rule, the mechanism will not block this field. Details hide buttons and menu items for performing operations with the record. A detail with an editable list still features an ability to access the object page, however all fields will be block in accordance with the business rules.

> ⚠ ATTENTION
>
> The blocking mechanism is intended for blocking details with a list and an editable list. To ensure the correct operation of the mechanism for details with editable fields, create a replacement schema for this detail and control the availability of fields using the *IsEnabled* attribute.

To enable the blocking mechanism, set the source code of the edit page to *false* for the *IsModelItemsEnabled* model attribute:

```
this.set(“IsModelItemsEnabled”, false);
```

Or set the default value for the attribute:

```
"IsModelItemsEnabled": {
    dataValueType: Terrasoft.DataValueType.BOOLEAN,
    value: true,
    dependencies: [{
        columns: ["PaymentStatus"],
        methodName: "setCardLockoutStatus"
    }]
}
```

Additionally, to operate the locking mechanism on a specific edit page in the *diff* array of this page, specify the *DisableControlsGenerator* generator for the containers in which you want to block fields. Therefore, to block all

fields of the edit page, specify the global *CardContentWrapper* container:

```
diff: /**SCHEMA_DIFF*/[
    {
        "operation": "merge",
        "name": "CardContentWrapper",
        "values": {
            "generator": "DisableControlsGenerator.generatePartial"
        }
    }
]/**SCHEMA_DIFF*/
```

Blocking exceptions

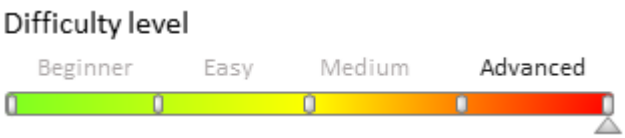
It is possible to exclude blocking for some fields and details. To do this, override the *getDisableExclusionsDetailSchemaNames()* and *getDisableExclusionsColumnTags()* methods. These methods return lists of fields and details that should not be blocked by the mechanism. The implementation of methods is available below:

```
getDisableExclusionsColumnTags: function() {
    return ["SomeField"];
}
getDisableExclusionsDetailSchemaNames: function() {
    return ["SomeDetailV2"]
}
```

More complex exception logic can be implemented by overriding the *isModelItemEnabled()* method for edit fields and the *isDetailEnabled()* method for details. These methods are called for each field and detail. They receive the name and return the availability signal of the field or detail. The implementation of methods is available below:

```
isModelItemEnabled: function(fieldName) {
    var condition = this.get("SomeConditionAttribute");
    if (fieldName === "ExampleField" || condition)) {
        return true;
    }
    return this.callParent(arguments);
}

isDetailEnabled: function(detailName) {
    if (detailName === "ExampleDetail") {
        var exampleDate = this.get("Date");
        var dateNow = new Date(this.Ext.Date.now());
        var condition = this.Ext.Date.isDate(exampleDate) && exampleDate >= dateNow;
        return condition;
    }
    return this.callParent(arguments);
}
```

Dashboard widgets



General information

Dashboard widgets (analytic elements) are used for data analysis of sections. Go to the "Dashboards" view of the required section to work with its analytics. Use the [Dashboards] section to work with the entirety of bpm'online section data analytics.

To learn more about bpm’online dashboard widgets, please refer to the “**Section analytics**” article.

Data storage structure of dashboards

The dashboards section is a user-defined set of tab elements. The mechanism for working with dashboards is implemented with the help of the *DashboardManager* dashboard client manager and the *DashboardManagerItem* element client manager, which represents the tabs. The *SysDashboard* object is responsible for dashboards in the system. The *SysDashboard* object properties are described in table 1.

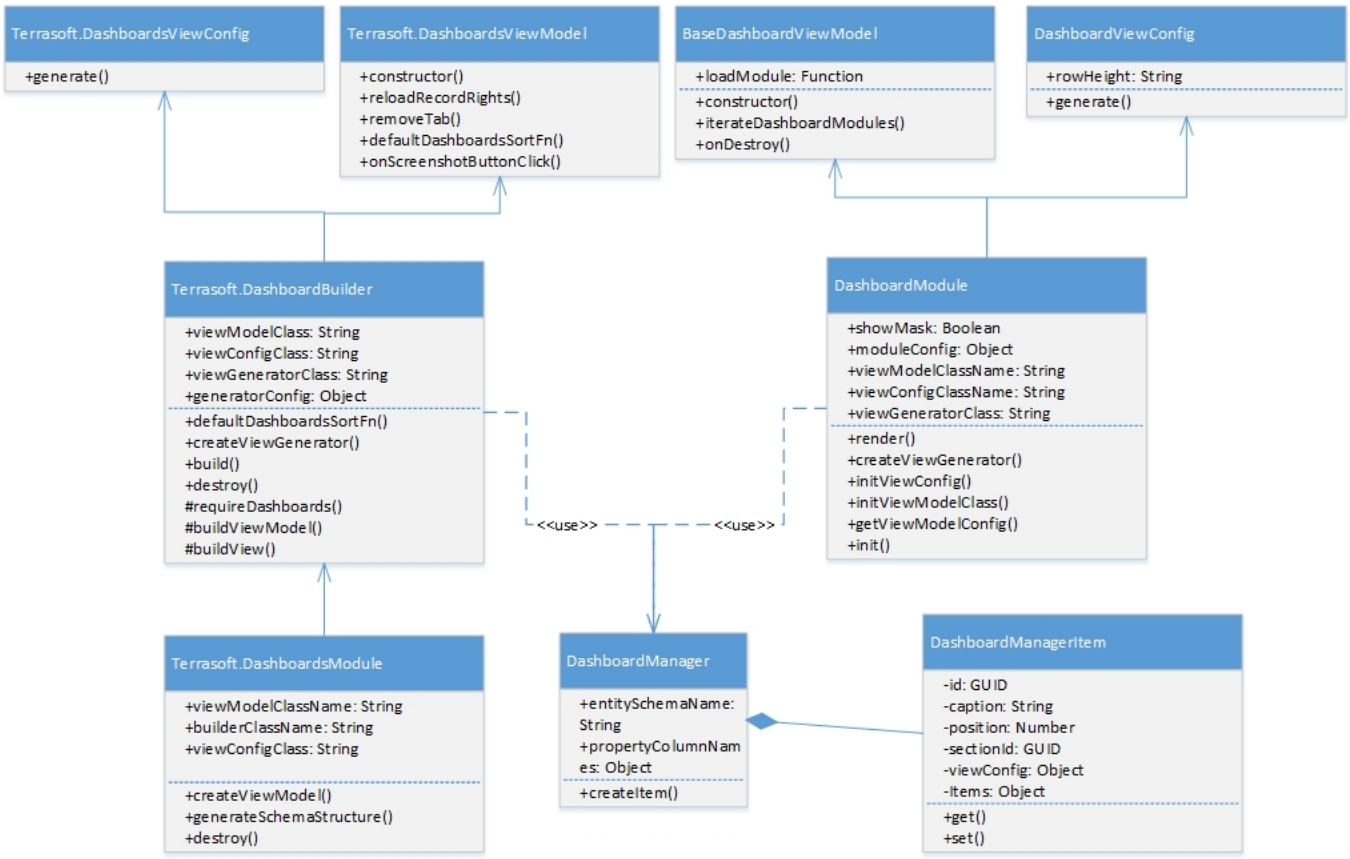
Table 1. SysDashboard object properties

Name	Header	Type	Details
Caption	Header	String	This information is displayed in the tab header.
Position	Position	Number	If a position is not specified, the elements are displayed in alphabetical order.
Section	Section	Lookup	System section.
ViewConfig	Element (widget) view configuration	Array	[{ // Element type (Terrasoft.ViewItemType). itemType: "4", // Element name. name: "SomeInvoiceChart", // View configuration. layout: { columns: 4, rows: 4, colspan: 4, rowspan: 4 } }, {...}]
Items	Element (widget) module configuration	JSON Object	{ // The name of the element for which the module settings are defined. "SomeInvoiceChart": { // Name of the “DashboardItem” view element. “widgetType”: "Chart", // Parameters required to display data for a particular “DashboardItem” element. "parameters": { “caption”: "some caption", ... }, }, {...} }

Implementing functionality in the dashboards view mode

The hierarchy of classes that implement functionality in the dashboards view mode is displayed on Fig. 1.

Fig. 1. The hierarchy of classes that implement functionality in the dashboards view mode



The *SectionDashboardModule:* module:

- The *SectionDashboardBuilder* encapsulates the view generation logic and view model class for the [Dashboards] section module.
- *SectionDashboardsViewModel* – the model class of the [Dashboards] section view model.
- *SectionDashboardsModule* – [Dashboards] section class module.

The *DashboardModule* module:

- *DashboardViewConfig* – a class that generates the view configuration for the dashboards page view module.
- *BaseDashboardViewModel* – a base class for the dashboards page view model.
- *DashboardModule* – a class that contains functionality for working with dashboard modules.

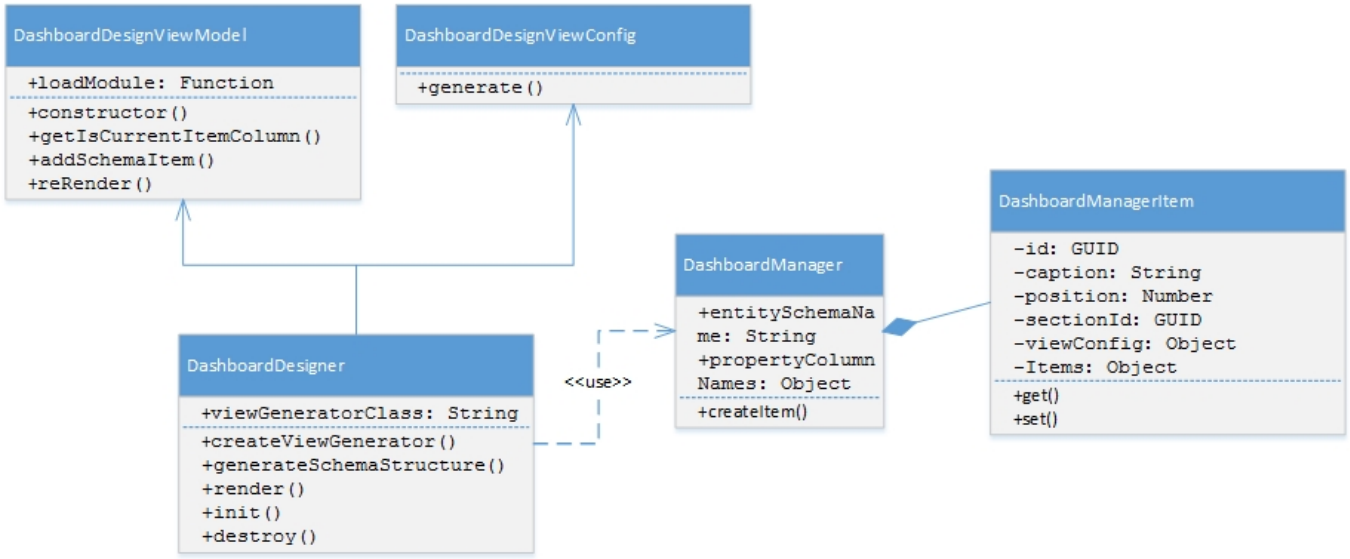
The *DashboardBuilder* module:

- *DashboardsViewConfig* – a class that generates a dashboards module view configuration.
- *BaseDashboardsViewModel* – a base class of the dashboards section view model.
- *DashboardBuilder* – a class for dashboards module construction.

Implementing functionality in the dashboards view mode

The hierarchy of classes that implement the functionality in the dashboards view mode is displayed in Fig. 2.

Fig. 2. The hierarchy of classes that implement the functionality in the dashboards view mode



The *DashboardDesigner* module:

- *DashboardDesignerViewConfig* – a class that generates the view configuration for the dashboards designer module.
- *DashboardDesignerViewModel* – a class of the dashboards designer view model.
- *DashboardDesigner* – dashboard visual module class.

Base classes that implement widget functionality

BaseDashboardsViewModel – a base class of the dashboards section view model. To use this class, register the following messages in the module:

- *GetHistoryState* (publish; ptp);
- *ReplaceHistoryState* (publish; broadcast);
- *HistoryStateChanged* (subscribe; broadcast);
- *GetWidgetParameters* (subscribe; ptp);
- *PushWidgetParameters* (subscribe; ptp) – if the parameters are drawn from modules (useCustomParameterMethods = true).

BaseWidgetDesigner – base widget settings view schema. Main methods:

- *GetWidgetConfig()* – returns the current widget settings object.
- *GetWidgetConfigMessage()* – returns the name of the message used for getting widget module settings.
- *GetWidgetModuleName()* – returns the name of the widget module.
- *GetWidgetRefreshMessage()* – returns the name of the widget update message.
- *getWidgetModulePropertiesTranslator()* – returns the connecting object of widget module properties and widget module settings.

BaseAggregationWidgetDesigner – contains methods for working with aggregate columns and aggregation types.

DashboardEnums – contains an enumeration of widget properties.

Terrasoft.DashboardEnums.WidgetType – contains the widget view mode and design mode configuration of the dashboards. The configuration is defined by the following properties:

- *moduleName* – widget module name.
- *ConfigurationMessage* – the name of the module settings receiving message.
- *ResultMessage* – the name of the message that returns widget designer module settings.
- *StateConfig (stateObj)* – widget designer schema name.

Charts

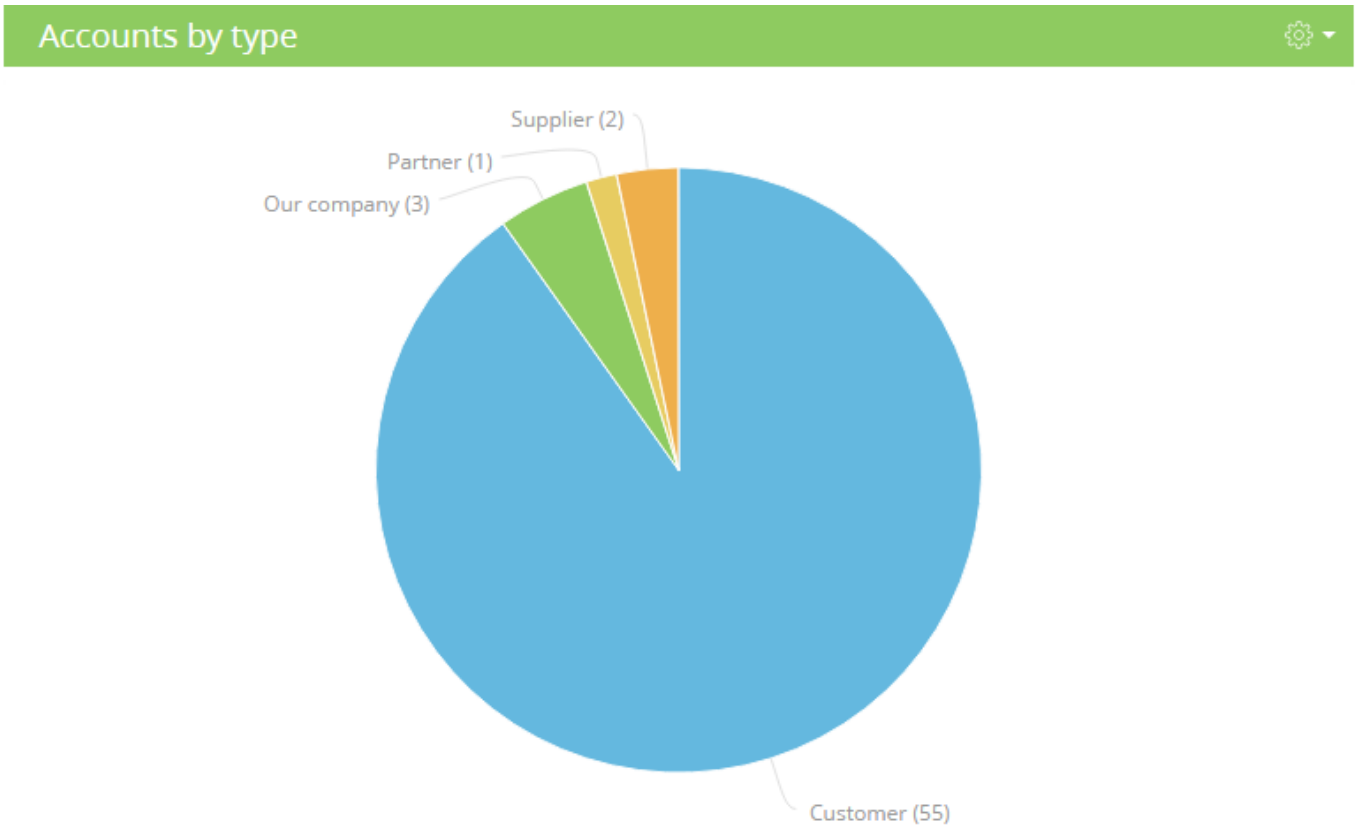
Difficulty level

Beginner Easy Medium Advanced

General information

Charts display multiple system records in the form of diagrams of different types. For example, you can display a pie chart of accounts distributed by type. Charts display information in the form of different diagram types or in a data list form. Learn more about charts in the “The “Chart” dashboard component” article. Chart settings are described in the How to set up a “Chart” dashboard component article.

The “Charts” dashboard



Charts functionality implementation classes

ChartViewModel – chart view model.

ChartViewConfig – generates the chart view model.

ChartModule – a module designed to work with charts.

ChartDesigner – view model schema of a chart.

ChartModuleHelper – generates a query using the Terrasoft.EntitySchemaQuery object.

ChartDrillDownProvider – contains methods for working with the “Show data” function (used for working with chart series).

Chart setup parameters

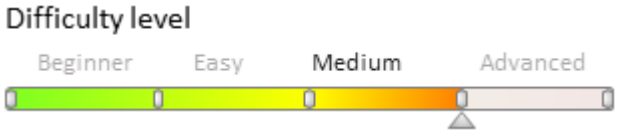
To configure a chart, you need to add the JSON configuration object with the chart properties to the widget module configuration. The widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “**Dashboard widgets**” article.

Set the “Chart” value to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible chart parameters are listed in table 1.

Table 1. Chart setup parameters

Name	Type	Details
seriesConfig	*object*	The settings of an embedded chart in a series.
orderBy	*string*	Sorting field.
orderDirection	*string*	Sorting direction.
caption	*string*	Chart header.
sectionId	*string*	Section id.
xAxisDefaultCaption	*string*	Default X-axis header.
yAxisDefaultCaption	*string*	Default Y-axis header.
primaryColumnName	*string*	Name of initial column. The “id” column is the default one.
yAxisConfig	*object*	Array of the Y-axis name settings.
schemaName	*string*	Chart object.
sectionBindingColumn	*string*	Section link column.
func	*string*	Aggregate function.
type	*string*	Chart type.
XAxisCaption	*string*	X-axis caption.
YAxisCaption	*string*	Y-axis caption.
xAxisColumn	*string*	The X-axis grouping column.
yAxisColumn	*string*	The Y-axis grouping column.
styleColor	*string*	Chart color.
filterData	*object*	Filter settings.

Metrics



The “Metric” dashboard (Fig. 1) displays the number (or date) received by inquiring system data, for example, a total number of company’s employees. Learn more about analytics in the “The “Metric” dashboard component” article. Analytics settings are described in the “Setting up the “Metric” dashboard component” article.

(Fig. 1). A “metric” dashboard

Functionality implementation classes of the “Metric” dashboard

IndicatorViewModel – metric view model.

IndicatorViewConfig – generates the configuration of the metric view model.

IndicatorModule – a module designed to work with metrics.

IndicatorDesigner – view model schema of the metric edit page.

Metric settings

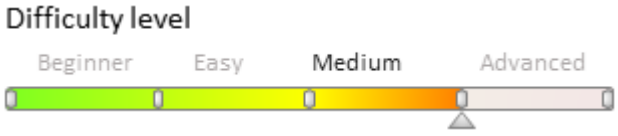
To configure a metric, you need to add the JSON configuration object with Metric properties to the widget module configuration. The widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “**Dashboard widgets**” article.

Set the “Metric” value to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible metric parameters are listed in table 1.

Table 1. Metric settings

Name	Type	Details
caption	*string*	Metric header
sectionId	*string*	Section id.
entitySchemaName:	*string*	Metric object.
sectionBindingColumn	*string*	Section link column.
columnName	*string*	Name of aggregating column.
format	*object*	Metric format.
filterData	*object*	Filter settings.
aggregationType	*number*	Type of aggregating function.
style	*string*	Metric color.

Gauge



General information

A “gauge" dashboard element displays aggregate data from multiple system records in the form of a dial with green, yellow and red areas on its scale. For example, you may use this dashboard to display a number of performed activities and compare it to a desired rate.

(Fig. 1). A gauge dashboard

Gauge functionality implementation classes

GaugeViewModel – gauge view model.

GaugeViewConfig – generates the gauge view model.

GaugeModule – module designed to work with gauges.

GaugeChart – implements a gauge chart component.

GaugeDesigner – view model schema of a gauge.

Gauge settings

To configure a gauge, you need to add the JSON configuration object with the gauge properties to the widget module configuration. The widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “**Dashboard widgets**” article.

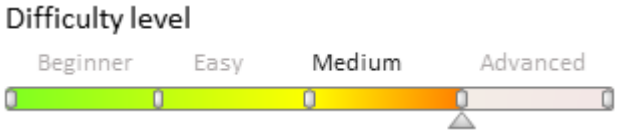
Set the “Gauge” value to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible gauge parameters are listed in table 1.

Table 1. Gauge settings

Name	Type	Details
caption	*string*	Gauge header.
sectionId	*string*	Section id.
entitySchemaName:	*string*	Gauge object.

sectionBindingColumn	*string*	Section link column.

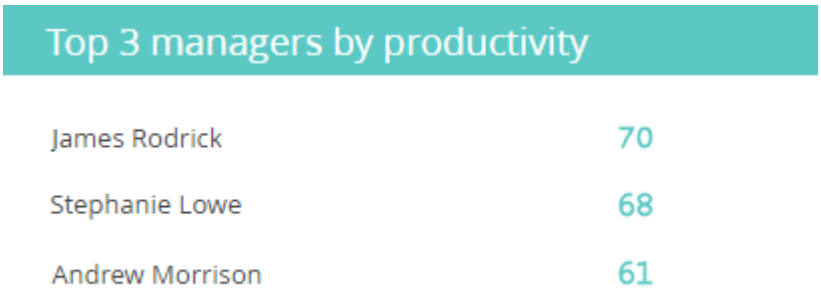
Lists



Introduction

A list displays multiple system records in a unified visual form. Lists enable you to limit the number of records displayed to create such dashboards as the “Top ten most productive managers by the number of closed deals”, for example. Learn more about the lists in the The “List” dashboard component article. List settings are described in the How to set up the “List” dashboard component article.

Fig. 1. A “list” dashboard



List functional classes

DashboardGridViewModel – list view model.

DashboardGridViewConfig – generates list view configuration.

DashboardGridModule – module designed to work with lists.

DashboardGridDesigner – list editing page schema.

List settings

To configure a list, you need to add the JSON configuration object with list properties to the widget module configuration. The widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “**Dashboard widgets**” article.

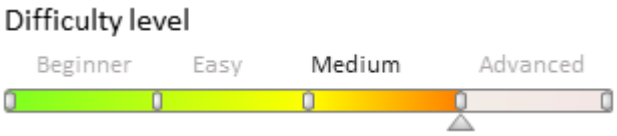
Set the “DashboardGrid” value of to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible list parameters are listed in table 1.

Table 1. List settings

Name	Type	Details
caption	*string*	List header.
sectionBindingColumn	*string*	Section link column.
filterData	*object*	Filter settings.
sectionId	*string*	Section id.
entitySchemaName:	*string*	List object.

style	*string*	List color.
orderDirection	*number*	Sorting options (1 - ascending, 2 - descending).
orderColumn	*string*	List sorting column.
rowCount	*number*	The number of rows to display.
gridConfig	*object*	List configuration.

Web-page



The “web-page” dashboard is used to display web-pages on the dashboard panel. It may be an online currency calculator, your corporate website, etc. Web-page dashboards are described in the “Setting up the “Web-page” dashboard” article.

Web-page functionality implementation classes

WebPageViewModel – web-page view model.

WebPageViewConfig – generates the web-page view model configuration.

WebPageModule – module used to work with web-pages.

WebPageDesigner – web-page widget view schema.

Web-page settings parameters

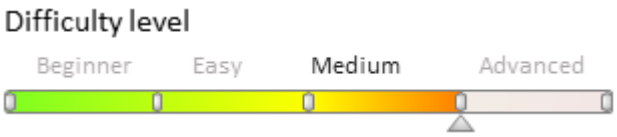
To configure a web-page, you need to add the JSON configuration object with web-page properties to the widget module configuration. Widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “**Dashboard widgets**” article.

Set the “WebPage” value to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible web-page parameters are listed in table 1.

Table 1. Web-page settings

Name	Type	Details
caption	*string*	Web-page widget title.
sectionId	*string*	Section id.
url	*string*	Web-page link.
style	*string*	Web-page widget CSS-styles

Sales pipeline



The “Sales pipeline” dashboard is used to analyze sales dynamics by stages. Learn more about the sales pipeline dashboard settings in the “Setting up the “Sales pipeline” dashboard component” article.

Sales pipeline functionality implementation classes

OpportunityFunnelChart – a class inherited from *Chart*.

Sales pipeline settings

To configure a sales pipeline, you need to add the JSON configuration object with sales pipeline properties to the widget module configuration. Widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “**Dashboard widgets**” article.

Set the “OpportunityFunnel” value to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible sales pipeline parameters are listed in table 1.

Table 1. Sales pipeline settings

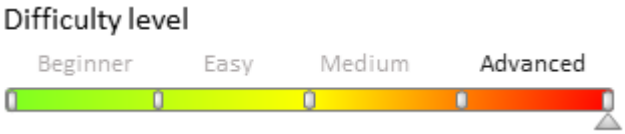
Name	Type	Details
caption	*string*	Sales pipeline header.
sectionId	*string*	Section id.
defPeriod	*string*	Pipeline period (last week by default).
sectionBindingColumn	*string*	Section link column.
type	*string*	Chart type (“funnel”).
filterData	*object*	Filter settings.

Scheduler setup

Contents

- **Recommendations on scheduler setup**
- **Quartz policies for the processing of overdue tasks**

Recommendations on scheduler setup



Selecting the Quartz policies for the processing of overdue tasks

All Quartz policies for the processing of overdue tasks can be divided into three groups: *Ignore misfire policy*, *Run immediately and continue* and *Discard and wait for next*. Recommendations about using each specific policy are given below.

Ignore misfire policy

This policy is represented by the *MisfireInstruction.IgnoreMisfirePolicy = -1* constant. It is recommended to use it when it is necessary to ensure that all trigger firings will be executed even with overdue tasks. For example, the task A with 2 minutes execution periodicity. Due to lack of Quartz threads or the scheduler shutdown, the next fire time of task A (NEXT_FIRE_TIME) lags 10 minutes from the current time. If execution of all 5 overdue tasks is required, use the *IgnoreMisfirePolicy* utility.

This policy is recommended to use for triggers that operate with a unique data at each fire and it is important to perform all trigger fires. For example, the task B that is executed once per 1 hour and generates the report for the time period from PREV_FIRE_TIME till PREV_FIRE_TIME + 1 hour. The scheduler was turned off for 8 hours. In this case, all 8 fires of the task B should be performed after launch of the scheduler and all reports should be generated.

Applying this policy to triggers that do not operate with unique data may cause to unnecessary clogging of the scheduler queue and application performance decrease. For example, the Exchange email synchronization is configured with the 1 minute interval for each bpm'online user. An update was performed for 1.5 hours. After the update, the Quartz will synchronize user mailboxes 90 times before proceeding with tasks scheduled for the current time. Although it is enough to perform the delayed synchronization of mailboxes once, and then proceed the task according to the schedule.

Run immediately and continue

This group includes:

- *SimpleTrigger.FireNow*;
- *SimpleTrigger.RescheduleNowWithExistingRepeatCount*;
- *SimpleTrigger.RescheduleNowWithRemainingRepeatCount;*
- *CronTrigger.FireOnceNow*;
- *CalendarIntervalTrigger.FireOnceNow*.

More information about these policies can be found in the "**Quartz policies for the processing of overdue tasks**" article.

This policies should be used if the overdue task should be executed one time and as a priority and then execute scheduled tasks. For example, the email synchronization (<user>@<server>_LoadExchangeEmailsProcess_<userId>, SyncImap_<user>@<server>_<userId>) or the RemindingCountersJob and SyncWithLDAPProcess tasks.

For example, email synchronization is configured with the 5 minutes interval for all users and is fired by the <user>@<server>_LoadExchengeEmailProcess_<userId>Trigger triggers. The update was performed since 1:30 AM till 2:43 AM. After update, the next fire time for the <user>@<server>_LoadExchengeEmailsProcess_<userId>Trigger triggers will be changed to 2:43 AM. All overdue tasks will be fired once at 2:43 AM and further will be fired according to the schedule (2,48 AM, 2:53 AM, 2:58 Am, etc.).

Discard and wait for next

This group includes:

- *SimpleTrigger.RescheduleNextWithRemainingCount*;
- *SimpleTrigger.RescheduleNextWithExistingCount*;
- *CronTrigger.DoNothing*;
- *CalendarIntervalTrigger.DoNothing*.

More information about these policies can be found in the "**Quartz policies for the processing of overdue tasks**" article.

These policies should be applied to the tasks that should be fired strictly at a specific time. For example the statistics collection launched daily at 3:00 AM when there is no active users on the website (the *CronTrigger* is used). This task is resource intensive and time consuming, and it can not be run during working hours, because this can slow down the work of users. In this case, use the *CronTrigger.DoNothing* policy. As a result, if the task was not fired, the next fire will be at 3:00 AM of the next day.

Quartz configuration

thread count

If the scheduler delays the tasks or if some tasks have not been executed, increase the number of Quartz threads. For this, set necessary number of threads in the Web.config of the application loader:

```
<add key="quartz.threadPool.threadCount" value="5" />
```

> **Note**
>
> The Web.config file of the application loader located in the root folder of the installed bpm'online application.

misfireThreshold

If the increase in the number of Quartz threads is undesirable (for example, due to limited resources), the change in the misfireThreshold setting in the Web.config file of the application loader can optimize the task execution.

For example, an application with a number of tasks much bigger than a number of threads. The most of tasks are executed with a small interval (1 minute). The value of the *misfireThreshold* setting is 1 minute and the number of threads is 3:

```
<add key="quartz.jobStore.misfireThreshold" value="60000" />
<add key="quartz.threadPool.threadCount" value="3" />
```

For the most of the tasks used the policies from the *Run immediately and continue* group. This means following. For the most tasks that have the MISFIRE_INSTR not equal "-1" and the NEXT_FIRE_TIME less than the current time for 1 minute (60000 ms) the Quartz will periodically set the time of the next fire for the current time. This means that the initial order of the scheduled tasks will be lost because all tasks will have the current time as the fire time. The probability that Quartz will often process the same tasks and ignore the other tasks will increase.

The scheduler queue after 15 minutes of working is displayed on the Fig. 1. Tasks that have the PREV_FIRE_TIME = NULL never have been executed. There is a big number of these tasks.

Fig. 1. Scheduler queue with the misfireThreshold, = 1 minute

TRIGGER_NAME	TRIGGER_GROUP	NEXT_FIRE_TIME	PREV_FIRE_TIME	Last repeat interval, min	TRIGGER_STATE	TRIGGER_TYPE	START_TIME	MISFIRE_INSTR
MinutelyJob_15Trigger	TestGroup	2017-11-21 19:17:20.860	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.860	3
MinutelyJob_16Trigger	TestGroup	2017-11-21 19:17:20.863	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.863	3
MinutelyJob_17Trigger	TestGroup	2017-11-21 19:17:20.867	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.867	3
MinutelyJob_18Trigger	TestGroup	2017-11-21 19:17:20.867	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.867	3
MinutelyJob_19Trigger	TestGroup	2017-11-21 19:17:20.870	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.870	3
SyncImap_postulga.alexey@gmail.com_7f3b869f-34f3...	IMAP	2017-11-21 19:17:20.870	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.870	3
MinutelyJob_5Trigger	TestGroup	2017-11-21 19:17:20.873	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.873	3
Terrasoft.Configuration.CampaignFailoverHandlerTrigger	CampaignJobGroup	2017-11-21 19:17:20.873	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.873	3
Terrasoft.Configuration.TriggerEmailFailoverHandlerTri...	Mailing	2017-11-21 19:17:20.877	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.877	3
Terrasoft.Configuration.BulkEmailFailoverHandlerTrigger	Mailing	2017-11-21 19:17:20.880	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.880	3
MinutelyJob_6Trigger	TestGroup	2017-11-21 19:17:20.880	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.880	3
MinutelyJob_7Trigger	TestGroup	2017-11-21 19:17:20.883	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.883	3
MinutelyJob_0Trigger	TestGroup	2017-11-21 19:17:20.883	2017-11-21 19:04:37.643	12.72065943000	WAITING	SIMPLE	2017-11-21 19:17:20.883	3
MinutelyJob_1Trigger	TestGroup	2017-11-21 19:17:20.887	2017-11-21 19:04:37.663	12.72035926333	WAITING	SIMPLE	2017-11-21 19:17:20.887	3
MinutelyJob_2Trigger	TestGroup	2017-11-21 19:17:20.887	2017-11-21 19:04:37.667	12.72034262833	WAITING	SIMPLE	2017-11-21 19:17:20.887	3
CESWebHooksSyncTrigger	Mailing	2017-11-21 19:17:20.890	2017-11-21 19:08:20.733	9.00257776333	WAITING	CAL_INT	2017-11-21 18:39:38.140	1
MandrillScheduledMailingTrigger	Mailing	2017-11-21 19:17:20.890	2017-11-21 19:08:20.740	9.00252771833	WAITING	CAL_INT	2017-11-21 18:39:38.197	1
MT_2874667341633338083	DEFAULT	2017-11-21 19:18:20.857	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:08:42.410	0
Terrasoft.Configuration.ML.MLModelTrainerJob, Terra...	DEFAULT	2017-11-21 19:18:20.860	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:08:42.433	0
Terrasoft.Configuration.EmailMining.EmailMiningJob, T...	DEFAULT	2017-11-21 19:18:20.860	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:08:42.450	0
Terrasoft.Configuration.NotificationsJobTrigger	NotificationsCountersGroup	2017-11-21 19:18:20.863	2017-11-21 19:08:20.727	10.00226696166	WAITING	SIMPLE	2017-11-21 19:18:20.863	3
A.Postulga@terrasoft.ru_LoadExchangeEmailsProces...	Exchange	2017-11-21 19:18:20.863	2017-11-21 19:08:20.743	10.00205025333	WAITING	SIMPLE	2017-11-21 19:18:20.863	3
MinutelyJob_8Trigger	TestGroup	2017-11-21 19:18:20.867	2017-11-21 19:08:20.743	10.00205029000	WAITING	SIMPLE	2017-11-21 19:18:20.867	3
MinutelyJob_9Trigger	TestGroup	2017-11-21 19:18:20.870	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:18:20.870	3
MinutelyJob_10Trigger	TestGroup	2017-11-21 19:18:20.870	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:18:20.870	3
MinutelyJob_11Trigger	TestGroup	2017-11-21 19:18:20.873	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:18:20.873	3
MinutelyJob_12Trigger	TestGroup	2017-11-21 19:18:20.873	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:18:20.873	3
c999bb9e-070e-411f-8d19-181e6393b6e8	DEFAULT	2017-11-21 19:18:25.517	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:08:25.517	0
RemindingCountersJob_7f3b869f-34f3-4f20-ab4d-748...	RemindingCountersGroup	2017-11-21 19:18:48.613	2017-11-21 19:17:48.613	1.00000000000	BLOCKED	CAL_INT	2017-11-21 19:08:48.613	0
MinutelyJob_3Trigger	TestGroup	2017-11-21 19:19:20.723	2017-11-21 19:18:20.723	1.00000000000	BLOCKED	SIMPLE	2017-11-21 19:08:20.723	3
MinutelyJob_4Trigger	TestGroup	2017-11-21 19:19:20.723	2017-11-21 19:18:20.723	1.00000000000	BLOCKED	SIMPLE	2017-11-21 19:08:20.723	3
Terrasoft.Configuration.DelayedNotifyingTrigger	DelayedNotificationGroup	2017-11-21 19:23:20.697	2017-11-21 19:18:20.697	5.00000000000	WAITING	SIMPLE	2017-11-21 19:08:20.697	3
SyncWithLDAPProcessTrigger	LDAP	2017-11-21 19:43:35.327	NULL	NULL	WAITING	CAL_INT	2017-11-21 18:43:35.327	0
NotificationCleanerTrigger	NotificationCleanerGroup	2017-11-22 02:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 18:39:37.000	0
ActualizeActiveContactsTrigger	Mailing	2017-11-22 02:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 18:39:37.997	1
GenerateAnniversaryRemindingsTrigger	GenerateAnniversaryRem...	2017-11-22 03:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 19:08:25.000	0
Terrasoft.Configuration.DelayedNotificationCleaningTri...	DelayedNotificationGroup	2017-11-22 18:39:37.843	NULL	NULL	WAITING	SIMPLE	2017-11-21 18:39:37.843	3

Increase the *misfireThreshold* value to 10 minutes (and clear the PREV_FIRE_TIME в QRTZ_TRIGGERS):

```
<add key="quartz.jobStore.misfireThreshold" value="600000" />
```

The scheduler queue after 15 minutes of working is displayed on the Fig. 2.

Fig. 2. Scheduler queue with the misfireThreshold, = 10 minutes

TRIGGER_NAME	TRIGGER_GROUP	NEXT_FIRE_TIME	PREV_FIRE_TIME	Last repeat interval, min	TRIGGER_STATE	TRIGGER_TYPE	START_TIME	MISFIRE_INSTR
MinutelyJob_13Trigger	TestGroup	2017-11-21 19:23:20.960	2017-11-21 19:22:20.960	1.00000000000	WAITING	SIMPLE	2017-11-21 19:20:20.960	3
Terrasoft.Configuration.TriggerEmailFailoverHandlerTri...	Mailing	2017-11-21 19:25:20.980	2017-11-21 19:20:20.980	5.00000000000	WAITING	SIMPLE	2017-11-21 19:20:20.980	3
Terrasoft.Configuration.BulkEmailFailoverHandlerTrigger	Mailing	2017-11-21 19:25:20.980	2017-11-21 19:20:20.980	5.00000000000	WAITING	SIMPLE	2017-11-21 19:20:20.980	3
RemindingCountersJob_7f3b869f-34f3-4f20-ab4d-748...	RemindingCountersGroup	2017-11-21 19:25:50.787	2017-11-21 19:24:50.787	1.00000000000	BLOCKED	CAL_INT	2017-11-21 19:22:50.787	0
MinutelyJob_3Trigger	TestGroup	2017-11-21 19:26:20.723	2017-11-21 19:25:20.723	1.00000000000	BLOCKED	SIMPLE	2017-11-21 19:08:20.723	3
MinutelyJob_4Trigger	TestGroup	2017-11-21 19:26:20.723	2017-11-21 19:25:20.723	1.00000000000	BLOCKED	SIMPLE	2017-11-21 19:08:20.723	3
Terrasoft.Configuration.DelayedNotifyingTrigger	DelayedNotificationGroup	2017-11-21 19:28:20.697	2017-11-21 19:23:20.697	5.00000000000	WAITING	SIMPLE	2017-11-21 19:08:20.697	3
SyncImap_postulga.alexey@gmail.com_7f3b869f-34f3...	IMAP	2017-11-21 19:32:39.110	2017-11-21 19:20:20.973	12.30228216333	WAITING	SIMPLE	2017-11-21 19:32:39.110	3
MinutelyJob_5Trigger	TestGroup	2017-11-21 19:32:39.117	2017-11-21 19:20:20.977	12.30234882000	WAITING	SIMPLE	2017-11-21 19:32:39.117	3
MinutelyJob_6Trigger	TestGroup	2017-11-21 19:32:39.117	2017-11-21 19:20:20.983	12.30224896000	WAITING	SIMPLE	2017-11-21 19:32:39.117	3
MinutelyJob_7Trigger	TestGroup	2017-11-21 19:32:39.120	2017-11-21 19:20:20.983	12.30226555000	WAITING	SIMPLE	2017-11-21 19:32:39.120	3
MinutelyJob_0Trigger	TestGroup	2017-11-21 19:32:39.120	2017-11-21 19:20:20.987	12.30224884166	WAITING	SIMPLE	2017-11-21 19:32:39.120	3
MinutelyJob_1Trigger	TestGroup	2017-11-21 19:32:39.123	2017-11-21 19:20:20.987	12.30224826833	WAITING	SIMPLE	2017-11-21 19:32:39.123	3
MinutelyJob_2Trigger	TestGroup	2017-11-21 19:32:39.123	2017-11-21 19:20:20.990	12.30224890666	WAITING	SIMPLE	2017-11-21 19:32:39.123	3
Terrasoft.Configuration.NotificationsJobTrigger	NotificationsCountersGroup	2017-11-21 19:32:39.127	2017-11-21 19:20:21.057	12.30116543500	WAITING	SIMPLE	2017-11-21 19:32:39.127	3
A.Postulga@terrasoft.ru_LoadExchangeEmailsProces...	Exchange	2017-11-21 19:32:39.130	2017-11-21 19:20:21.060	12.30116547833	WAITING	SIMPLE	2017-11-21 19:32:39.130	3
MinutelyJob_8Trigger	TestGroup	2017-11-21 19:32:39.130	2017-11-21 19:20:21.060	12.30116555333	WAITING	SIMPLE	2017-11-21 19:32:39.130	3
MinutelyJob_9Trigger	TestGroup	2017-11-21 19:32:39.133	2017-11-21 19:20:21.063	12.30116544666	WAITING	SIMPLE	2017-11-21 19:32:39.133	3
MinutelyJob_10Trigger	TestGroup	2017-11-21 19:32:39.133	2017-11-21 19:20:21.067	12.30116546166	WAITING	SIMPLE	2017-11-21 19:32:39.133	3
MinutelyJob_11Trigger	TestGroup	2017-11-21 19:32:39.137	2017-11-21 19:20:21.067	12.30116550166	WAITING	SIMPLE	2017-11-21 19:32:39.137	3
MinutelyJob_12Trigger	TestGroup	2017-11-21 19:32:39.140	2017-11-21 19:20:21.070	12.30118216666	WAITING	SIMPLE	2017-11-21 19:32:39.140	3
CESWebHooksSyncTrigger	Mailing	2017-11-21 19:32:39.143	2017-11-21 19:20:38.140	12.01673093833	WAITING	CAL_INT	2017-11-21 18:39:38.140	1
MandrillScheduledMailingTrigger	Mailing	2017-11-21 19:32:39.143	2017-11-21 19:20:38.197	12.01579751166	WAITING	CAL_INT	2017-11-21 18:39:38.197	1
MinutelyJob_14Trigger	TestGroup	2017-11-21 19:32:39.147	2017-11-21 19:21:20.963	11.30308236666	WAITING	SIMPLE	2017-11-21 19:32:39.147	3
MinutelyJob_15Trigger	TestGroup	2017-11-21 19:32:39.150	2017-11-21 19:21:20.963	11.30308227500	WAITING	SIMPLE	2017-11-21 19:32:39.150	3
MinutelyJob_16Trigger	TestGroup	2017-11-21 19:32:39.150	2017-11-21 19:21:20.967	11.30308224833	WAITING	SIMPLE	2017-11-21 19:32:39.150	3
MinutelyJob_17Trigger	TestGroup	2017-11-21 19:32:39.153	2017-11-21 19:21:20.967	11.30308234666	WAITING	SIMPLE	2017-11-21 19:32:39.153	3
MinutelyJob_18Trigger	TestGroup	2017-11-21 19:32:39.210	2017-11-21 19:21:20.970	11.30399938833	WAITING	SIMPLE	2017-11-21 19:32:39.210	3
MinutelyJob_19Trigger	TestGroup	2017-11-21 19:32:39.213	2017-11-21 19:21:20.970	11.30401564500	WAITING	SIMPLE	2017-11-21 19:32:39.213	3
MT_520821574169140644	DEFAULT	2017-11-21 19:32:39.217	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:22:30.010	0
Terrasoft.Configuration.ML.MLModelTrainerJob, Terra...	DEFAULT	2017-11-21 19:32:39.227	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:22:30.037	0
Terrasoft.Configuration.EmailMining.EmailMiningJob, T...	DEFAULT	2017-11-21 19:32:39.227	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:22:30.047	0
Terrasoft.Configuration.CampaignFailoverHandlerTrigger	CampaignJobGroup	2017-11-21 19:35:20.977	2017-11-21 19:20:20.977	15.00000000000	WAITING	SIMPLE	2017-11-21 19:20:20.977	3
0778a81c-ef91-402a-826a-77ab0e3afb25	DEFAULT	2017-11-21 19:37:25.223	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:22:25.223	0
SyncWithLDAPProcessTrigger	LDAP	2017-11-21 19:43:35.327	NULL	NULL	WAITING	CAL_INT	2017-11-21 18:43:35.327	0
NotificationCleanerTrigger	NotificationCleanerGroup	2017-11-22 02:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 18:39:37.000	0
ActualizeActiveContactsTrigger	Mailing	2017-11-22 02:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 18:39:37.997	1
GenerateAnniversaryRemindingsTrigger	GenerateAnniversaryRem...	2017-11-22 03:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 19:22:25.000	0
Terrasoft.Configuration.DelayedNotificationCleaningTri...	DelayedNotificationGroup	2017-11-22 18:39:37.843	NULL	NULL	WAITING	SIMPLE	2017-11-21 18:39:37.843	3

The number of non-executed tasks is decreased.

Increasing of the value of the *misfireThreshold* will lead to more equal tasks execution. Almost all jobs from the queue will be executed. Due to lack of threads, the scheduler does not have time to fire each of the tasks in a minute. This is displayed in the [Last repeat interval] column that has the value NEXT_FIRE_TIME - PREV_FIRE_TIME, min. However, the scheduler fires each of the tasks.

batchTriggerAcquisitionMaxCount

Increase the *batchTriggerAcquisitionMaxCount* to optimize the scheduler performance if you do not use the clustered Quartz configuration (one scheduler node is used).

Quartz policies for the processing of overdue tasks

Difficulty level

Introduction

The Quartz has policies common to all types of triggers, and policies that are specific to a specific type of trigger. All policies used for the *SimpleTrigger*, *CronTrigger* or *CalendarIntervalTrigger* triggers are listed in the Table 1.

Table 1. Trigger policies

Quartz policy	The MISFIRE_INSTR value	The Terrasoft.Core.Scheduler.AppSchedulerMisfireInstruction value	Trigger type
IgnoreMisfirePolicy	-1	IgnoreMisfirePolicy	for all types

***IgnoreMisfirePolicy* behavior description**

Triggers with the IgnoreMisfirePolicy always will be fired in time. For such triggers, the Quartz will not update the next fire time (NEXT_FIRE_TIME).

The Quartz will fire all overdue tasks as priority, and then return to the initial trigger schedule. For example, the task with the Simple Trigger was planned for 10 iterations. Initial conditions:

- START_TIME = 9:00
- REPEAT_COUNT = 9 (first fire + 9 iterations)
- REPEAT_INTERVAL = 0:15.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will try to fire 2 overdue tasks as priority (in 9:00 and 9:15). After that the Quartz fires the 8 remaining tasks according to the

schedule (at 9:30, 9:45, etc.).

SmartPolicy	0	*SmartPolicy*	for all types

***SmartPolicy* behavior description**

By default is used by Quartz for all types of triggers. According to the type and configuration of the trigger, the Quartz will select corresponding policy. The selection algorithm for the Quartz version 2.3.2 is shown in the pseudo-code below.

```
if (TRIGGER_TYPE == 'SIMPLE') // Simple trigger.
    if (REPEAT_COUNT == 0) // Without repeats.
        MISFIRE_INSTR = 1 // SimpleTrigger.FireNow
    else if (REPEAT_COUNT == -1) // COntinious repeating.
        MISFIRE_INSTR = 4 // SimpleTrigger.RescheduleNextWithRemainingCount
    else // The number of repetitions is indicated.
        MISFIRE_INSTR = 2 // SimpleTrigger.RescheduleNowWithExistingRepeatCount
else if (TRIGGER_TYPE == 'CAL_INT') // CalendarInterval trigger.
    MISFIRE_INSTR = 1 // CalendarIntervalTrigger.FireOnceNow
else if (TRIGGER_TYPE == 'CRON') // Cron trigger.
    MISFIRE_INSTR = 1 // CronTrigger.FireOnceNow
```

SimpleTrigger.FireNow	1	*FireNow*	SimpleTrigger

***SimpleTrigger.FireNow* behavior description**

Applied for the *SimpleTrigger* triggers that have the REPEAT_COUNT=0 (triggers fired for one time). If the REPEAT_COUNT value is not "0", then the *SimpleTrigger.RescheduleNowWithRemainingRepeatCount* policy will be applied.

For example, the task planned with the *SimpleTrigger*. Initial conditions:

- START_TIME = 9:00;
- REPEAT_COUNT = 0.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will try to fire the task as priority. As a result, the task will be fired at 9:20 or later.

SimpleTrigger.RescheduleNowWithExistingRepeatCount	2	RescheduleNowWithExistingRepeatCount	SimpleTrigger

***SimpleTrigger.RescheduleNowWithExistingRepeatCount* behavior description**

Scheduler will try to fire the first overdue task as a priority, All other triggers will be fired with the REPEAT_INTERVAL interval.

For example, the task with the Simple Trigger was planned for 10 iterations. Initial conditions:

- START_TIME = 9:00
- REPEAT_COUNT = 9
- REPEAT_INTERVAL = 0:15.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will try to fire first overdue task scheduled for 9:00 (from tasks scheduled on 9:00 and 9:15) at 9:20. The remained 9 tasks will be fired at 9:35, 9:50, etc.

> NOTE
>
> For the *AppScheduler.ScheduleMinutelyJob* methods the behavior of the *RescheduleNowWithExistingRepeatCount* is similar to the *RescheduleNowWithRemainingRepeatCount*, and the behavior of the *RescheduleNextWithRemainingCount* is similar to the *RescheduleNextWithExistingCount*, because the triggers with the REPEAT_COUNT = -1 are used.

SimpleTrigger.RescheduleNowWithRemainingRepeatCount	3	RescheduleNowWithRemainingRepeatCount	SimpleTrigger

***SimpleTrigger.RescheduleNowWithRemainingRepeatCount* behavior description**

Scheduler will try to fire the first overdue task as a priority, Other overdue tasks are ignored. The scheduler fires remained tasks that are not overdue with the REPEAT_INTERVAL interval.

For example, the task with the Simple Trigger was planned for 10 iterations. Initial conditions:

- START_TIME = 9:00
- REPEAT_COUNT = 9
- REPEAT_INTERVAL = 0:15.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will try to fire first overdue task (from tasks scheduled on 9:00 and 9:15) at 9:20. The second overdue task will be ignored and other 8 tasks will be fired at 9:35, 9:50, etc.

SimpleTrigger.RescheduleNextWithRemainingCount	4	RescheduleNextWithRemainingCount	SimpleTrigger

***SimpleTrigger.RescheduleNextWithRemainingCount* behavior description**

The scheduler ignores overdue tasks and waits for the next planned fire of the task. At the next fire time, the remaining non-overdue tasks will be executed with the REPEAT_INTERVAL interval.

For example, the task with the Simple Trigger was planned for 10 iterations. Initial conditions:

- START_TIME = 9:00
- REPEAT_COUNT = 9
- REPEAT_INTERVAL = 0:15.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will fire the rest of 8 non-overdue tasks at 9:30, 9:45, etc.

SimpleTrigger.RescheduleNextWithExistingCount	5	RescheduleNextWithExistingCount	SimpleTrigger

***SimpleTrigger.RescheduleNextWithExistingCount* behavior description**

The scheduler will wait for the next launch time and will fire all remained tasks with the REPEAT_INTERVAL interval.

For example, the task with the Simple Trigger was planned for 10 iterations. Initial conditions:

- START_TIME = 9:00
- REPEAT_COUNT = 9
- REPEAT_INTERVAL = 0:15.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will fire all 10 tasks at 9:30, 9:45, etc.

CronTrigger.FireOnceNow	1	-	CronTrigger

***CronTrigger.FireOnceNow* behavior description**

Scheduler will try to fire the first overdue task as a priority. Other overdue tasks are ignored. Remained non-overdue tasks are fired by the scheduler according to the schedule.

For example, the task planned with the CronTrigger: CRON_EXPRESSION = '0 0 9-17 ? * MON-FRI' (from Monday to Friday 9:00 AM – 17:00 PM). If the scheduler was disabled from 8:50 till 10:20, then after launch at 10:20 the Quartz will try to fire first overdue task from two (at 9:00 and 10:00). After that, tasks will be fired at 11:00, 12:00, etc.

CronTrigger.DoNothing	2	-	CronTrigger

***CronTrigger.DoNothing* behavior description**

Scheduler ignores all overdue tasks. Remained non-overdue tasks will be fired according to the schedule.

For example, the task planned with the CronTrigger: CRON_EXPRESSION = '0 0 9-17 ? * MON-FRI' (from Monday to Friday 9:00 AM – 17:00 PM). If the scheduler was disabled from 8:50 till 10:20, then after launch the Quartz will start to fire tasks since 11:00 (at 11:00, 12:00, etc).

CalendarIntervalTrigger.FireOnceNow	1	-	CalendarIntervalTrigger

***CalendarIntervalTrigger* behavior description**

The behavior is similar to the *CronTrigger.FireOnceNow*.

CalendarIntervalTrigger.DoNothing	2	-	CalendarIntervalTrigger

***CalendarIntervalTrigger.DoNothing* behavior description**

The behavior is similar to the *CronTrigger.DoNothing*.

Integration

Contents

- **Phone integration**
- **Email integration**

Phone integration

Difficulty level

General provisions

Bpm'online can be integrated with a number of automatic telephone exchanges (Private Branch Exchange, PBX), which enables users to manage calls directly in bpm'online UI. Phone integration functions are available in the form of a CTI (Computer Telephony Integration) panel, as well as the [Calls] section. Standard CTI panel functions:

- Displaying incoming calls with contact/account identification by the subscriber's phone number
- One-click calls initiated from bpm'online UI
- Call management (reply, place on hold, end or transfer call)
- Displaying call history for managing connections of calls to various system records and call follow-up.

All calls made or received are stored in the [Calls] section. In this section, you can view when a call was started, when it ended and how long the call was; as well as the list of system records connected to the call.

By default, bpm'online cloud has a function for making calls between system users without using any additional software.

Depending on the integrated phone system and specifics of its API (Application Programming Interface), different architectural mechanisms are used. The API also affects available phone integration functions. For example, the call playback function is not available for all phone systems, the web phone is available when integrating with Webitel, etc. Regardless of the phone integration mechanism being used, the CTI panel interface remains the same for all bpm'online users.

Phone integration methods in bpm'online

There are two types of integration methods: *first party* and *third party* integrations.

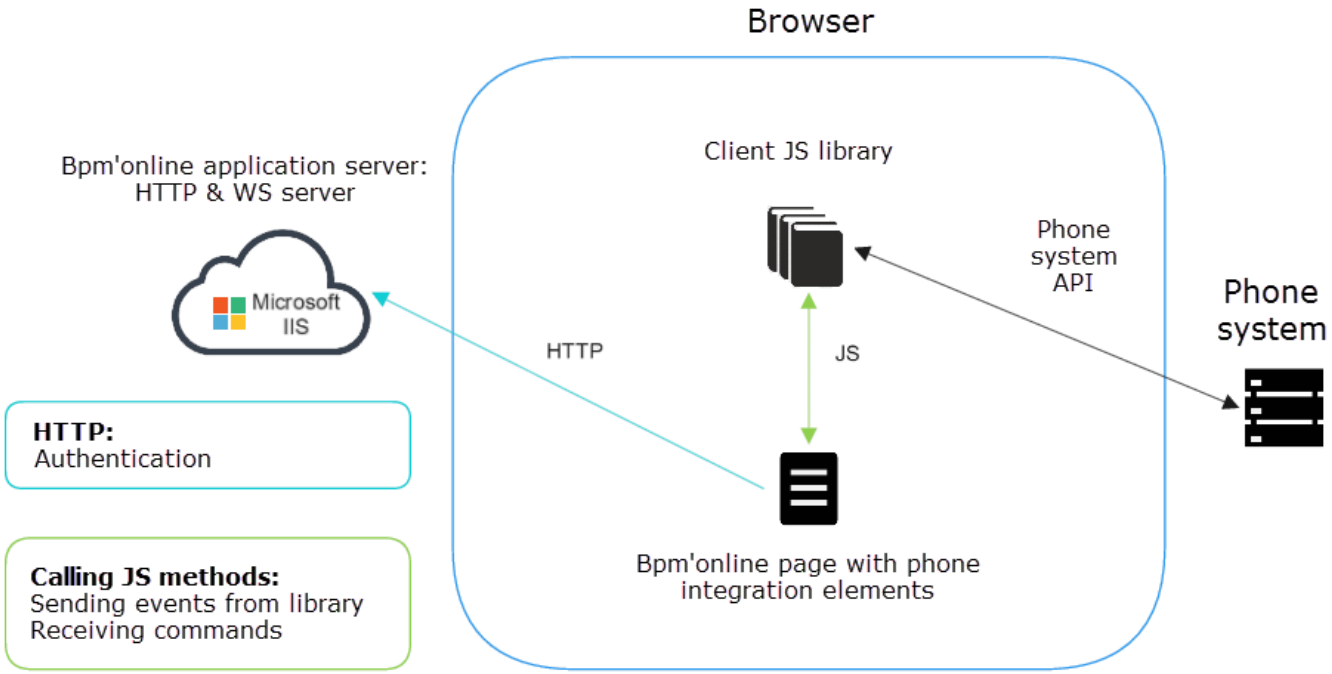
In a *first party* integration each user has a separate integration connection. Phone system events are handled as part of that connection.

For a *third party* integration, a single connection to the prone system server is used for handling phone system events for all users. In a third party integration an intermediate *Messaging Service* link is used for distributing information streams for all users.

JavaScript adapter on the client side

When integrating with JavaScript adapter on the client side (Fig. 1), the work with the prone system is done directly from a web browser. Interactions with the phone system and JavaScript-library, usually supplied by the prone system manufacturer, is done through the phone system API. The library broadcasts events and accepts execution commands using JavaScript. In the context of this integration, the bpm'online page interacts with the application server for authentication using the HTTP(S) protocol.

Fig. 1. First party phone system API integration with a javascript adapter on the client side



This integration method can be used with a first party phone system API, such as Webitel, Oktell, Finesse. Webitel and Oktell connectors use WebSocket as connection protocol, while the Finesse connector uses long-polling http queries.

The advantage of the *first party* integration method is that it does not require any additional nodes, such as Messaging Service. Using an integration library, the CTI panel connects directly to the phone system server API from a browser on the user's PC (Fig. 1).

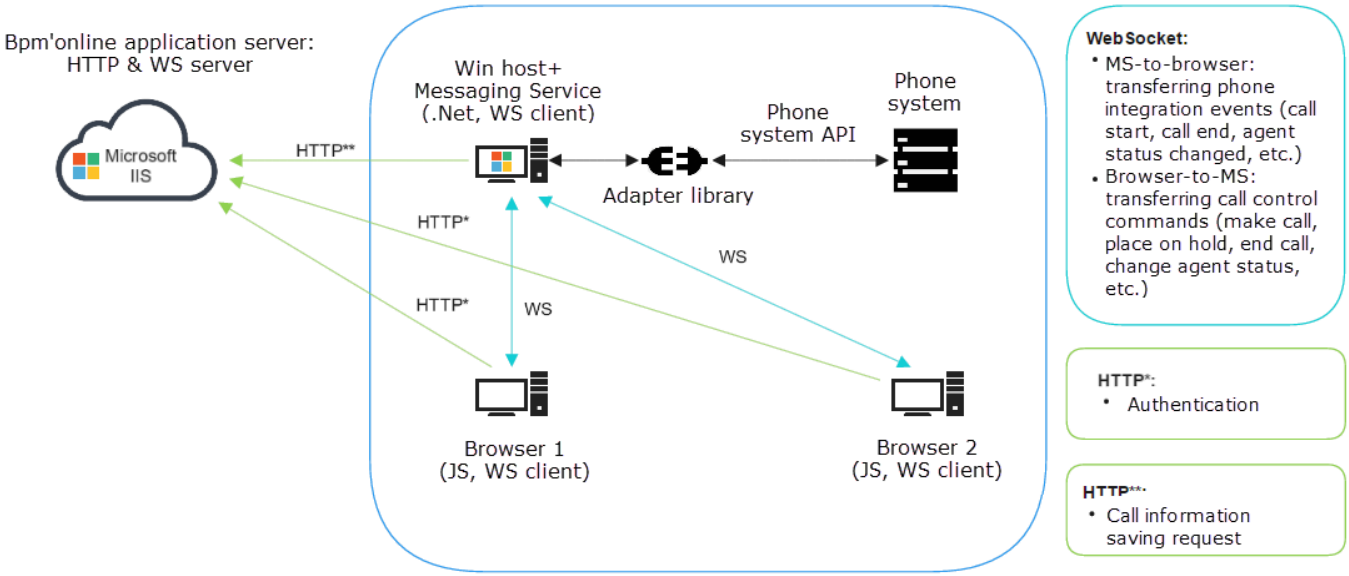
For incoming calls the phone server passes the new call start event and call parameters through WebSocket to the client integration library. When receiving a new call command, the library generates the *RingStarted* event that is passed to the application page.

For incoming calls, client part generates the call start command that is passed through WebSocket to the phone integration server.

Terrasoft Messaging Service on the server side

If integrating with Terrasoft Messaging Service (TMS) on the server side (Fig. 2), all phone integration events pass through TMS, which interacts with the phone system through the manufacturer's library. The library interacts with the phone system through the API. TMS also interacts with the bpm'online application server for executing query for saving call information in the database using HTTP(S). Interaction with a client application, such as passing events and receiving commands, is done via WebSocket. In case of integration with JavaScript adapter on the client side, bpm'online page interacts with the application server for authentication, using HTTP(S).

Fig. 2. Third party API integration with TMS on server side



This integration method applies to third party phone system API (TAPI, TSAPI, New Infinity protocol, WebSocket Oktell). This integration type requires *Messaging Service* – a Windows proxy service that works with the phone system adapter library. The *Messaging Service* is a universal phone system library hoster, such as Asterisk, Avaya, Callway, Ctios, Infinity, Infra, Tapi. When receiving client messages, the *Messaging Service* automatically connects used bpm'online library and initiates connection to phone system. The Messaging Service is essentially a functional wrapper for those phone integration connectors that do not support client integration for interacting with phone functions in browsers (event generation and handling, data transfer). A user's PC conducts two types of communication:

- HTTP connection with bpm'online application server for authentication with host on which the *Messaging Service* is installed
- WebSocket connection for working directly with phone integration (Fig. 2).

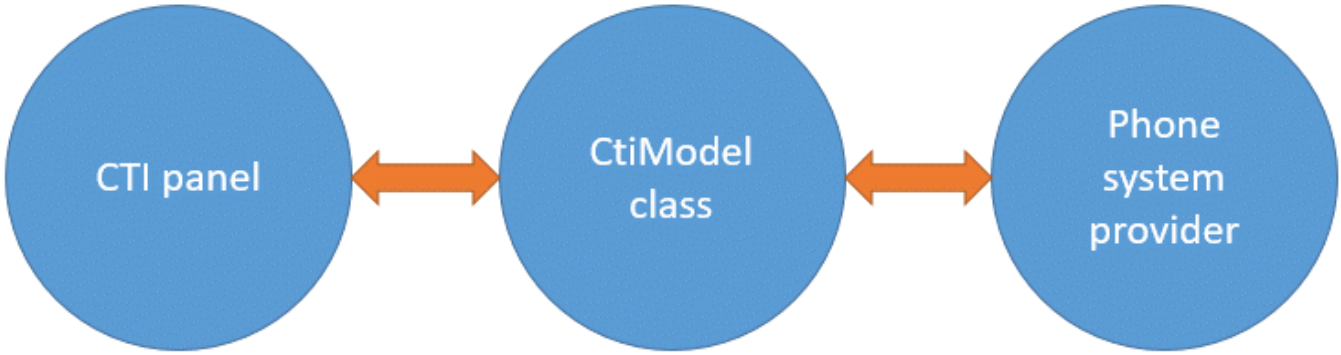
For incoming calls the phone system passes the new call start event and call parameters through the adapter library. When receiving a new call command, the Messaging Service generates the *RingStarted* event that is passed to the client.

For an outgoing calls, the client generates a call start command, which is passed via WebSocket to the Messaging Service, which generates an outgoing call message for the phone system.

Interaction between the phone connectors and bpm'online

All connectors interact with configuration through the CtiModel class. It handles the events received from the connector.

Fig. 3. Phone system component interaction with bpm'online

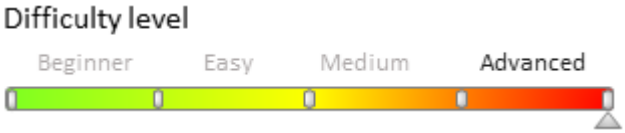


The list of supported class events is provided in table 1.

Table 1. Supported events of the CtiModel class

Event	Description
initialized	Triggered on completion of provider initialization.
disconnected	Triggered on provider disconnection.
callStarted	Triggered at the start of a new call.
callFinished	Triggered after call completion.
commutationStarted	Triggered after establishing call connection.
callBusy	Triggered on changing call status to "busy" (TAPI only).
hold	Triggered after placing call on hold.
unhold	Triggered after resuming a call.
error	is triggered on errors.
lineStateChanged	Triggered after changing available operations for a line or a call.
agentStateChanged	Triggered on changing the agent status.
activeCallSnapshot	Triggered on updating the list of active calls.
callSaved	Triggered after creating or updating a call in the database.
rawMessage	Generic provider event. Triggered on any provider event.
currentCallChanged	Triggered on changing the main call. For example, primary call ends during a consultation.
callCentreStateChanged	Triggered if an agent enters or exits Call center mode.
callInfoChanged	Triggered on modifying a call data by database Id.
dtmfEntered	Triggered if Dtmf signals were sent to the phone line.
webRtcStarted	Triggered on a webRtc session start.
webRtcVideoStarted	Triggered on a webRtc video stream session start.
webRtcDestroyed	Triggered on a webRtc session end.

Oktell

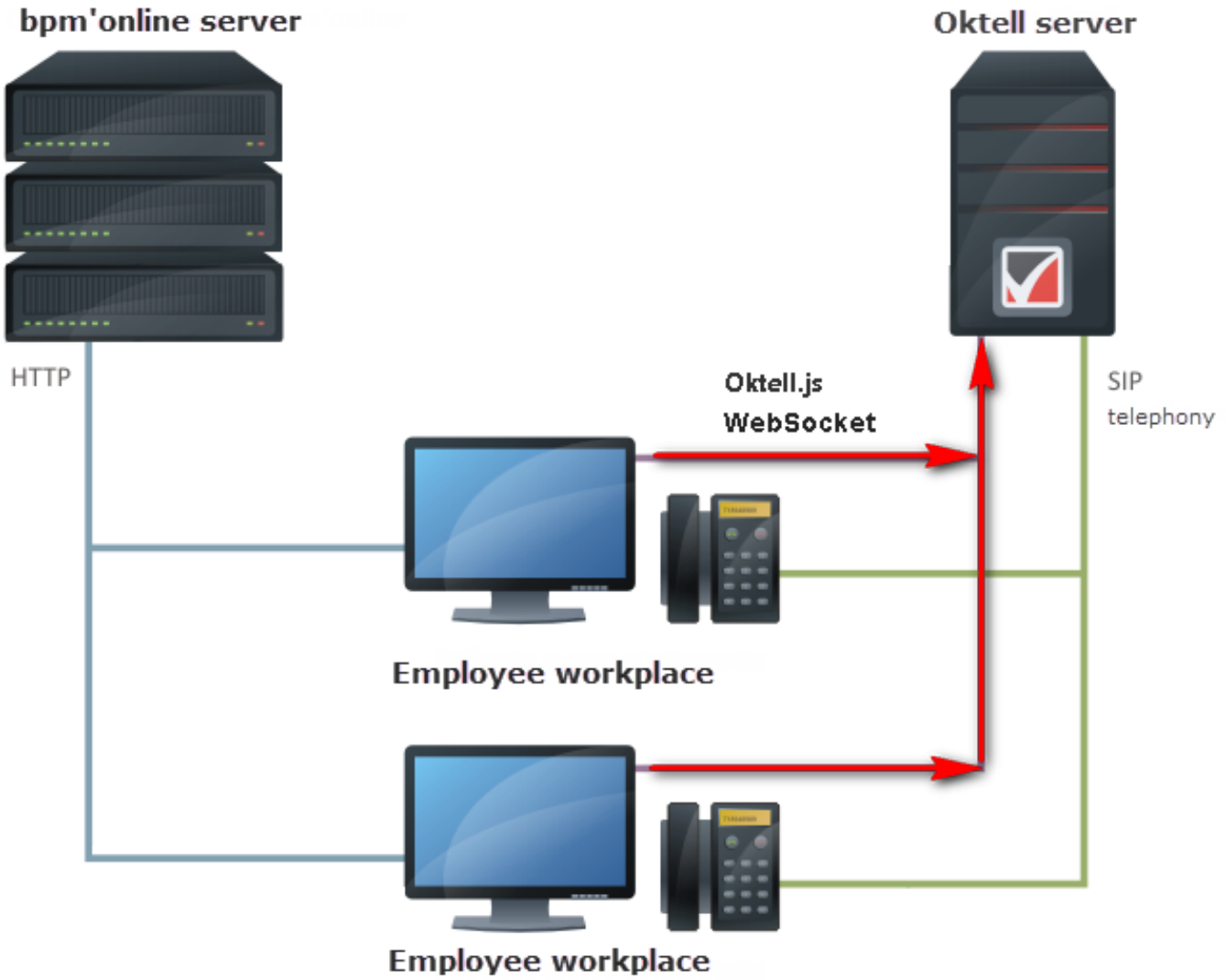


General information

Oktell integration with bpm'online is implemented on the client level using the oktell.js library. The oktell.js source code is located in the *OktellModule* configuration schema of the *CTIBase* package.

The Oktell server communicates with phones and with the end clients (browsers). With this integration method bpm'online does not requires its own WebSocket server. Each client connects via the WebSocket Protocol directly to the Oktell server. The bpm'online application server creates pages and provides data from the application database. There is no direct relationship between bpm'online and Oktell server. Access is not required, so customers process and combine the data of the two systems independently. The Oktell web client and the oktell.js plugin, embedded in other projects, are implemented according to this principle (Fig. 1).

Fig. 1. Oktell integration with bpm'online schema



Oktell.js

Oktell.js is a javascript library for embedding the functionality of the call control in a CRM system. Oktell.js uses the Oktell WebSocket Protocol to connect to the Oktell server. The advantage of this Protocol is the establishing of a permanent asynchronous connection to the server, which enables you to receive events from the server Oktell and execute certain commands. Because the Oktell WebSocket protocol is quite complicated to implement, the Oktell.js wraps the WebSocket Protocol methods inside itself thus providing simple management functionality.

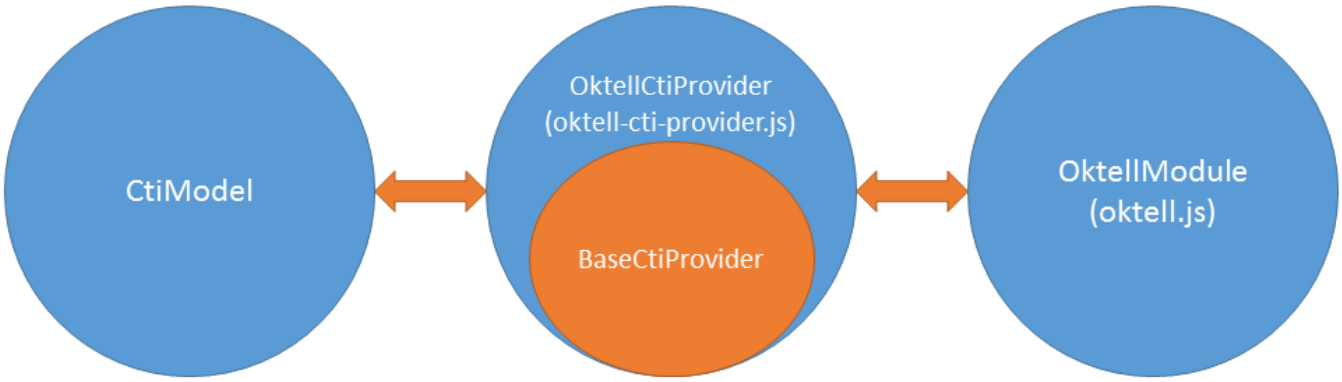
Voice transmission between subscribers

In a conversation between the oktell and bpm'online operators, voice is transmitted via the Session Initiation Protocol (SIP). This requires that either the VoIP phone or the Softphone operator be installed on your computer (Fig. 1).

Interaction of components

The interaction with the oktell.js library is executed via the *OktellCtiProvider* class, which is a link between *CtiModel* and *OktellModule* that contains the oktell.js code. The *OktellCtiProvider* class implements the *BaseCtiProvider* interface class (Fig. 2).

Fig. 2. The components interaction schema in the process of Oktell integration with bpm'online



Examples of interaction between CtiModel, OktellCtiProvider and OktlellModule are displayed on Fig. 3 and Fig. 4.

Fig. 3. Operator outgoing call to a subscriber: putting a call on hold, putting off hold by a subscriber and finishing the call by the operator.

('scr_oktell_events_01.png' in the on-line documentation)

Fig. 4. Incoming call of a subscriber 1 to an operator with a consultation call to subscriber 2 with the subsequent connection of the subscriber 1 and subscriber 2 by the operator.

('scr_oktell_events_02.png' in the on-line documentation)

The list of supported oktell.js class library events is listed in table 1.

Table 1. The list of supported oktell.js class library events

Event	Description
connect	Successful connection to server event
connectError	Connection to server error in the 'connect' method event. Error codes are the same as for the callback function of the 'connect' method
disconnect	Server connection closing event. The object describing the reason of the disconnection is passed to the callback function.
statusChange	Agent status change event. Two string parameters are passed to the callback function — the new and previous state
ringStart	Incoming call start event
ringStop	Incoming call stop event
backRingStart	Returning call start event
backRingStop	Returning call stop event
callStart	Outgoing call start event
callStop	Call UUID change event
talkStart	Conversation start event.
talkStop	Conversation stop event.
holdAbonentLeave	Caller hold leave event The abonent object is passed to the callback function with information on the caller.
holdAbonentEnter	Caller hold enter event The abonent object is passed to the callback function with information on the caller.
holdStateChange	Hold status change event. The information on the hold is passed to the hold function.
stateChange	Line status change event.

abonentsChange	Current abonents list change event
flashstatechanged	Hold status change low-level event
userstatechanged	User status change low-level event

Webitel

Difficulty level

General information

Webitel integration is implemented in the form of separate bpm'online modules. Modules in the integration include:

The WebitelCore package — modules that contain low-level interactions with Webitel using the Verto module and the CTI panel on the bpm'online application page.

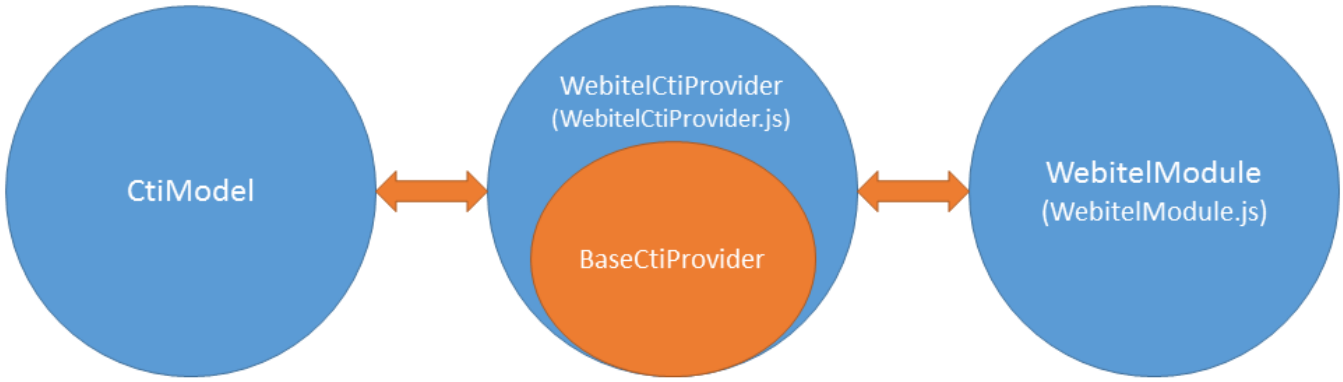
The WebitelCollaborations package implements basic interfaces for working with Webitel in bpm'online. The package contains the *WebitelCtiProvider* module, the *WebitelCtiProvider* class, Webitel connector, the connection parameters settings page, the lookup to edit Webitel users directly in bpm'online.

Detailed information about Webitel architecture can be found in the documentation.

Interaction of components

The *WebitelCtiProvider* class (the heir of the *Terrasoft.BaseCtiProvider* class) implements the required interaction between CtiModel and the Webitel low-level global object (the *WebitelCore.WebitelModule.js* module) (Fig. 1).

Fig. 1. The components interaction schema in the process of Webitel integration with bpm'online



The integration is as follows. If a user has set the system setting of the Webitel integration library, *CtiProviderInitializer* loads the *WebitelCtiProvider* module. Next, it calls the *init* method in *WebitelCtiProvider*, which carries out the user login in the telephony session (the *LogInMsgServer* of the *MsgUtilService.svc* service). If the login was successful, the *connect* method is invoked, which verifies that you don't have an existing connection (the *this.isConnected* property is set to *false* and *this.webitel* — to empty). After that, the *connect* method requests the connection settings to Webitel that are stored in the system settings of the *webitelConnectionString* and *webitelWebrtcConnectionString* (Fig. 2).

Fig. 2. Loading the WebitelCtiProvider provider and connecting the CTI panel

('scr_webitel_events_01.png' in the on-line documentation)

After receiving the system settings, the user settings are received from the [Webitel users] lookup by using the *GetUserConnection* method of the *WUserConnectionService* customer service. After receiving the user settings, the

WebitelModule and *WebitelVerto* are loaded if the [Use web phone] checkbox is selected in the user settings. Next, the *onConnected* method is called that creates the *Webitel* global object, in which properties are populated with the connection settings. The subscription to *Webitel* object events occurs and the *connect* method is invoked, which performs connection via *WebSocket*, authorization of Webitel and other low-level connection operations. When the *onConnect* event occurs, the connection is considered successful and the user can work with calls. During the connector operation, *WebitelCtiProvider* reacts to *Webitel* object events, processes them, and optionally generates connector events described in the *Terrasoft.BaseCtiProvider* class. To manage calls, *WebitelCtiProvider* implements abstract methods of the *Terrasoft.BaseCtiProvider* class by using the *Webitel* object methods.

Examples of CtiPanel, CtiModel and WebitelCtiProvider interaction

Outgoing call to a subscriber: putting a call on hold, taking a call off hold by a subscriber and finishing a call.

Fig. 3. Sequence of events during a call

('scr_webitel_events_02.png' in the on-line documentation)

Webitel list of ports

- 871 — the WebSocket port for the Webitel server and receiving events.
- 5060 and 5080 — signal ports for SIP phones and telephony providers.
- 5066 — the port for the Web phone and WebRTC signal port.
- 4004 — the port for receiving call records.

Webitel events

Table 1. WebitelCtiProvider events

Event	Description
onNewCall	New call start event.
onAcceptCall	Accept call event.
onHoldCall	Call hold event.
onUnholdCall	Call Unhold event.
onDtmfCall	Tone dialing event.
onBridgeCall	Connection to channel event.
onUuidCall	Call UUID change event
onHangupCall	Call stop event.
onNewWebRTCCall	New WebRTC session event.

Asterisk

Difficulty level

General information

Use AMI (Asterisk Manager Interface) to interact with the Asterisk server. The API enables client programs to connect to Asterisk server by using TCP/IP protocol. The Application Programming Interface enables you to process events in the digital multiplex system (DMS), and send commands to control calls.

NOTE

Currently the integration of bpm'online with Asterisk is supported up to version 11.

A client uses a simple text protocol for communication between the Asterisk and the connected Manager API: "parameter: value". The end of a string is determined by the sequence of Carriage Return and Line Feed (CRLF).

NOTE

In the future, for a set of strings like "parameter: value", followed by a blank line containing only a CRLF, for simplicity the term "package" will be used.

How to set up the configuration file of the Messaging Service to integrate Asterisk to bpm'online

For integration to work with bpm'online, you need to install Terrasoft Messaging Service (TMS) on a dedicated computer that will be used as the integration server. You must set the following parameters for Asterisk in the *Terrasoft.Messaging.Service.exe.config* configuration file:

```
<asterisk filePath="" url="Name_or_address_of_Asterisk_server"
port="Asterisk_server_port" userName="Asterisk login" secret="Asterisk password"
originateContext="Originate context" parkingLotContext="Parking lot context"
autoPauseOnCommutationStart="true" queueExtensionFormat="Local/{0}@from-queue/n"
asyncOriginate="true" sendRingStartedOnRingingState="true" traceQueuesState="false"
packetInfoConfig="Additional package parameters to be processed in configuration" />
```

Ports for Asterisk integration with bpm'online

- TMS accepts WebSocket connection to the 2013 port via TCP.
- TMS connects to the Asterisk server by default via the 5038 port.

The Terrasoft Messaging Service for Asterisk integration with bpm'online

The integration part of the Messaging Service is implemented in the main bpm'online solution kernel in the *Terrasoft.Messaging.Asterisk* library.

Library main classes:

- *AsteriskAdapter* — an Asterisk class that transforms events to the top-level call model events used in bpm'online integration.
- *AsteriskManager* — a class that creates and deletes user connections to the Asterisk server.
- *AsteriskConnection* — a class that represents the user connection for integration with Asterisk.
- *AsteriskClient* — a class used to send commands to the Asterisk server.

Example of CtiModel, Terrasoft Messaging Service and Asterisk Manager API interaction

Operator outgoing call to a subscriber: putting a call on hold, putting off hold by a subscriber and finishing the call by the operator.

Fig. 1 shows the occurrence of events for this example While table 1 shows an example of processing of events — how these events are interpreted by the TMS, which values from these events are used in processing an incoming call.

Fig. 3. Sequence of events during a call

('scr_asterisk_events.png' in the on-line documentation)

Table 1. Asterisk events

Asterisk log	TMS	Action	Client
{ Event: newchannel Channel: <channel_name> UniqueID: <unique_id> }	A channel is created and added to a collection *new AsteriskChannel({* *Name: <channel_name>,* *UniqueId: <unique_id>* *});*		
{ Event: Hold UniqueID: <unique_id> Status: "On" }	Search for the channel by <unique_id> and generate an event by using the fireEvent method.	PutHoldAction	Processing the PutHoldAction and displaying the call on hold.
{ Event: Hold UniqueID: <unique_id> Status: "Off" }	Search for the channel by <unique_id> and generate an event by using the fireEvent method.	EndHoldAction	Processing the EndHoldAction and displaying the call on hold.
{ Event: Hangup UniqueID: <unique_id> }	Search for the channel by <unique_id> and generate an event by using the fireEvent method.	RingFinished	Processing event and displaying the call end.
{ Event: Dial SubEvent: Begin UniqueID: <unique_id> }	Search for the channel by <unique_id>, fill in the data and generate an event by using the *fireEvent* method.	RingStarted	Processing the RingStarted event and displaying it on the outgoing call panel.
{ Event: Bridge UniqueID: <unique_id> }	Search for the channel by <unique_id> and generate an event by using the *fireEvent* method.	CommutationStarted	Processing the CommunicationStarted event and displaying the communication.
			Clicking the "Answer" button initiates a new event in Asterisk.

Asterisk events

A detail list of events and information about their parameters is described in the Asterisk documentation.

Email integration

Contents

- **Working with email threads**

Working with email threads

Difficulty level

Introduction

The mechanism for creating email threads is available in bpm’online since version 7.10.0. The main purpose of this mechanism is to improve the email interface. Use this mechanism to easily find email connections, e.g. by copying the connections of the previous email.

The thread construction mechanism is based on the *In-Reply-To* email header data. According to generally accepted agreements, this header should contain the *Message-ID* email identifier. The bpm'online application retrieves these headers from the synchronized Email service (IMAP / Exchange).

The mechanism can be divided into two logical parts – creating threads and populating activity connection fields.

Creating email threads

The *EmailMessageData* detail contains 3 fields:

- *MessageId* – a string of 500 characters in length;
- *InReplyTo* – a string of 500 characters in length;
- *ParentMessageId* – a lookup field that references the *EmailMessageData* detail.

The *MessageId* and *InReplyTo* fields are populated with the corresponding message header values during synchronization.

The *ParentMessageId* field is populated with the following values:

1. The *EmailMessageData* table searches for records where *MessageId* is identical to *InReplyTo*. The first found record is set as the current *ParentMessageId*.
2. The *ParentMessageId* field is populated with the current *Id* value in all *EmailMessageData* records if the *InReplyTo* field of these records is identical to the *MessageId* field.

Email thread connections are updated for every email.

Copying previous email connections in a thread

Spreading activity connections across the thread upon adding an email. The [Case] field is used in this case.

A parent record with the activity that includes the populated [Contact] field is searched for the current *EmailMessageData* record. The [Case] field values from this activity will be spread across the thread. Starting with the found *EmailMessageData* record, all child *EmailMessageData* records are found and their [Case] field value is updated.

A thread with 3 emails:

ActivityId	Title	CaseId	EmailMessageId	EmailMessageParentId
28BD6D59-B9D7-4FF9-89F5-FEE1DD003912	Re: relation	NULL	66812FBF-411B-4FE0-94C9-1E70FBBEB2D3	F05B529D-C98C-4E26-BE00-21F8721AEF58
DC0A40D4-700A-40EB-B394-90E0376C3B5D	Re: relation	1C6E18E3-48B1-495E-8EF9-ACA35DB9DE0B	F05B529D-C98C-4E26-BE00-21F8721AEF58	E1A0120E-B7C0-4261-9DE0-C63341BF1E0B

ActivityId	Title	CaseId	EmailMessageId	EmailMessageParentId
D7A9B82C-ED46-437C-980A-B2650D4FF3DA	relation	906909E8-4D64-47FD-AF92-B65B0826AEC3	E1A0120E-B7C0-4261-9DE0-C63341BF1E0B	NULL

Another email is received in the thread:

ActivityId	Title	CaseId	EmailMessageId	EmailMessageParentId
6623B052-73AD-4AE5-AE61-A6F9BCD930A0	Re: relation	NULL	60C00B01-D0BF-40F6-923E-1830E433AEA1	66812FBF-411B-4FE0-94C9-1E70FBBEB2D3
28BD6D59-B9D7-4FF9-89F5-FEE1DD003912	Re: relation	NULL	66812FBF-411B-4FE0-94C9-1E70FBBEB2D3	F05B529D-C98C-4E26-BE00-21F8721AEF58
DC0A40D4-700A-40EB-B394-90E0376C3B5D	Re: relation	1C6E18E3-48B1-495E-8EF9-ACA35DB9DE0B	F05B529D-C98C-4E26-BE00-21F8721AEF58	E1A0120E-B7C0-4261-9DE0-C63341BF1E0B
D7A9B82C-ED46-437C-980A-B2650D4FF3DA	relation	906909E8-4D64-47FD-AF92-B65B0826AEC3	E1A0120E-B7C0-4261-9DE0-C63341BF1E0B	NULL

Starting with the record in which *EmailMessageId* is "60C00B01-D0BF-40F6-923E-1830E433AEA1", a record with the populated *CaseId* column is searched (does not contain *NULL*). This is a record where *EmailMessageId* = "F05B529D-C98C-4E26-BE00-21F8721AEF58", and *CaseId* = "1C6E18E3-48B1-495E-8EF9-ACA35DB9DE0B".

Now, starting with the record in which *EmailMessageId* is "F05B529D-C98C-4E26-BE00-21F8721AEF58", bpm'online updates the value of the *CaseId* field for the connected records throughout the thread.

ActivityId	Title	CaseId	EmailMessageId	EmailMessageParentId
6623B052-73AD-4AE5-AE61-A6F9BCD930A0	Re: relation	1C6E18E3-48B1-495E-8EF9-ACA35DB9DE0B	60C00B01-D0BF-40F6-923E-1830E433AEA1	66812FBF-411B-4FE0-94C9-1E70FBBEB2D3
28BD6D59-B9D7-4FF9-89F5-FEE1DD003912	Re: relation	1C6E18E3-48B1-495E-8EF9-ACA35DB9DE0B	66812FBF-411B-4FE0-94C9-1E70FBBEB2D3	F05B529D-C98C-4E26-BE00-21F8721AEF58
DC0A40D4-700A-40EB-B394-90E0376C3B5D	Re: relation	1C6E18E3-48B1-495E-8EF9-ACA35DB9DE0B	F05B529D-C98C-4E26-BE00-21F8721AEF58	E1A0120E-B7C0-4261-9DE0-C63341BF1E0B
D7A9B82C-ED46-437C-980A-B2650D4FF3DA	relation	906909E8-4D64-47FD-AF92-B65B0826AEC3	E1A0120E-B7C0-4261-9DE0-C63341BF1E0B	NULL

NOTE

Mail servers sometimes send out letters in the wrong sequence, disregarding the way they were written in the thread

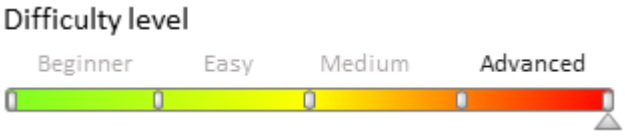
(e.g. during synchronization, emails are received first from the inbox and then from the outbox). This complicates the mechanism. Building a thread for the emails that were not received consistently is impossible during synchronization. In that case, a thread can be built once the synchronization is complete. If certain mailbox folders are not loaded, or if the conversation was interrupted by other participants, the thread search logic may not work. However, the thread can be built when all emails are downloaded from the inbox in most cases.

Recommendations for adding a custom search process for all thread connections after downloading an email

Use the following guidelines to start working on a new email after thread formation:

1. The *Id* field of the synchronization session appears in the *EmailMessageData* table (the values are unique for all synchronization processes). This field is populated only for synchronized emails.
2. A record is added to the new *FinishedSyncSession* table if at least one email was received during synchronization.
3. Certain processes that responded to the signal of saving activities now respond to the (*Inserted*) insertion signal of the *FinishedSyncSession* object. Emails from *EmailMessageData* are selected based on the synchronization session *Id*. The *MailboxSyncSettings* field of the *EmailMessageData* object can be used to select Email messages from the service box.

Self-service Portal



Introduction

The Self-service Portal (SSP) is an integral part of the bpm’online service enterprise and bpm’online customer center products, as well as all bundles that these products are part of.

The SSP is a workplace where portal users are automatically redirected after login.

Portal users have access to the [Cases] and [Knowledge base] sections and to the [Self-service portal main page], which contains general summary information and works as a single workplace for a portal user.

The [Cases] section is used for registration of cases by the customers, viewing the status of their cases, entering additional case information and for obtaining information about case resolution process. By default, a portal user has access to those cases where this user is specified as a contact. The user can enter additional information, publish messages and interact with the service staff on the case page. The case history is displayed on the case page at the [Processing] tab.

The portal’s [Knowledge base] section is used for searching for reference information or a solution. This section can be filled only by the helpdesk staff from the main interface of the system.

Portal interface

From the development point of view, the portal is a preconfigured separate workplace. By default, this workplace is not available for the ordinary portal users. A system user with the “portal user” type automatically enters this workplace (the portal main page) after authorization.

Portal sections and pages are the same as **sections** and **system edit pages** from the main system interface and they work with the same *Entities*. The case edit page on the portal is simpler compared to the regular case page and does not contain the majority of fields. These edit pages are different objects in configuration (*CasePage* and *PortalCasePage*).

The system of portal access permissions slightly differs. To grant access to the specific entities (EntitySchema) for the portal users, you need to specify these entities in the [List of objects available for portal users] lookup. Self-service portal licenses limit the number of records that can be added to this lookup. By default, the number is limited to 70 records.

Working with the page wizard on the portal

The portal user cannot access the functions of the page-, detail- and section wizards These functions can be accessed from the main system interface with administrator permissions in the following way:

1. Enter the [Workplace setup] section in the system designer.
2. Select the [Portal] workplace and click the [Open] button.
3. Select the required section and click the [Section wizard] button.

The standard section wizard will open.

Configuring the portal and portal users

To start using the portal:

1. Ensure that the */configuration/terrasoft/auth* option in the *web.config* file of the application loader (the “external” *web.config*) has the following in it:

```
<terrasoft>
   <auth providerNames="InternalUserPassword,SSPUserPassword" …>
...
</terrasoft>
```

This setting is responsible for the authorization in the portal users in the system.

2. Create a contact for the user.

3. Create a user with the “Portal user” type. Fill out the required fields.

4. Provide all necessary licenses for the user.

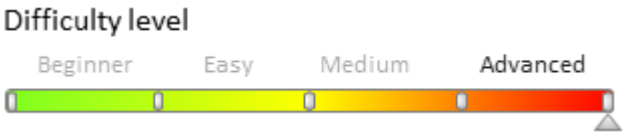
A portal user is required to have a valid time zone specified in the profile. The time zone is not specified for new users by default. Portal users must edit their profiles and select the proper time zone The system will display all dates and times in the portal user’s local time.

ClientMessageBridge

Contents

- **ClientMessageBridge. Message history save mechanism**
- **ClientMessageBridge. API description**
- **ClientMessageBridge. The client-side WebSocket message handler**

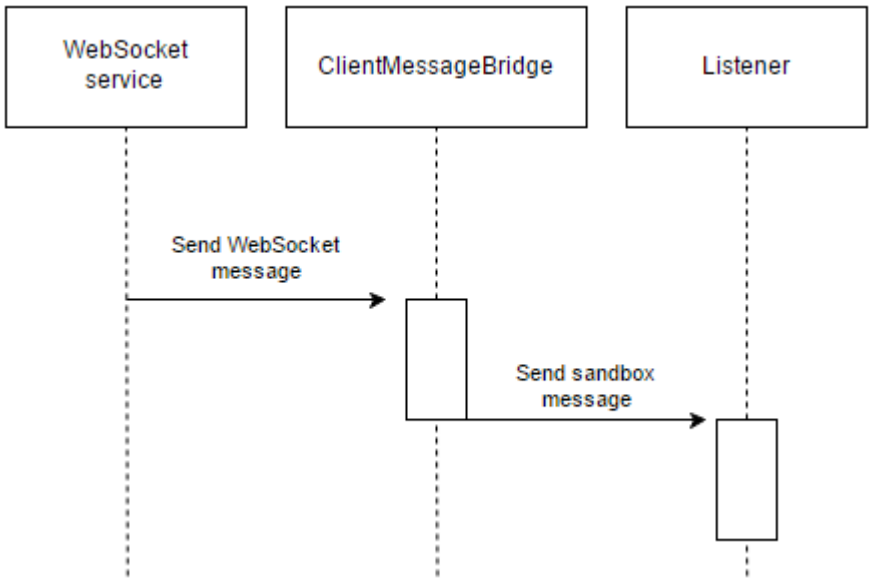
ClientMessageBridge. Message history save mechanism



General information

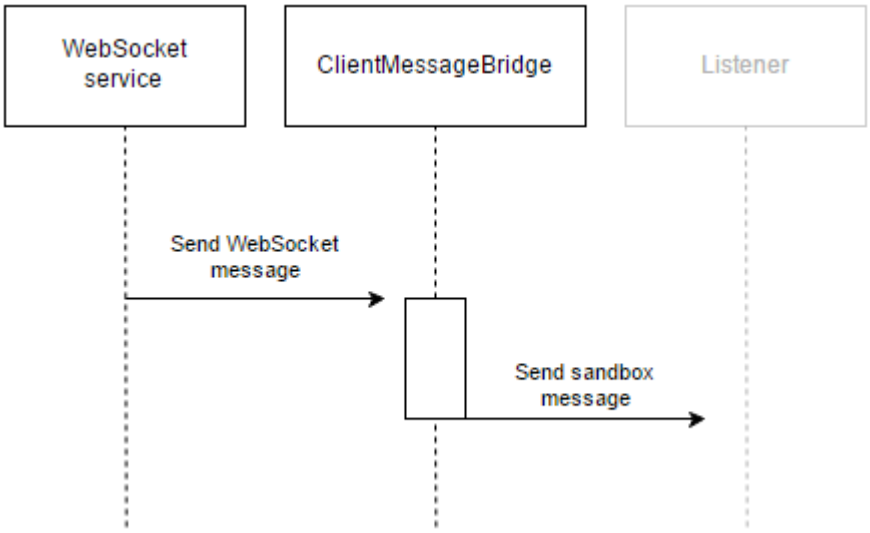
In a best-case scenario a handler is located within the system at the time of publication of the message (Fig. 1).

Fig. 1. Perfect interaction mechanism



However, there may be situations when a handler is not yet loaded (Fig. 2).

Fig. 2. The process of interaction with an absent handler



In order to not lose unprocessed messages, you can wait for a listener before publication. If a listener is absent, the messages are saved in history. Before publication, each message is checked for a listener. When the listener is loaded, all stored messages are published in the order they were received. After the messages have been published, the history is cleared.

Configuring the processing of messages stored in the history example

To implement the described feature, you have to set the *isSaveHistory* checkbox to *true*.

```
init: function() {
    // Calling the init() parent method.
    this.callParent(arguments);
    // Adding a new configuration object to the configuration object collection.
    this.addMessageConfig({
        // sender — name of the message received via WebSocket.
        sender: "OrderStepCalculated",
        // Name of the message sent within the system.
```

```
        messageName: "OrderStepCalculated",
        // isSaveHistory – checkbox indicating that the history must be saved.
        isSaveHistory: true
    });
},
```

History saving mechanism

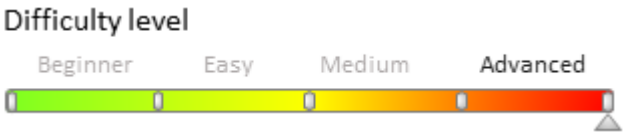
The *ClientMessageBridge* class is the heir of the *BaseMessageBridge* abstract class, which contains abstract methods (*saveMessageToHistory, getMessagesFromHistory, deleteSavedMessages*). In *ClientMessageBridge* message history saving is implemented with the use of the *localStorage* of the browser, and the implementation of abstract methods enables you to manipulate data in storage. To work with the *localStorage* class, use the *Terrasoft.LocalStore* class.

Methods:

- *saveMessageToHistory* — ensures that new messages are saved in the message collection
- *getMessagesFromHistory* — provides an array of messages based on the transferred name
- *deleteSavedMessages* — deletes all messages based on the transferred name

If there is a need to implement another type of storage, you must create an heir class of the *BaseMessageBridge* class and implement all abstract methods (*saveMessageToHistory, getMessagesFromHistory, deleteSavedMessages*).

ClientMessageBridge. API description



Properties

WebSocketMessageConfigs: *Array*

Collection of configuration objects.

LocalStoreName: *String*

Name of the repository, where the message history is stored.

LocalStore: *Terrasoft.LocalStore*

An instance of class that implements access to local repository.

Methods

init()

Initializes default values.

getSandboxMessageListenerExists*(sandboxMessageName)*

Checks for available listeners of message with passed name.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

Returned value:

Boolean – the result of checking for available message listeners

publishMessageResult*(sandboxMessageName, webSocketMessage)*

Publishes the message within the system.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

webSocketMessage: Object – message received by WebSocket.

Returned value:

* – result obtained from the message handler.

beforePublishMessage*(sandboxMessageName, webSocketBody, publishConfig)*

Handler that is called before publishing the message within the system.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

webSocketBody: Object – message received by WebSocket.

publishConfig: Object – configuration object of message broadcast.

afterPublishMessage*(sandboxMessageName, webSocketBody, result, publishConfig)*

Handler that is called after publishing the message within the system.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

webSocketBody: Object – message received by WebSocket. result: * – result of publishing the message within the system.

publishConfig: Object – configuration object of message broadcast.

addMessageConfig*(config)*

Adds a new configuration object to a collection of configuration objects.

Parameters:

config: Object – configuration object.

Configuration object parameter:

```
{
    "sender": "webSocket sender key 1",
    "messageName": "sandbox message name 1",
    "isSaveHistory": true
}
```

Where:

- *sender: String* – name of the message that is expected from WebSocket.
- *messageName: String* – message name that will be used when sending the message within the system.
- *isSaveHistory: Boolean* — determines whether message history must be saved.

saveMessageToHistory*(sandboxMessageName, webSocketBody)*

Saves message in the repository if there is no subscriber and the save checkbox is selected in the configuration object.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

webSocketBody: Object – message received by WebSocket.

getMessagesFromHistory*(sandboxMessageName)*

Gets an array of saved messages from the repository.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

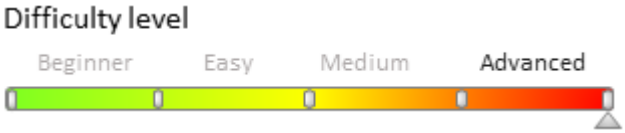
deleteSavedMessages*(sandboxMessageName)*

Deletes saved messages from the repository.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

ClientMessageBridge. The client-side WebSocket message handler



General information

The ClientMessageBridge schema is used to broadcast messages received via WebSocket. If additional logic in the extending ClientMessageBridge schemas wasn't specified for each message received via WebSocket, a broadcast is used to send messages within the system through the SocketMessageReceived sandbox. After subscribing to a message, you can easily process the data received through WebSocket.

For additional message handling, it is necessary to implement an extending schema before publishing and changing the message name. Using the available API, you can configure specific message types in the extending schema.

Setting up a new subscriber example

Case description

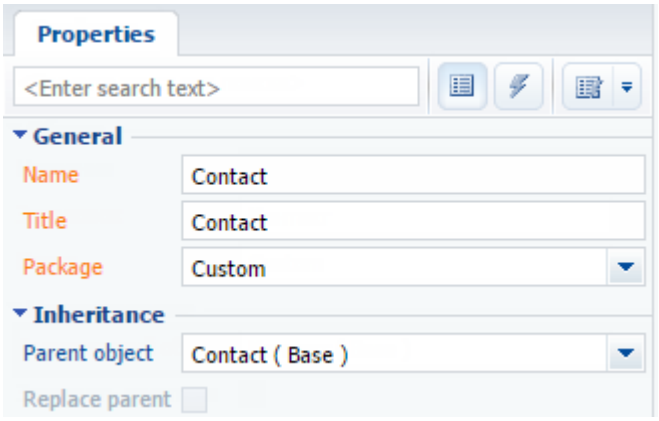
When a contact is saved, you need to publish a message with the *NewUserSet* name that contains information about the contact's birth date and name on the server side. On the client side, you must implement the NewUserSet messaging within the system. Additionally, before messaging, you must process the *birthday* message property received through WebSocket, and you must invoke the afterPublishUserBirthday utility class method after messaging. Finally, you need to implement the subscription to a message sent on the client side, for example, in the schema of the contact edit page.

Case implementation algorithm

1. Create the replacing [Contact] object

Before adding message publishing via WebSocket, you have to create a replacing object and set the [Contact] as a parent (Fig. 1).

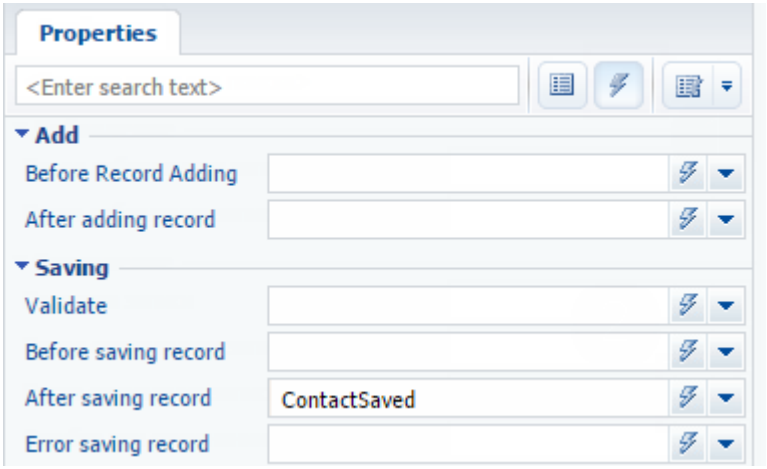
Fig. 1. Creating the replacing [Contact] object



2. Create the "Record saved" event

Next, you need to add message publishing via WebSocket after the contact record has been saved. To do this, go to the tab with the object events (Fig. 2, 1) and click the "Record saved" button (Fig. 2, 2).

Fig. 2. Creating the "Record saved" event



3. Implement event subprocess in the "Record saved" event

In the Record Saved event handler, implement the event subprocess which is run by the ContactSaved message. To do this:

- Add an event subprocess element (Fig. 3, 1);
- Add a message element (Fig. 3, 2), setting ContactSaved as the message name (Fig. 4);
- Add a script element (Fig. 3, 3);
- Connect the message object and script (Fig. 3, 4).

Fig. 3. Creating message handler subprocess

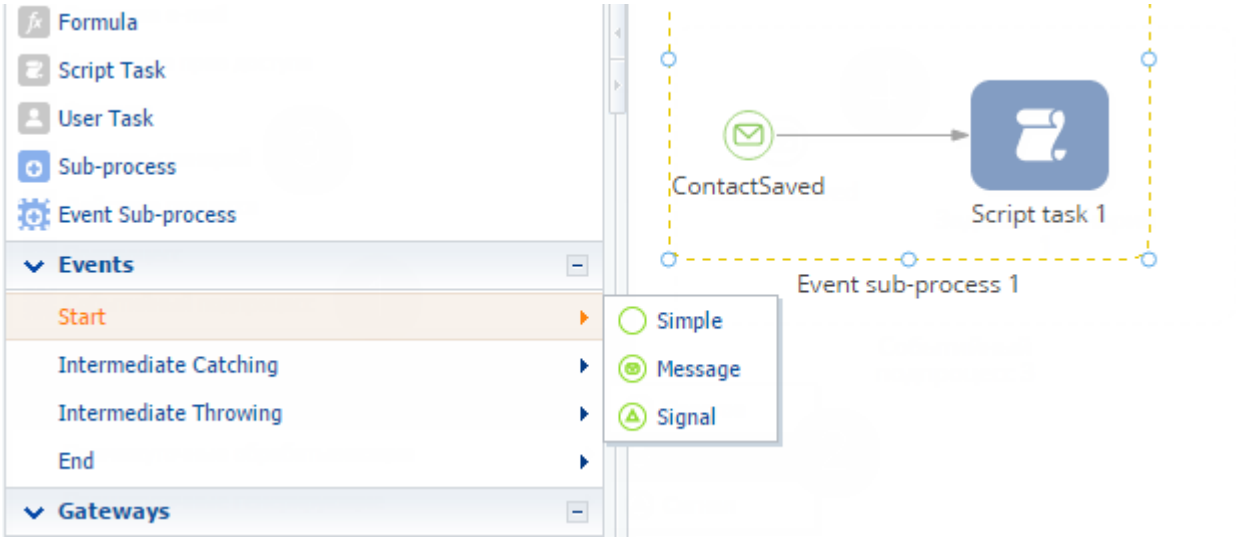
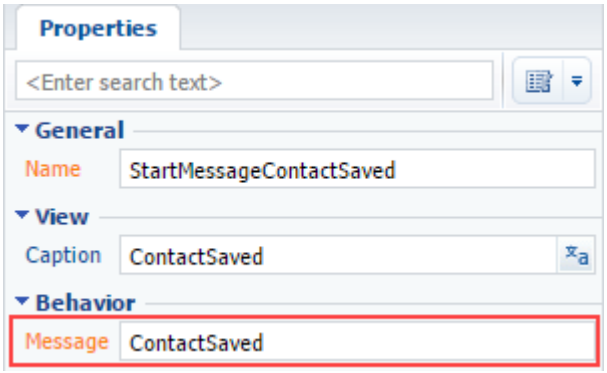


Fig. 4. Initial message properties



4. Add message publication logic through WebSocket

To do this, double-click to open the [Script-task] event subprocess and add the following source code:

```
// Receiving contact name
string userName = Entity.GetTypedColumnValue<string>("Name");
// Receiving contact birth date.
DateTime birthDate = Entity.GetTypedColumnValue<DateTime>("BirthDate");
// Forming message text.
string messageText = "{\"birthday\": \"" + birthDate.ToString("s") + "\", \"name\":
\"" + userName + "\"}";
// Setting message name.
string sender = "NewUserSet";
// Publishing message through WebSocket.
MsgChannelUtilities.PostMessageToAll(sender, messageText);
return true;
```

After that, save and close the tab containing the [Script-task] element source code, and then save and publish the whole event subprocess.

5. Implement message sending inside the application

To do this, create a replacing client module in a custom package(Fig. 5) and set the ClientMessageBridge of the NUI package as a parent object (Fig. 6).

Fig. 5. Creating a replacing client module

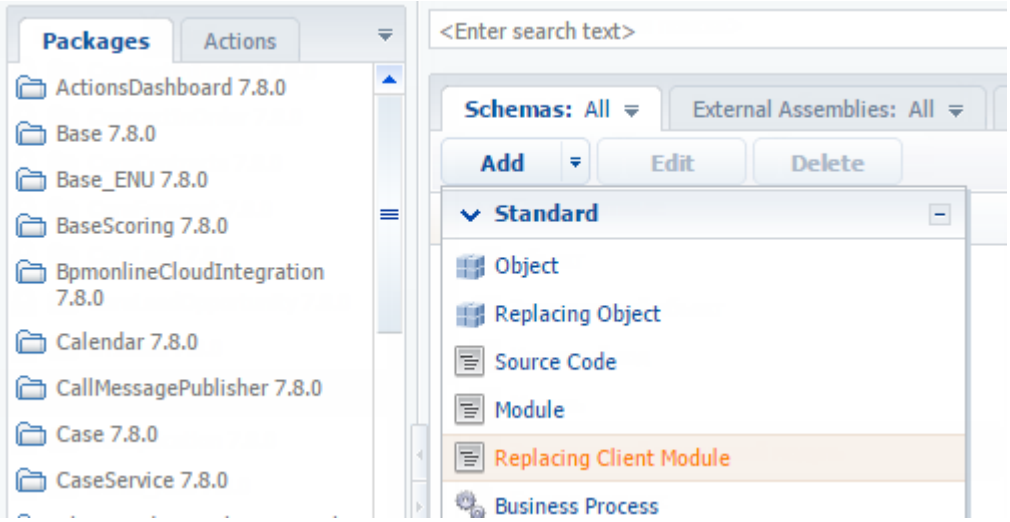
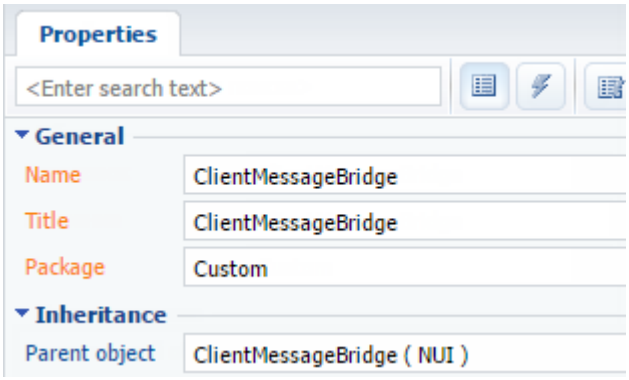


Fig. 6. Client module properties.



Read more about replacing client modules in the "**Creating a custom client module schema**" article.

To implement the distribution of messages NewUserSet within the system, it is necessary to add the following source code in the schema:

```
define("ClientMessageBridge", ["ConfigurationConstants"],
    function(ConfigurationConstants) {
        return {
            // Messages.
            messages: {
                // Message name.
                "NewUserSet": {
                    // Message type – broadcasting, without a specific subscriber.
                    "mode": Terrasoft.MessageMode.BROADCAST,
                    // Message direction – publication.
                    "direction": Terrasoft.MessageDirectionType.PUBLISH
                }
            },
            methods: {
                // Schema initialization.
                init: function() {
                    // Parent method calling
                    this.callParent(arguments);
                    // Adding new configuration object to the configuration object
collection.
                    this.addMessageConfig({
                        // Name of the message received via WebSocket.
                        sender: "NewUserSet",
                        // Name of the message sent within the system.
                        messageName: "NewUserSet"
```

```
                    });
                },
                // Method executes after the message publication.
                afterPublishMessage: function(
                    // Name of the message sent within the system.
                    sandboxMessageName,
                    // Message contents.
                    webSocketBody,
                    // Message result
                    result,
                    // Message configuration object.
                    publishConfig) {
                    // Check whther the message matches the one added to the
configuration object.
                    if (sandboxMessageName === "NewUserSet") {
                        // Saving the content to local variables.
                        var birthday = webSocketBody.birthday;
                        var name = webSocketBody.name;
                        // Displaying content in the console.
                        window.console.info("Published message: " +
sandboxMessageName +
                            ". Data: name: " + name +
                            "; birthday: " + birthday);
                    }
                }
            }
        };
    });
```

Go the *messages* section and bind the *NewUserSet* broadcast message, which can only be published within the system. Go to the the *methods* section and restart the *Init* parent method to add the messages received via WebSocket in the configurational schema message object. To track messaging launch time, reload the *afterPublishMessage* parent method.

After the schema has been saved and the application page has been refreshed, the NewUserSet messages received via WebSocket will be sent within the system. Read more about debugging in the browser in the "**Client code debugging**" article.

6. Implement message subscription

To obtain an object transmitted via WebSocket, you must subscribe to NewUserSet messages in any scheme, for example, "Page contact V2". To do this, you need to create a replacing client module (see section 5), specifying the "Contact page display schema" as a parent object. To do this, add the following source code:

```
define("ContactPageV2", [],
    function(BusinessRuleModule, ConfigurationConstants) {
        return {
            //entitySchemaName: "Contact",
            messages: {
                // Message name.
                "NewUserSet": {
                    // Message type – broadcasting, without a specific subscriber.
                    "mode": Terrasoft.MessageMode.BROADCAST,
                    // Message direction – subscription.
                    "direction": Terrasoft.MessageDirectionType.SUBSCRIBE
                }
            },
            methods: {
                // Schema initialization.
                init: function() {
                    // Init() parent method calling.
                    this.callParent(arguments);
```

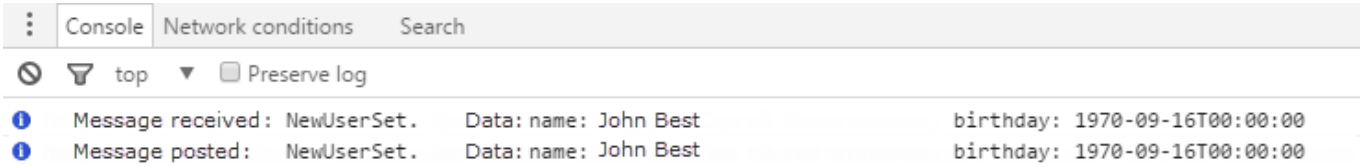
```
                    // Subscription to receiving the NewUserSet message.
                    this.sandbox.subscribe("NewUserSet", this.onNewUserSet, this);
                },
                // Receiving NewUserSet message event handler.
                onNewUserSet: function(args) {
                    // Saving the message content to local variables.
                    var birthday = args.birthday;
                    var name = args.name;
                    // Displaying content in the console.
                    window.console.info("Message received: NewUserSet. Data: name: "
+
                        name + "; birthday: " + birthday);
                }

            }
        };
    });
```

Go the messages section and bind the NewUserSet broadcast message, which can only be published within the system. Go to the *methods* section and restart the *Init* parent method to subscribe to the NewUserSet message and indicate the onNewUserSet, method-handler that processes in the message and displays the result in the browser console.

After you have added the source code, you must save the schema and refresh the application page in the browser.

The result of the case is two informational messages in the browser console after you save the contact (Fig. 7).

Fig. 7. Browser console

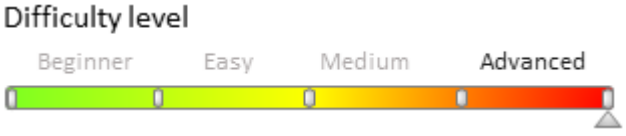


Sync Engine synchronization mechanism

Contents

- **Bpm'online synchronization with external storages**
- **Synchronizing metadata in bpm'online**
- **Synchronizing tasks with MS Exchange**
- **Synchronizing email with MS Exchange**
- **Synchronizing contacts with MS Exchange**
- **Synchronizing appointments with MS Exchange**

Bpm'online synchronization with external storages



General information

The mechanism in bpm'online for synchronization with external data storages is the Sync Engine. This mechanism enables you to create, modify, and delete *Entity* in the system based on synchronization with external systems and

export data to external systems.

Synchronization is performed by using the *SyncAgent* class implemented in the *Terrasoft.Sync* namespace of the application kernel.

Classes used in the synchronization mechanism

- Synchronization agent (*SyncAgent*) — a class with one public *Synchronize* method that triggers synchronization between storages.
- Synchronization context (*SyncContext*) — a class representing the aggregation of providers and metadata for *SyncAgent*.
- Storage — storage of synchronized data.
    - Local storage (*LocalProvider*) — enables you to work with *LocalItem* in bpm'online.
    - External storage (*RemoteProvider*) — an external service or application from which data is synchronized with bpm'online.
- Synchronization item (*SyncItem*) — a set of objects from external and local storage which are synchronized.
    - External storage synchronization item (*RemoteItem*) — represents a set of data from external storage that syncs automatically. It can consist of one or more entities (records) from the external storage.
    - Synchronization item (*SyncEntity*) — a wrapper of a specific *Entity*. *SyncEntity* is required to work with *Entity* as the synchronizing object (add, delete, change).
    - Synchronization item (*LocalItem*) — one or more objects from bpm'online that are synchronized with the external storage as a unit. The synchronization item from the external storage, converted in the *LocalItem* entity contains a set of instances of the *SyncEntity* class.
- *SysSyncMetadata* metadata table — contains service information of the synchronized elements and is essentially *RemoteItem-LocalItem* interchanges table. Metadata sync description can be found in the "**Synchronizing metadata in bpm'online**" article.

General synchronization algorythm

Before starting synchronization, you must create an instance of *SyncAgent* and the *SyncContext* sync context, then update records in the metadata table with data from bpm'online. For this, you need to call the *CollectChangesInSyncedEntities* class method that implements the *IReplicaMetadata* interface.

The algorithm for updating metadata records is the following:

1. If any previously synchronized entity in bpm'online has been modified since the last synchronization, then the corresponding record in the metadata changes its modification date, the *LocalState* property is set to “Modified”, and the source of the modification is set to the *LocalStore* ID.
2. If a synchronized entity in bpm'online has been deleted since the last synchronization — the corresponding record in the metadata LocalState is set to “Deleted".
3. If there is no corresponding record in the metadata for the entity in bpm'online — it is ignored.

The process of synchronizing storages then starts the following:

1. All changes from the external storage are requested alternately.
2. You need to obtain the metadata for each item in the external storage .
    a. If the metadata can not be obtained — this is a new item which should be converted to a bpm'online element. To fill in a synchronization object, a *FillLocalItem* method is called from the specific *RemoteItem* instance. It is also recorded in the metadata (ID of the external storage, the element ID in the external storage, date of creation and modification is set as current, the source of creation and modification — external storage).
    b. If the metadata is received, so this item has already been synchronized with bpm'online. You need to go to the version conflict resolution. By default, the last change in the application or external storage (*RemoteProvider*) has the priority.
    c. The metadata for the current pair of synchronization items is actualized.

After looking through all the modified items from the external storage, the elements that were changed in bpm'online, but was not changed in the external storage remain in the metadata (in the interval between the last and

the current synchronization).

1. You need to get elements changed in bpm'online in the interval between the last synchronization and the current synchronization.
2. Save the changes in the external storage.
3. You must update the modification date of the items in the metadata (bpm'online is the change source).

After that, you need to add new, not synchronized records to the external storage, and add metadata for new items.

The synchronization context

SyncContext class

A class representing the aggregation of providers and metadata for *SyncAgent*. The properties of the *SyncContext* class are presented in Table 1.

Table 1. *SyncContext class properties*

Field	Type	Purpose
Logger	*ISyncLogger*	The object that enables messages to be saved into the integration log.
LocalProvider	*LocalProvider*	Enables you to work with *LocalItem*.
RemoteProvider	*RemoteProvider*	External service or application, data from which is synchronized with bpm'online.
ReplicaMetadata	*IReplicaMetadata*	Works with metadata.
LastSyncVersion	*DateTime*	The date and time of the last synchronization in UTC.
CurrentSyncStartVersion	*DateTime*	The current date and time synchronization in UTC. Set after the metadata update.

Requirements for synchronization with external storage

External storage

External storage (*RemoteProvider*) — encapsulates data from the external storage.

RemoteProvider — a basic class that enables you to work with an external storage. In fact, it is the only way to work with external systems. Properties of the *RemoteProvider* class are presented in table 2 and the methods — in table 3.

Table 2. *RemoteProvider class properties*

Field	Type	Description
StoreId	*Guid*	ID of external storage that will be synchronized.
Version	*DateTime*	Date and time of the last synchronization in UTC.
SyncItemSchemaCollection	*List*	External storage element schema
RemoteChangesCount	*Int*	Number of items processed from external storage.
LocalChangesCount	*Int*	Number of items processed from local storage.

Table 3. *RemoteProvider class methods*

Method	Returning value type	Description

KnownTypes()	*IEnumerable*	Returns a collection of all types that implement the *IRemoteItem* interface. *SyncAgent* builds the *SyncItemSchema* instances that describe the entities that participate in synchronization.
ApplyChanges(SyncContext context, IRemoteItem synItem)	*Void*	Applies changes to external storage element.
CommitChanges(SyncContext context)	*Void*	Called after processing changes in external and local storage. Intended for the implementation of necessary additional steps for the specific integration implementation.
EnumerateChanges(SyncContext context)	*IEnumerable*	Returns an enumeration of new and modified elements of external storage.
LoadSyncItem(SyncItemSchema schema, string id)	*IRemoteItem*	Fills in the *IRemoteItem* instance with data from external storage.
CreateNewSyncItem(SyncItemSchema schema)	*IRemoteItem*	Creates a new instance of *IRemoteItem*.
CollectNewItems(SyncContext context)	*IEnumerable*	Returns an enumeration of new bpm'online entities that will be synchronized with external storage.
ResolveConflict(IRemoveItem syncItem, ItemMetadata itemMetaData, Guid localStoreId)	*SyncConflictResolution*	Resolves conflicts between changed elements of local and external storages. By default, (*RemoteProvider*) priority is given to changes in bpm'online.
NeedMetaDataActualization()	*Boolean*	Returns the sign that checks whether there is a need to update the metadata before starting synchronization.
GetLocallyModifiedItemsMetadata(SyncContext context, EntitySchemaQuery modifiedItemsEsq)	*IEnumerable*	Returns synchronization elements changed in the local store since the last synchronization.

IRemoteItem interface

The class that implements the *IRemoteItem* interface is an indivisible unit of synchronization and represents one element of the synchronization of the external data storage. This class is a container for data coming from an external system, and it knows how to convert the data to the *Entity* entity, and vice versa. The interface contains two methods: *FillLocalItem* and *FillRemoteItem* for converting external synchronization elements (*RemoteItem*) to *LocalItem*, and vice versa. Interface methods are presented in Table 4.

Table 4. IRemoteItem interface methods

Method	Returning value type	Description
FillLocalItem(SyncContext context, ref LocalItem localItem)	*Void*	Fills in properties of an element of the *LocalItem* local storage with values of external storage element. Used to apply changes in local storage.
FillRemoteItem(SyncContext context,	*Void*	Fills in properties of an element of external

ref LocalItem localItem)	storage with element values from the *LocalItem* local storage. Used to apply changes in external storage.

Map attribute

The *Map* attribute decorates the *iRemoteItem* interface implementations. SchemaName is the main parameter. This is the name of the *EntitySchema* that is included in the current synchronization element.

```
[Map("Activity", 0)]
[Map("ActivityParticipant", 1)]
public class GoogleTask: IRemoteItem {
. . .
```

This class declaration task from Google Calendar will sync with the activity and its participants from bpm'online.

The second parameter, *Order*, specifies in which order *Entity* will be stored in the local storage. *Activity* is indicated first, because *ActivityParticipant* stores a link to the created activity.

In most cases, *SyncAgent* can automatically generate a request for a sample of the new elements of synchronization with bpm'online. To do this, you must specify the basic entity and method of communication with the details:

```
[Map("Activity", 0, IsPrimarySchema = true)]
[Map("ActivityParticipant", 1, PrimarySchemaName = "Activity", ForeingKeyColumnName =
"Activity")]
public class GoogleTask: IRemoteItem {
. . .
```

In this case, a request for new activities will be sent to the database along with a request for each selected activity to receive their participants. The attribute properties list is presented in Table 5.

Table 5. Map attribute properties

Parameter	Type	Description
SchemaName	*String*	Object schema name.
Order	*Int*	The entity processing order for the synchronization element.
IsPrimarySchema	*Boolean*	A flag that indicates that this schema is a key element of this synchronization element. It can be installed in one schema only.
PrimarySchemaName	*String*	The schema name of the main object. It can not be set in tandem with *IsPrimarySchema*.
ForeignKeyColumnName	*String*	The column name for the connection with the details of the main object It can not be set in tandem with *IsPrimarySchema*.
Direction	*SyncDirection*	It specifies the synchronization direction for the objects of this type. By default - *DownloadAndUpload*. If it contains the *Download* flag - the changes will not apply to bpm'online. If it does not contain the *Upload* flag - the new entities will not be selected from bpm'online.
FetchColumnNames	*String[]*	The names of the columns that will be loaded from the local storage.

Local storage

Local Storage (*LocalProvider*) - encapsulates the work with data in internal storage (bpm'online).

LocalProvider - basic class that implements work with the local storage. Enables you to work with *LocalItem*. Methods of this class are immutable and are listed in Table 6.

Table 6. *LocalProvider class methods*

Method	Returning value type	Description
AddItemSchemaColumns(EntitySchemaQuery esqForFetching, EntityConfig entityConfig)	*Void*	Adds *EntitySchemaQuery* column specified in *EntityConfig*.
ApplyChanges(SyncContext context, LocalItem entities)	*Void*	Applies changes to each *SyncEntity* in *LocalItem*.
FetchItem(ItemMetadata itemMetaData, SyncItemSchema itemSchema, bool loadAllColumns = false)	*LocalItem*	Loads a collection of entities associated with a particular synchronization element.

SyncEntity class

The class encapsulates SyncEntity Entity instance and all the necessary actions to perform the synchronization of the instance properties. Class Properties are summarized in Table 7.

Table 7. *SyncEntity class properties*

Parameter	Type	Description
EntitySchemaName	*String*	The name of the schema for which the wrapper is created.
Entity	*Entity*	*Entity* for which the wrapper is created.
State	*SyncState*	The last action performed on the entity (0 - not changed, 1 - new, 2 - changed 3 - deleted).

SystemSchema class

Synchronization element entity settings schema. Class properties are shown in Table 8 and methods in Table 9.

Table 8. *SyncItemSchema class properties*

Parameter	Type	Description
SyncValueName	*String*	Entity type name
SyncValueType	*Type*	Entity type
PrimaryEntityConfig	*EntityConfig*	Synchronization element entity settings.
Configs	*List*	Synchronization element entity settings list.
DetailConfigs	*List*	Synchronization element detail entity settings list.
Direction	*SyncDirection*	It specifies the synchronization direction for the objects of this type. By default - *DownloadAndUpload*. If it does not contain the *Download* flag, changes will not be applied in bpm'online. If it does not contain the *Upload* flag, new entities will not be selected from bpm'online.

Table 9. *SyncItemSchema class methods*

Method	Returning value type	Description
CreateSyncItemSchema(Type syncValueType)	*SyncItemSchema*	It creates a configuration element synchronization entity with all the settings of the related entities.
Validate(UserConnection userConnection)	*Void*	The method checks that *EntityConfig is well-formed.* If authentication fails, an exception is applied. If the *EntityConfig* schema name is specified twice, *DublicateDataException* is generated. If the name of the defunct schema is specified, *InvalidSyncItemSchemaException* is generated).
FetchItem(ItemMetadata itemMetaData, SyncItemSchema itemSchema, bool loadAllColumns = false)	*LocalItem*	Loads a collection of entities associated with a particular synchronization element.

EntityConfig Class

Synchronization element entity settings. Class Properties are summarized in Table 10.

Table 10. *EntityConfig class properties*

Parameter	Type	Description
SchemaName	*String*	Object schema name.
Order	*Int*	The order of processing entities for a synchronization element. The lower the value, the sooner the entity will be processed in the processing of the synchronization element.
Direction	*SyncDirection*	It specifies the synchronization direction for the objects of this type. By default - *DownloadAndUpload*. If it does not contain the *Download* flag, changes will not be applied in bpm'online. If it does not contain the *Upload* flag, new entities will not be selected from bpm'online.
FetchColumnNames	*String[]*	The names of the columns that will be loaded from the local storage. If the value is not specified, it will load all object columns

DetailEntityConfig class

Synchronization element detail entities settings. Class properties are displayed on Table 11.

Table 11. *DetailEntityConfig class properties*

Parameter	Type	Description
PrimarySchemaName	*String*	Bpm'online main synchronized entity schema name.
ForeignKeyColumnName	*String*	The column name for the connection with the details of the main object

LocalItem class

One or more objects from bpm'online that are synchronized with external storage as a unit. It contains a set of

SyncEntity class instances. Class properties are shown in Table 12 and methods in Table 13.

Table 12. *LocalItem class properties*

Parameter	Type	Description
Entities	*Dictionary>*	The *SyncEntity* collection that is set in accordance with a *SyncItem*. It contains a collection of "key-value" pairs, where the key is the schema name, and the value is the SyncEntity collection of the scheme.
Version	*DateTime*	The highest value of the date and time of the modification of all *Entities* in LocalItem.
Schema	*SyncItemSchema*	Synchronization element entity settings schema.

Table 13. *LocalItem class methods*

Method	Returning value type	Description
AddOrReplace(string schemaName, SyncEntity syncEntity)	*Void*	Adds new *SyncEntity* to the collection. If SyncEntity with EntityId already exists, it replaces it.
Add(UserConnection userConnection, string schemaName)	*SyncEntity*	Creates and adds a new SyncEntity collection.

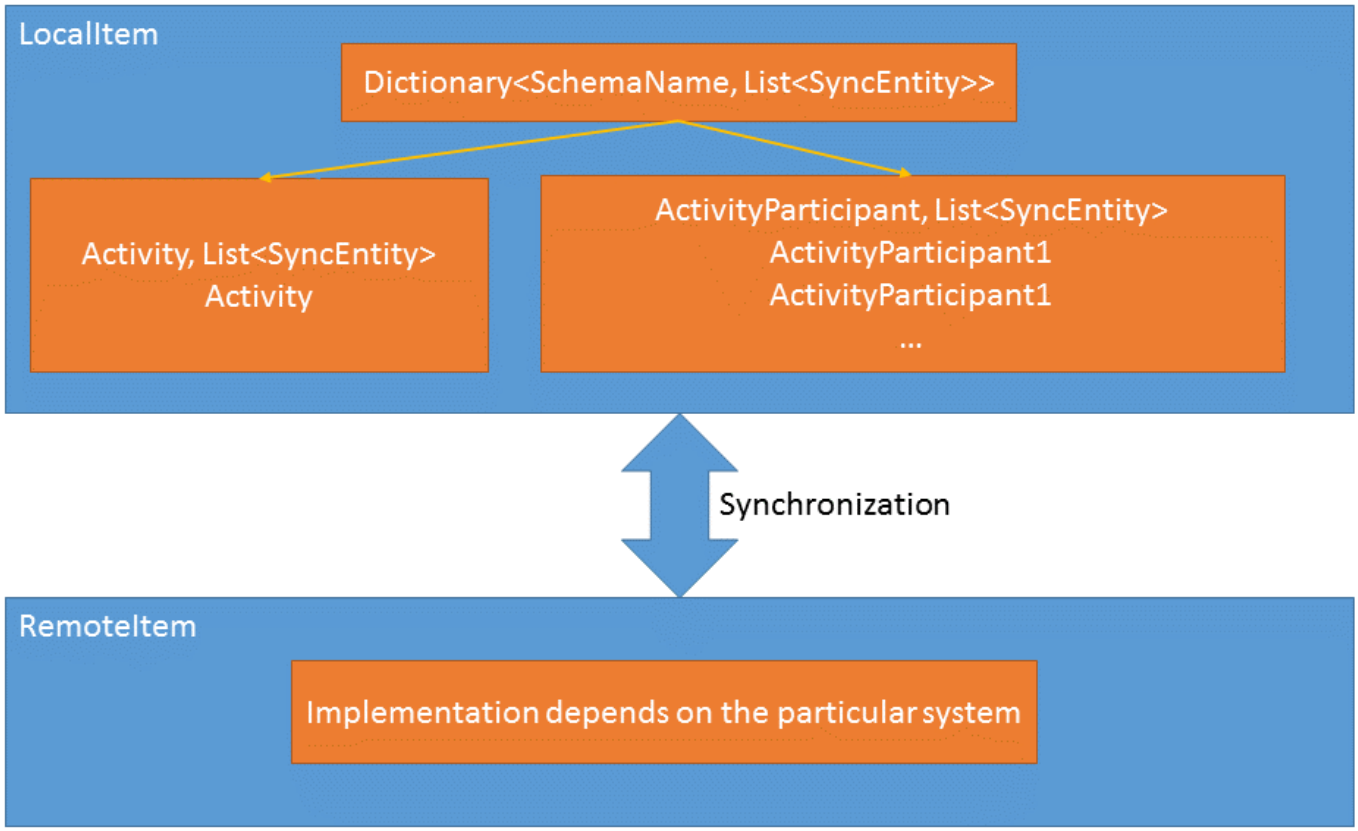
Synchronization example

An activity and participants are synchronized into one Google-calendar task. An activity (*Activity*) and participants (*SyncEntity*) are one element of the synchronization - *LocalItem*.

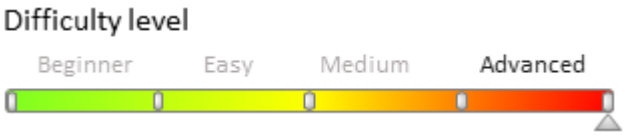
RemoteItem - Google task received outside bpm'online. LocalItem - a set of objects (SyncEntity), to which the Google task is converted.

The synchronization schema is displayed on Fig. 1.

Fig. 1. Synchronization schema



Synchronizing metadata in bpm'online



General information

The auxiliary *SysSyncMetaData* metadata table is used for synchronization, which is the junction between the outer *RemoteItem* table (synchronizing element in external storage) and *LocalItem* (synchronization element in bpm'online). Each table row is represented in the system as an instance of *SysSyncMetaData*. The *SysSyncMetaData* class properties are shown in Table 1.

Table 1. *SysSyncMetaData* class properties.

Parameter	Type	Description
RemoteId	*String*	Element ID in external storage
SyncSchemaName	*String*	Synchronized element schema name.
LocalId	*Guid*	Element ID in local storage
IsLocalDeleted	*Boolean*	It indicates whether an item has been removed from the local storage since the last synchronization. The parameter is updated before the synchronization and application of changes in the local storage. On the basis of its value, when selecting modified elements from local storage, the *SyncEntity* state is set. Obsolete, left for compatibility. *LocalState* is currently used.
IsRemoteDeleted	*Boolean*	It indicates whether an item has been removed from the external storage since the last

		synchronization. Obsolete, left for compatibility. *RemoteState* is currently used.
Version	*Date*	Date of the last element modification.
ModifiedInStoreId	*Guid*	ID of storage in which the last modification was performed.
CreatedInStoreId	*Guid*	ID of storage in which the synchronization element was created.
RemoteStoreId	*Guid*	ID of external storage with which the element was synchronized.
ExtraParameters	*String*	Additional element parameters.
LocalState	*Int*	Element state in local storage (0 - not modified, 1 - new, 2 - modified, 3 - deleted).
RemoteState	*Int*	Element state in external storage (0 - not modified, 1 - new, 2 - modified, 3 - deleted).

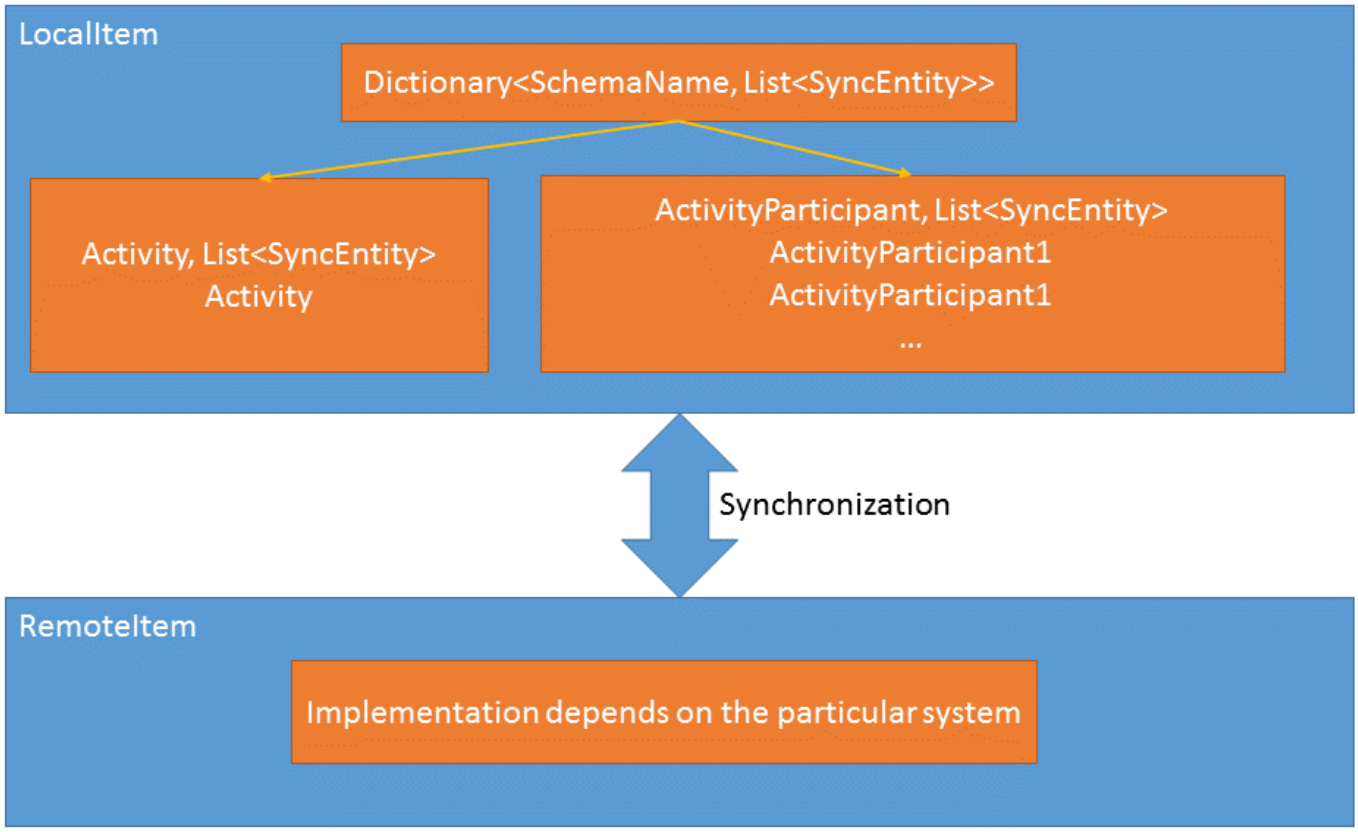
Only information about synchronized elements is stored in the metadata.

There can be multiple metadata table records for a single synchronization element - one for each application object included in a synchronization element.

Activity and participants — a single synchronization element. However, the metadata contains one record for each activity and one record for each participant.

Currently, only one object from external storage is transformed into several bpm'online objects, as shown in Fig. 1.

Fig. 1. Schema of transformation of an external storage object into a local storage object.



The metadata system for a single synchronization element is represented as the *ItemMetadata* object class (*SysSyncMetaData* element collection). Metadata management is carried out through the class that implements the *IReplicaMetadata* interface. An instance of the class that implements the *IReplicaMetadata* interface is created via the *MetaDataStore* factory class for a particular storage.

MetaDataStore class

Creates the specific class instance that implements the IReplicaMetadata interface for a storage. The class methods are shown in Table 2.

Table 2. *MetaDataStore* class methods

Method	Returned value type	Description
GetReplicaMetadata(Guid localStoreId, Guid remoteStoreId)	*IreplicaMetadata*	Creates the class instance that implements the *IReplicaMetadata* interface for the specific storage.

ItemMetadata class

This class is an indivisible object of metadata synchronization. It contains a set of metadata for each synchronization element (*SysSyncMetadata* element collection). The class properties are shown in Table 3.

Table 3. *SysSyncMetaData* class properties.

Parameter	Type	Description
RemoteId	*String*	Element ID in external storage
RemoteItemName	*String*	Element name in external storage

IReplicaMetadata interface

This class implements the *IReplicaMetadata* interface, encapsulates the synchronization metadata and works with *ItemMetadata* objects. The interface properties are shown in table 4 and methods in table 5.

Table 4. The *IReplicaMetadata* interface properties

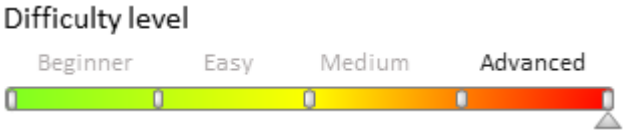
Parameter	Type	Description
RemoteStoreId	*Guid*	External storage ID.
LocalStoreId	*Guid*	Local storage ID.

Table 5. The *IReplicaMetadata* interface methods

Method	Returned value type	Description
FindItemStore (string remoteItemId)	*ItemMetadata*	Finds and returns the *ItemMetadata* synchronization metadata object by an ID in the *remoteItemnId* external storage.
UpdateItemMetadata (ItemMetadata oldItemMetaDatas, IRemoteItem remoteItem, LocalItem localItem, bool changesToBpm)	*Void*	Updates metadata after synchronization.
EnumerateItemsWithChangesInBpm (SyncContext context)	*IEnumerable<ItemMetadata>*	Returns a collection of *ItemMetadata* objects that have been modified since the last synchronization and not processed during the current synchronization session.
CollectChangesInSyncedEntities (UserConnection userConnection, string schemaName, DateTime lastSyncVersion)	*Void*	Updates metadata for synchronization elements modified in bpm'online. If an element has been modified since the last SysMetadata synchronization, the *Version* column will be filled in with the date of element modification. The

		ActualizeSysSyncMetaData procedure is used to update metadata.
CollectNewDetailsForSyncedEntities (UserConnection userConnection, DetailEntityConfig detailEntityConfig, string remoteItemName)	*Void*	Creates new records in the *SysSyncMetaData* table for the synchronization element details.
TryResolveRemoteId (Guid localId, out string remoteId)	*Boolean*	Returns the element ID in the external *remoteId* storage from metadata by a unique *localId* synchronization element in bpm'online. If an element is marked as deleted, the *remoteId* wil not be returned, and the method will return *false*.
TryResolveExtraParameters (Guid localId, out string extraParameters)	*Boolean*	Returns additional *extraParameters* parameters for the synchronization element by *localId*. If *extraParameters* are not found, the method returns *false*.

Synchronizing tasks with MS Exchange



General information

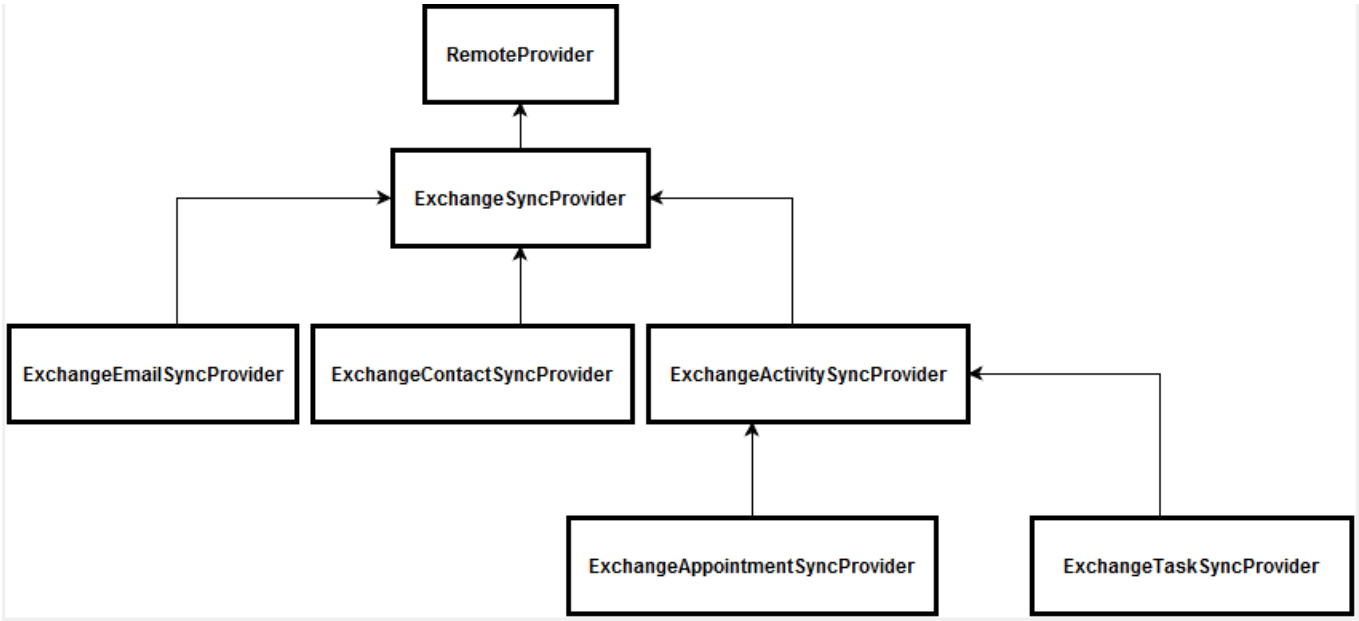
Integration with various entities of MS Exchange via EWS protocol (Exchange Web Services) is supported by the Sync Engine synchronization mechanism. This article describes synchronization of tasks between bpm'online and MS Exchange. The task synchronization algorithm is no different from that described in the "**Bpm'online synchronization with external storages**" article. The process runs in three stages:

1. Retrieving changes from MS Exchange and applying them;
2. Retrieving changes from bpm'online and applying them
3. Creating new tasks from bpm'online in MS Exchange.

Integration classes

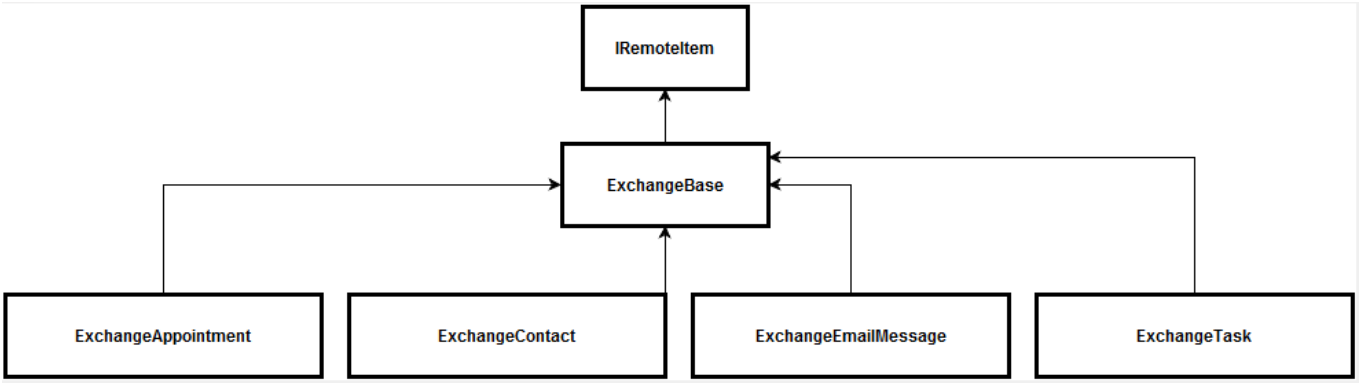
As described in the "**Bpm'online synchronization with external storages**" article, in order to implement an integration using this mechanism, you need a class that implements the logic of the external storage (an heir of the *RemoteProvider* class). The hierarchy of provider classes is shown in figure 1. You also need a class that implements the *IRemoteItem* interface, which represents a single instance of a synchronization item (in this case — the MS Exchange task). The *RemoteItem* hierarchy is shown in figure 2.

Fig. 1. RemoreProvider hierarchy schema



The ExchangeTaskSyncProvider class is the service provider for the MS Exchange external storage. This class implements the logic of selecting data and saving changes in bpm'online and MS Exchange. The *ExchangeTask* class implements the *IRemoteItem* interface. The logic of filling in data in the corresponding systems is implemented in it.

Fig. 2. RemoteItem hierarchy schema



Synchronized data

The correspondence of bpm'online objects to the *ExchangeTask* class fileds is shown on table 1.

Table 1. The correspondence of bpm'online objects to the *ExchangeTask* class fileds

Bpm'online object	Object field	*ExchangeTask*
Activity	*Title*	*Subject*
	StartDate	*StartDate*
	DueDate	*CompleteDate* or *DueDate* depending on whether a task is finished or not.
	Priority	*Importance*
	Status	*Status*
	RemindToOwner	*IsRemindSet*
	RemindToOwnerDate	*ReminderDueBy*

Notes	*Body.Text*

Logic of selecting data for synchronization

To select changes to the list of tasks selected for MS Exchange folder synchronization, use the following terms: select tasks for MS Exchange, which were modified after the last task synchronization or an MS Exchange task, which was not marked as synced. The MS Exchange task which were modified have corresponding activities in bpm'online. The updated changes are applied in the corresponding system.

When you select modified bpm'online activities, select the following:

- activities that are recorded in the metadata synchronization as MS Exchange tasks
- activities that have the current user as an author
- activities with a date of the last modification that does not correspond to the date of the last synchronization.

When selecting new bpm'online activities, configure a set of common and custom filters. The main filter conditions are:

1. Activity type is not "email".
2. Activity does not have the [Display in calendar] checkbox selected.

A user can specify activity groups that will be exported from bpm'online.

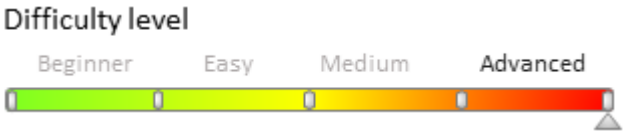
Extra

Filling in the [Start Date] and [Due Date] fields

The *ExchangeTask* object has several features for working with start date and due date:

- These fields are stored without time values. If you change a task in MS Exchange after synchronization, bpm'online will apply the date from the MS Exchange task, and the time from the bpm'online activity.
- The due date in *ExchangeTask* has two fields: *due date* and *complete date*.
- The start date and due date are optional in MS Exchange. If either of them is not filled in, the current date is set. Due to this, conflicts may arise, as both the start date and due date are required in bpm'online, and the start date should be earlier than the due date.

Synchronizing email with MS Exchange



General information

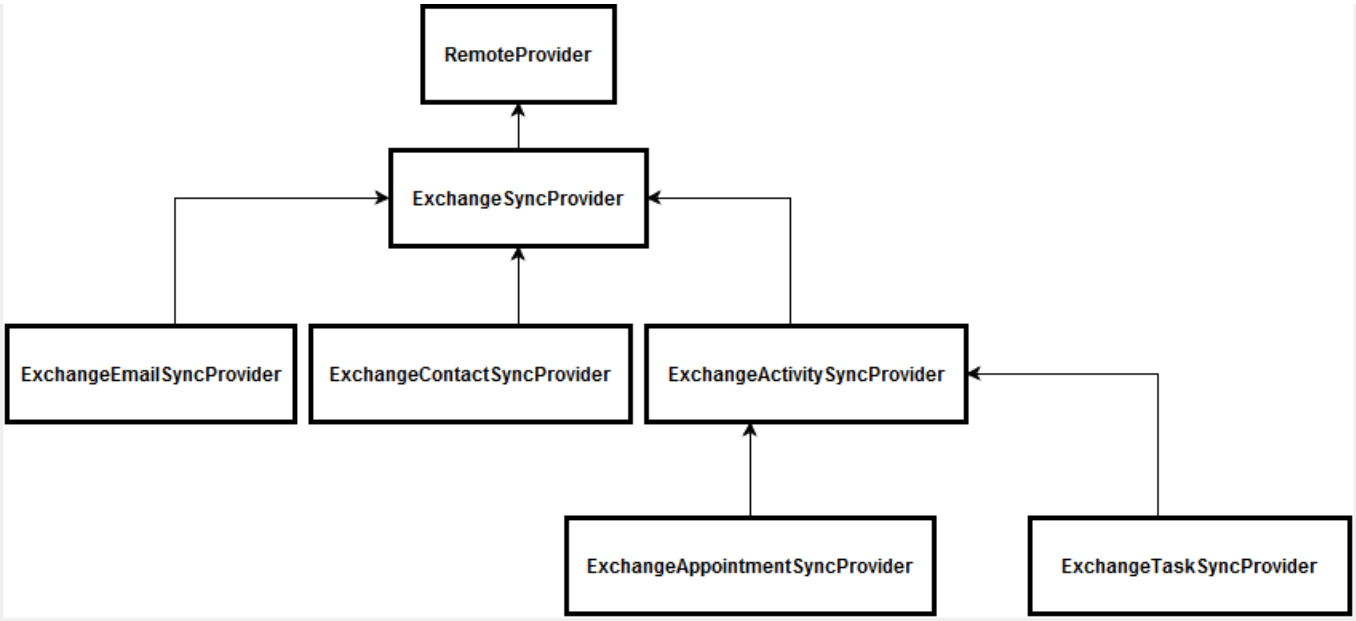
Synchronization with various services of MS Exchange via EWS protocol (Exchange Web Services) is supported by the Sync Engine synchronization mechanism. This article describes the synchronization of email in bpm'online with MS Exchange. Email in bpm'online is synchronized only from MS Exchange. Since emails can no longer be modified after they have been sent, only the emails that have not been synchronized previously are synchronized. The main difference between the email synchronization mechanism and integration is the email search engine in bpm'online. Since the same email can be synchronized on behalf of any of the recipients or even via IMAP protocol, metadata synchronization can not be used for searching for previously synchronized emails. Use *subject*, *send date* and *message* to search for emails . All markups and spaces are removed from the message. To speed up the search, use the md5 hash that is stored in the *MailHash* column of the *Activity* object.

The second difference of this synchronization is that attachments are synchronized by a separate process, after all emails are processed. This is done in order to make the attachment download time not affect the email save time.

Integration classes

As described in the "**Bpm'online synchronization with external storages**" article, to implement integration using this mechanism, a class is required that implements the logic of working with external storage (*RemoteProvider* heir) and a class that implements the *IRemoteItem* interface, which is an instance of the synchronization element (in our case — email MS Exchange).

Fig. 1. RemoreProvider hierarchy schema



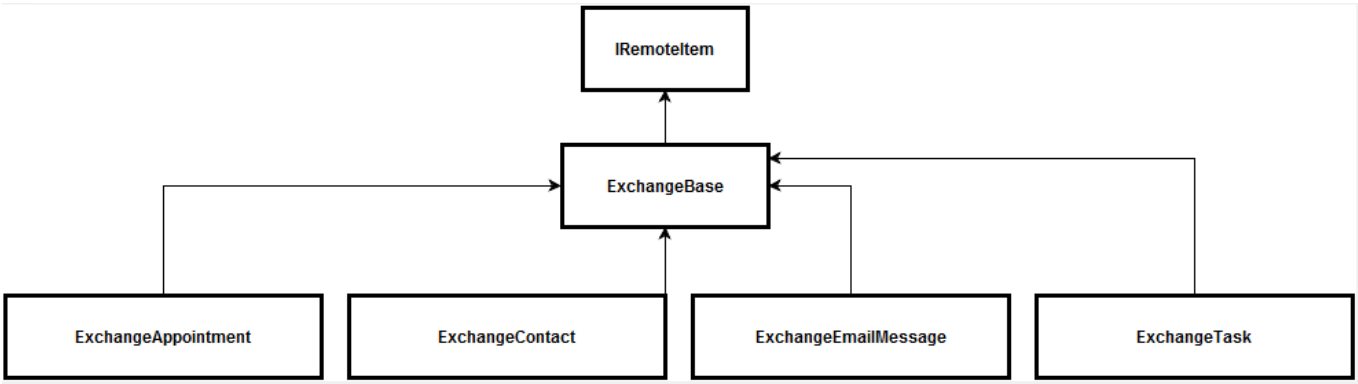
The *ExchangeEmailSyncProvider* class is the service provider for the exchange external storage. This class implements the logic of selecting data from MS Exchange.

The *ExchangeEmailMessage* class implements the IRemoteItem interface, in which the logic of filling in data in bpm'online objects is implemented.

The *ExchangeUtility* class contains methods for the EWS API library and the methods used to download email attachments.

The *ExchangeEmailMessageUtility* class contains methods for converting the lookup values of the email fields.

Fig. 2. RemoteItem hierarchy schema



Synchronized data

The correspondence of bpm'online objects to the *EmailMessage* class fields is shown on table 1.

Table 1. The correspondence of bpm'online objects to the *EmailMessage* class fields

Bpm'online object	Object field	EmailMessage corresponding field

Activity	*Title*	*Subject*
	Body	*Body.Text*
	Sender	*Sender*
	Recepient	*ToRecepients*
	CopyRecepient	*CcRecepients*
	BlindCopyRecepient	*BccRecepients*
	SendDate	*DateTimeSpent*
	Priority	*Importance*
	DueDate, StartDate	*DateTimeReceived*
ActivityFile	*Name*	*Name*
	Data	*Content*
	Size	*Content.Length*

Logic of filling in email participants

For an email to be displayed correctly only for users who have synchronized it, the following mechanism of filling in the [Activity participants] detail has been implemented. Conventionally, this logic can be divided into two parts:

1. Adding participants to a new email.
2. Updating the list of participants when an email changes (including re-synchronization).

Adding participants to a new email

The main value that affects who becomes the participants of an email is the [Communication] contact detail. If a contact has an email address in the [Communication] detail, and this email is listed in one of the email address fields ([From], [To], [CC], [BCC]), the contact can be added to the participants. Additionally, a check is made whether there is a system user for this contact who is not a portal user. A user is added to the participants only after they have synchronized their email.

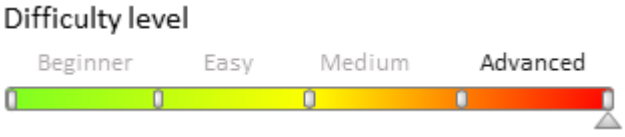
Updating the list of participants

For a user to become a participant after the synchronization of an existing email, the list of participants is updated - all participants who are not bpm'online users are removed from the detail, and the algorithm of filling in detail for a new email runs. Thus, the users who have previously synchronized the email remain as participants, a new user is added, and the contact list is updated.

Logic of selecting data for synchronization

When selecting emails for synchronization with MS Exchange, use the following filter set: select emails that have been modified since the last synchronization and are not drafts. There is a limit for synchronization folders: "Deleted" and "Conflicting elements" folders do not participate in synchronization. When selecting emails the synchronization metadata is not taken into account. Always "save changes in bpm'online". When processing each email, the system first checks for emails in bpm'online. If an email already exists in bpm'online, the participants are updated. If not, a new email is created. At the end of the synchronization, the system adds a task to synchronize attachments.

Synchronizing contacts with MS Exchange



General information

Integration with various entities of MS Exchange via EWS protocol (Exchange Web Service) is supported by the Sync Engine synchronization mechanism. This article describes synchronization of contacts between bpm'online and MS Exchange. The task synchronization algorithm is no different from that described in the article about Sync Engine synchronization. The process runs in three stages:

1. Retrieving changes from MS Exchange and applying them
2. Retrieving changes from bpm'online and applying them
3. Creating new contacts from bpm'online in MS Exchange.

Integration classes

As described in the "**Bpm'online synchronization with external storages**" article, to implement integration using this mechanism, a class is required that implements the logic of working with external storage (*RemoteProvider* heir) and a class that implements the *IRemoteItem* interface, which is an instance of the synchronization element (in our case — MS Exchange contact).

Fig. 1. RemoreProvider hierarchy schema

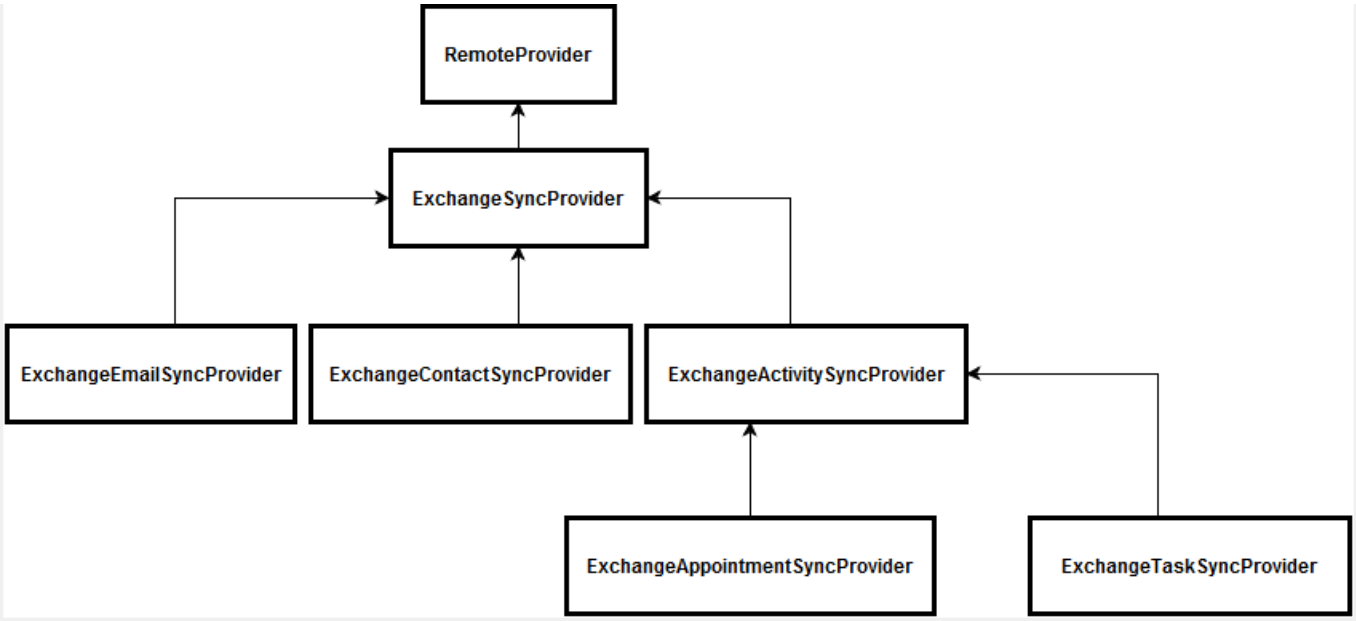
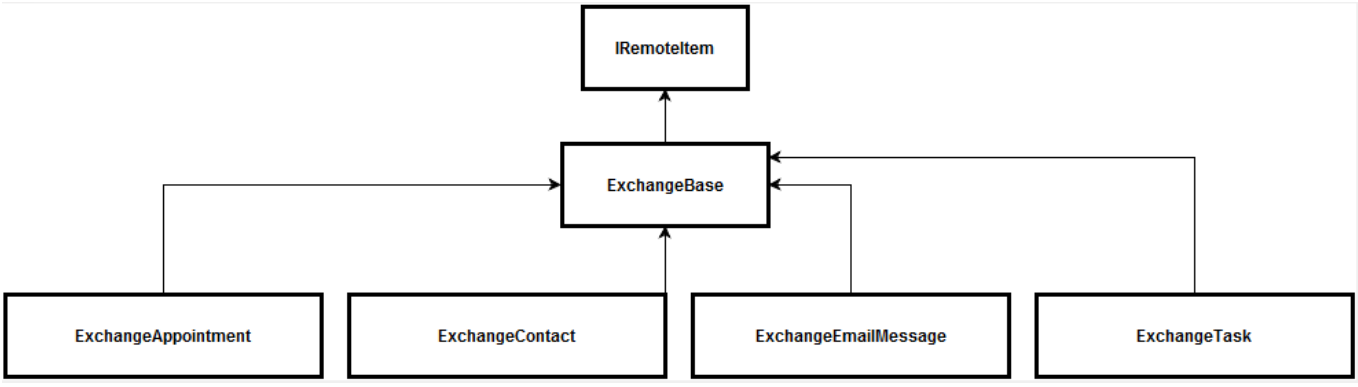


Fig. 2. RemoteItem hierarchy schema



The following classes are used for contact synchronization:

- The *ExchangeContactSyncProvider* class is the service provider for the MS Exchange external storage. This class implements the logic of selecting data and saving changes in bpm'online and MS Exchange.
- The *ExchangeContact* class implements the *IRemoteItem* interface. The logic of filling in data in the corresponding systems is implemented in it.
- The *ExchangeAddressDetailsSynchronizer* class contains methods for converting contact addresses.
- The *ExchangeEmailAddressDetailsSynchronizer* class contains methods for converting contact email

addresses.

- The *ExchangePhoneNumbersDetailsSynchronizer* class contains methods for convertingcontacts phones.

The logic of filling in details is located in separate classes, as there are significant differences in the formats of data storage in bpm'online and MS Exchange. Additional conversion is required.

Synchronized data

The correspondence of bpm'online objects to the Contact MS Exchange class fields is shown on table 1.

Table 1. The correspondence of bpm'online objects to the Contact MS Exchange class fields

Bpm'online object	Object field	The Contact MS Exchange class field
Contact	*Name*	*DisplayName*
	Surname	*Surname*
	GivenName	*GivenName*
	MiddleName	*MiddleName*
	Account	*CompanyName*
	JobTitle	*JobTitle*
	Department	*Department*
	BirthDate	*Birthday*
	SalutationType	*TitleTag*
	Gender	*GenderTag*
ContactCommunication	*Number*	The *PhoneNumbers* collection values
	CommunitactionType	The *PhoneNumbers* collection value key
ContactAddresses	*City*	The *PhysicalAddresses* collection element *City* field
	Country	The *PhysicalAddresses* collection element *CountryOfOrigin* field
	Region	The *PhysicalAddresses* collection element *State* field
	Address	The *PhysicalAddresses* collection element *Street* field
	Zip	The *PhysicalAddresses* collection element *PostalCode* field
	AddressType	The *PhysicalAddresses* collection value key

The correspondence of communication types is shown in Table 2.

Table 2. The correspondence of communication types of bpm'online to MS Exchange

Bpm'online communication type	MS Exchange communication type
Email	*EmailAddress1, EmailAddress2, EmailAddress3*
WorkPhone	*BusinessPhone, BusinessPhone2*
HomePhone	*HomePhone*
MobilePhone	*MobilePhone*

The correspondence of addresses is shown in Table 3.

Table 3. The correspondence of addresses of bpm'online to MS Exchange

Bpm'online address type	MS Exchange address type
HomeAddress	*Home*
BusinessAddress	*Business*

Logic of selecting data for synchronization

To select changes to the list of contacts selected for MS Exchange folder synchronization, use the following terms: select contacts for MS Exchange, which were modified after the last contact synchronization or an MS Exchange contact, which was not marked as synced. The contacts which were modified have corresponding contacts in bpm'online. The updated changes are applied in the corresponding system.

When you select bpm'online contacts for synchronization, select the following:

- contacts that have the current user as an author
- contacts with a date of last modification that does not correspond to the date of the last synchronization.
- contacts that were not used on the first step of synchronization

User settings also affect the rules for selecting new contacts in bpm'online. Three settings are available:

1. Synchronize employees contacts. When you select this setting, the "Contact type" filter will be added to the request, and only the "Employee" type contacts will be synchronized.
2. Synchronize customers contacts. When you select this setting, the "Contact type" filter will be added to the request, and only the "Customer" type contacts will be synchronized.
3. Sync contacts from certain groups. When you select this setting, the selected contact group filters will be added to the request.

Additional features

Binding contact and account

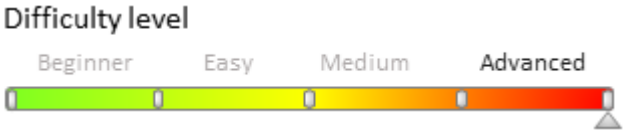
If an MS Exchange contact has the *CompanyName* field filled in, then there are three possible options for filling in the [Account] lookup field:

1. Always bind. If a nonexistent account is set, it will be created. This is the default option.
2. Bind, if an account exists. Same as above, but nonexistent accounts are not created.
3. Never bind. The [Account] field will not be filled in during synchronization.

Using the advanced contact keys in external storage

A situation may occur when there is a large number of MS Exchange contacts and of them will receive the same ID. As a result, synchronization may not correctly identify the appropriate contact in bpm'online. To work around this situation, there are advanced external keys, which can be enabled by the [Use composite IDs for MS Exchange synchronized contacts] setting. Setting code - *UseComplexExchangeContactId*. After enabling it, you may need to resynchronize.

Synchronizing appointments with MS Exchange



General information

Integration with various entities of Exchange via EWS protocol (Exchange Web Services) is supported by the Sync Engine synchronization mechanism. This article describes synchronization of appointments between bpm'online and MS Exchange.

Bpm'online appointment synchronization is performed only for new activities or when the *Title, Location,*

StartDate, DueDate, Priority, Notes fields are modified. The hash stored in the additional metadata parameters (in the *ExtraParameters* field) is formed according to these fields. If an appointment has been changed in bpm'online, and the hash for *ExtraParameters* does not match the new hash, this appointment should be synchronized.

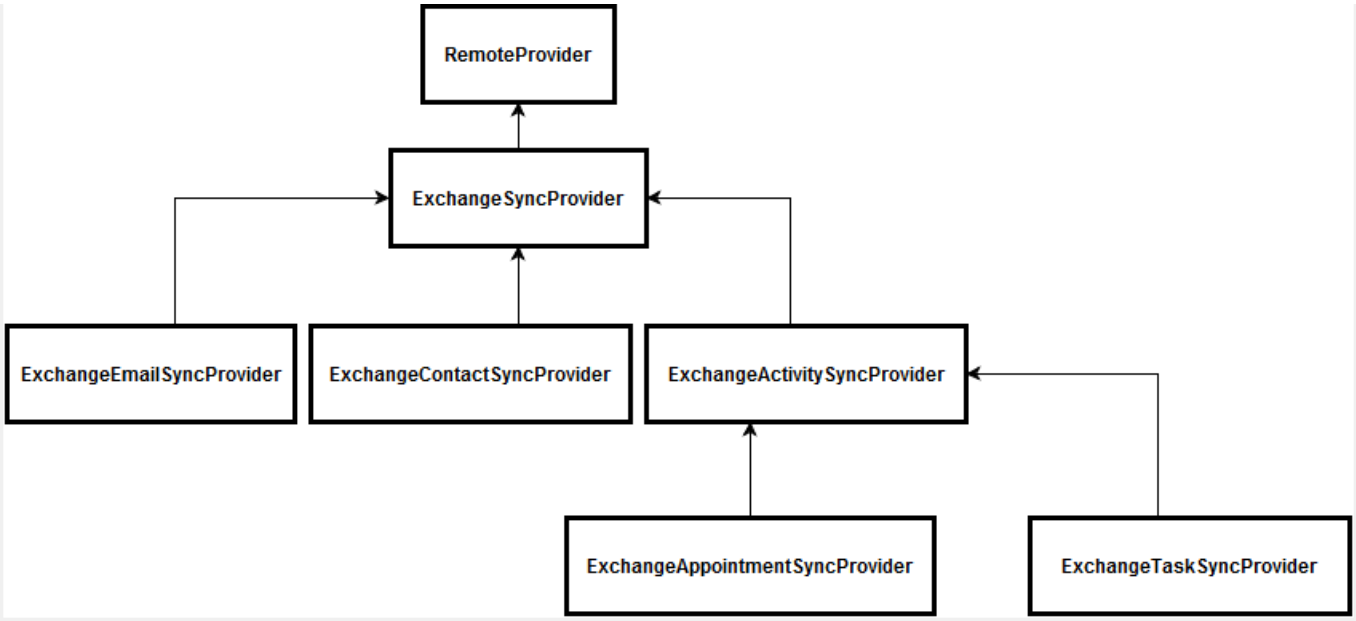
The appointment synchronization algorithm is no different from that described in the "**Bpm'online synchronization with external storages**" article. The process runs in three stages:

1. Retrieving changes from MS Exchange and applying them
2. Retrieving changes from bpm'online and applying them
3. Creating new appointments from bpm'online in MS Exchange.

Integration classes

As described in the "**Bpm'online synchronization with external storages**" article, to implement integration using this mechanism, a class is required that implements the logic of working with external storage (*RemoteProvider* heir) and a class that implements the *IRemoteItem* interface, which is an instance of the synchronization element (in our case — MS Exchange appointment).

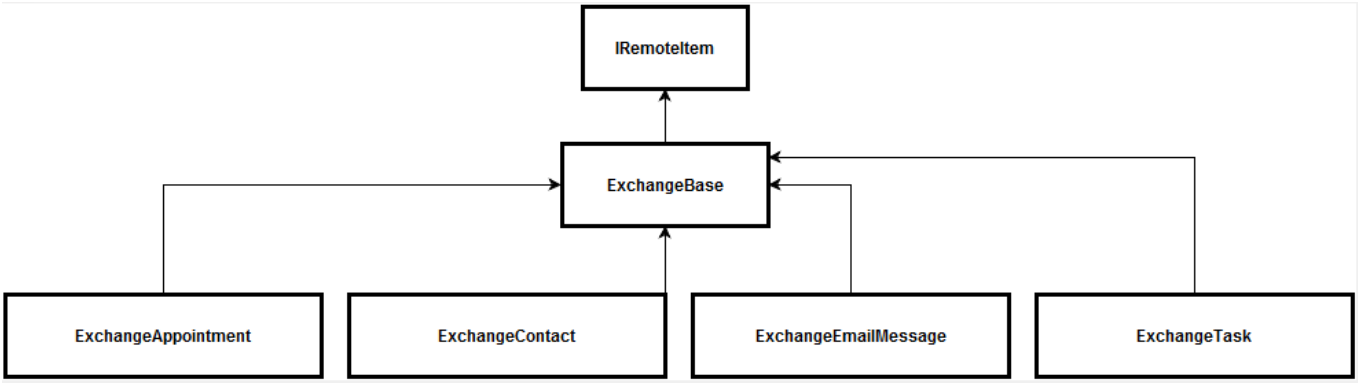
Fig. 1. RemoreProvider hierarchy schema



The *ExchangeAppointmentSyncProvider* is the provider used to work with the Exchange external storage. It contains the logic of selecting data and saving changes in bpm'online and Exchange.

The *ExchangeAppointment* class implements the IRemoteItem interface, in which the logic of filling in data in bpm'online objects is implemented.

Fig. 2. RemoteItem hierarchy schema



Synchronized data

The correspondence of bpm'online objects to the *ExchangeAppointmrnt* class fileds is shown on table 1.

Table 1. The correspondence of bpm'online objects to the *ExchangeAppointment* class fileds

Bpm'online object	Object field	MS Exchange Appointment corresponding field
Activity	*Title*	*Subject*
	Location	*Location*
	StartDate	*StartDate*
	DueDate	*CompleteDate* or *DueDate* depending on whether an appointment is finished or not.
	Priority	*Importance*
	Status	Filled in as follows: If the status is not specified and the due date is later than the current date — bpm'online sets the "New Appointment" status. If the due date is earlier than the current date, the status is set as a closed appointment with the "Information received" status.
	RemindToOwner	*IsReminderSet*
	RemindToOwnerDate	*ReminderDueBy*
	Notes	*Body.Text*
ActivityParticipant	*InviteResponse*	If the checkbox in MS Exchange is selected that identifies that an appointment was received and the user is its owner, and the "Appointment confirmed" checkbox is selected. Otherwise, it the checkbox is selected that identifies that the appointment was canceled.

Logic of selecting data for synchronization

To select changes to the list of appointments selected for MS Exchange folder synchronization, use the following terms: select appointments for MS Exchange, which were modified after the last contact synchronization or an MS Exchange appointment, which was not marked as synced. The appointments which were modified have corresponding contacts in bpm'online. The updated changes are applied in the corresponding system.

When you select modified bpm'online activities, select the following:

- activities which are recorded in the synchronization metadata as MS Exchange appointments via *RemoteId* (determined by a unique appointment ID in the *ICalId* calendar);
- activities with a date of the last modification that does not correspond to the date of the last synchronization.
- one appointment in bpm'online corresponds to several appointments in MS Exchange for each participant.

When selecting new bpm'online activities, configure a set of common and custom filters. The main filter conditions are:

1. Activity type is not "Email".
2. Activity has the [Display in calendar] checkbox selected.

A user can specify activity groups that will be exported from bpm'online.

Logic of selecting appointment participants

When you synchronize an activity from MS Exchange to bpm'online, only contacts that have email addresses from the list of appointment participants in MS Exchange are added to the [Participants] detail.

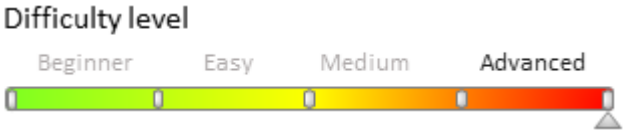
When you synchronize activities from bpm'online to MS Exchange, the appointment participants for MS Exchange are filled in with primary contact email addresses.

Data Enrichment and Prediction

Contents

- **Contact data enrichment from emails**
- **Machine learning service**
- **Creating data queries for the machine learning model**

Contact data enrichment from emails



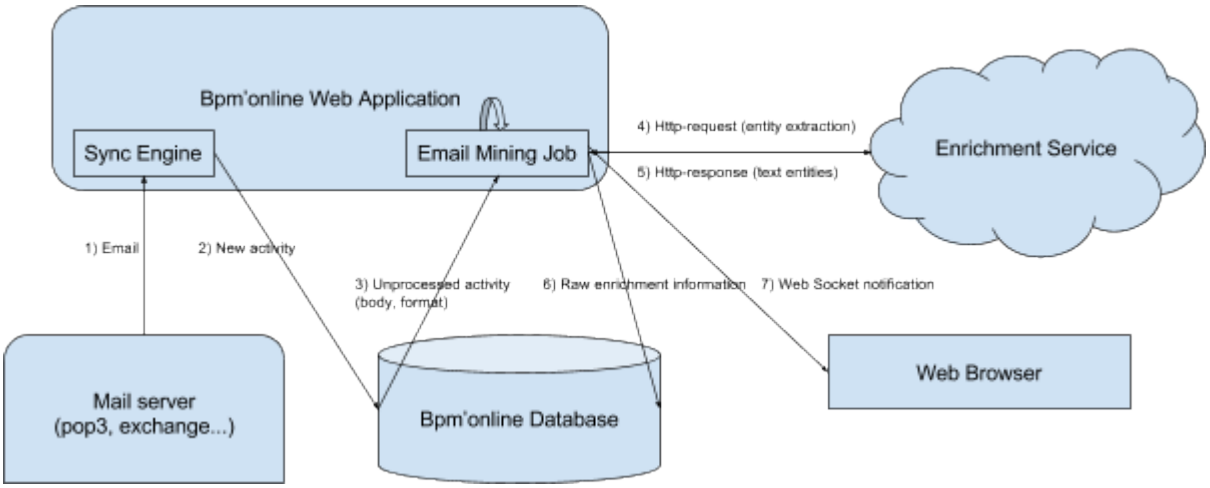
Introduction

Data enrichment from emails is available in bpm’online since version 7.10.0. The system scans emails and identifies information that can be used to enrich contact data.

The enrichment process

The main data enrichment stages (Fig. 1):

Fig. 1. Receiving contact/account data from an email



1. The existing **Sync Engine synchronization method** connects to the mail server. The mail server sends new emails to the Sync Engine.

2. Sync Engine saves the received emails in the database as activities with the *Email* type.

3. The bpm'online planner periodically performs a task that starts the *Email Mining Job* process. This process selects some of the last (by creation date) activities with the *Email* type that were not previously processed by it. From each activity record, the body of the email and its format (plain–text or html) are selected.

4. The *Email Mining Job* process for each selected email sends an http request to the *Enrichment Service* cloud service.

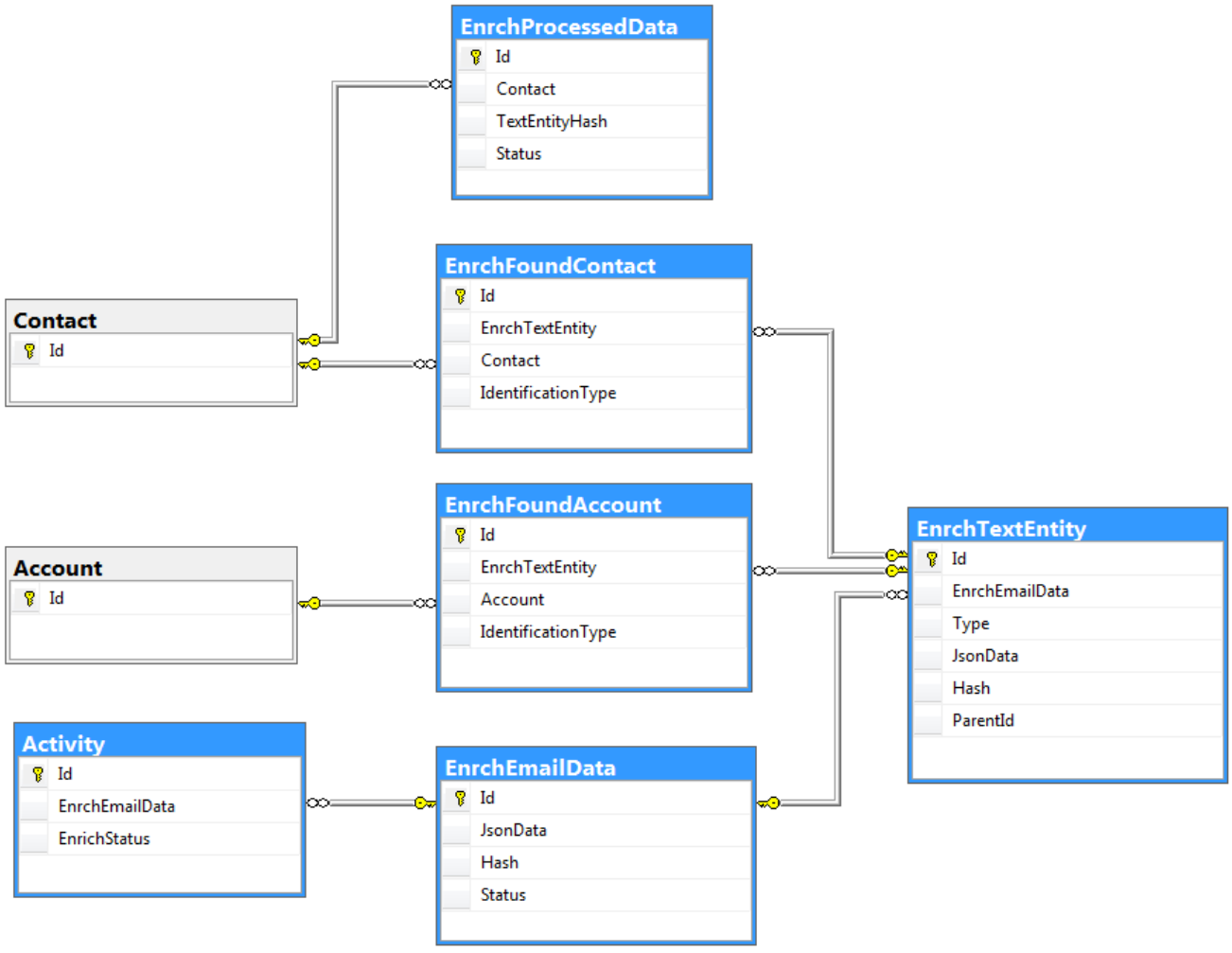
5. *Enrichment Service* performs the following actions:

- selects a chain of individual emails (replies) from the email;
- selects a signature for each email (signature);
- separates the entity (entity extraction) from the signature – contact (name, position), telephones, emails and web addresses, social networks, other means of communication, addresses, organization names.

Enrichment Service returns the gathered in the http–response as a specific structure in the JSON format.

6. The *Email Mining Job* process parses the structure received from the service and stores it in bpm'online tables (see Figıo 2).

Fig. 2. The data structure for storing entities selected from the email



The main purpose of the tables shown in Fig. 2:

- *EnrchTextEntity* – stores information about one entity selected from an email. The *Type* field defines the type of this entity (contact, communication, address, organization, etc.). The data itself is stored in the JSON format in the *JsonData* field.
- *EnrchEmailData* – defines a set of information for enrichment selected from a single email.
- *EnrchFoundContact* – a contact in bpm'online, identified by the data selected from the email. Stores a link to the bpm'online contact and *EnrchTextEntity* of the *Contact* type.
- *EnrchFoundAccount* – stores information about the identified bpm'online account (similar to the *EnrchFoundContact* table).

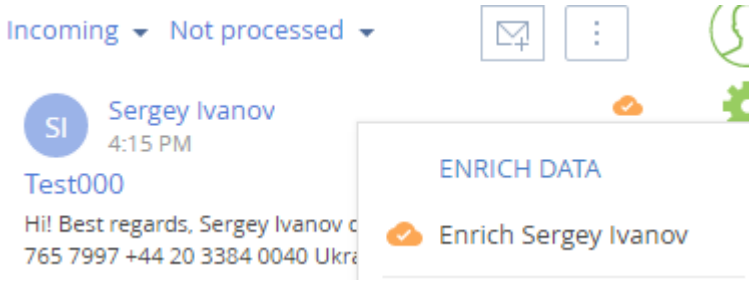
- *Activity* – the fields added to the existing activity table show the connection between the *Email* activity and the *EnrchEmailData* objects with the current status of the information extraction process.
- *EnrchProcessedData* – contains information about processed data, either accepted or rejected by the user in the enrichment process.

7. The *Email Mining Job* process notifies the user about the extraction process being finished. Messages are sent via the websocket channels to users who see the messages being processed in the communication panel. If the email contains information that may be used to enrich an associated contact, (or used to create a new contact altogether), the corresponding icon is displayed in the application interface in the upper right corner of the email (Fig. 3).

Fig. 3. Enrichment availability icon

An email like that will enable the user enrich or create a new contact of the system (Fig. 4).

Fig. 4. Enrichment action



System settings

Enrichment system settings:

- *TextParsingService* – the address of the *Enrichment Service* cloud–service for data enrichment. Filled automatically for on–demand users. Required field.
- *CloudServicesAPIKey* – the key for accessing the cloud service API. Filled automatically for on–demand users. Required field.
- *EmailMiningPackageSize* – the number of emails processed at once. The *Email Mining Job* process will process as many emails each time as it is specified in this system setting. Default value – 10.
- *EmailMiningPeriodMin* – the frequency (in minutes) of running the *Email Mining Job* process.

ATTENTION

If the value of *EmailMiningPeriodMin* is less than or equal to zero, then the process will not be scheduled and the functionality will be disabled. To re–enable the process, set the value of the setting >=1, restart the application pool of the application, go to the login page and enter the application.

- *EmailMiningIdentificationActualPeriod* – the period of relevance (in days) of contacts / accounts identification. If the specified period has expired and a new email is processed for the previously identified contact, the identification will be made again.

Identification sequence

Identifying contacts

1. Search by full name.
2. Search by name and last name.
3. Search by email addresses. Only those email addresses that do not belong to free or temporary email services are taken into account.
4. Search by phone. The search takes place only for the last digits of the contact's phone numbers.

If at any of the identification stages a data duplicate is detected, the identification process will be stopped.

Identifying accounts

1. Search by the [Name] or [Alternative name] columns (case– insensitive).
2. Search by web address.
3. Search by email addresses. Only those email addresses that do not belong to free or temporary email services are taken into account. From the email address, the domain is allocated and the search for the communication facilities of the account for the filter starts with one of the following domain variants: http://<domain>, https://<domain>, http://www.<Domain>, https://www.<domain>, www.<domain>, <domain>.

If at any of the identification stages a data duplicate is detected, the identification process will be stopped.

Hashing information

The information extracted from the email is hashed. As a result, in the *EnrchTextEntity* and *EnrchEmailData* tables, a hash value is written in the *Hash* field that uniquely identifies the given unit or set of extracted data in the system. This allows for two important improvements: resource savings when re–identifying contacts / accounts from a set of extracted information and grouping the information received for a contact.

Re–identification of contacts/accounts

For example, the system received an email with the signature of "John Smith Jr.", the telephone number "123–45–67" and the address "71 Pilgrim Avenue Chevy Chase, MD 20815". For the current data set, the system computed a hash of "Hash1" and recorded it in the *Hash* field of the *EnrchEmailData* table based on its contents. The identification of the contact revealed the contact "John Smith" in the system and recorded the result in the *EnrchFoundContact* table.

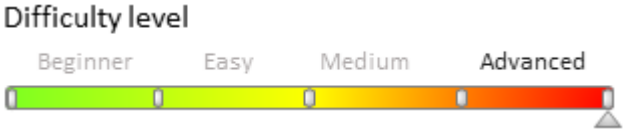
After a while the system received another email with a signature, which mentions the "John Smith Jr." contact with the same phone and address. The system calculated the same hash for the current data set – "Hash1", because the incoming hash data has not changed. Instead of creating new records in the *EnrchEmailData* and *EnrchTextEntity* tables and re–identifying this contact, the system found the previously created record in the Hash field of the *EnrichEmailData* table and wrote a reference to this record in the *Activity* table.

This process saves the amount of data stored and does not produce resource–intensive contact identification requests.

Grouping the highlighted information for a contact

Since each unit of the allocated information in *EnrchTextEntity* has a hash code based on its contents, when enriching the data of an existing contact, it becomes possible to use the information found in all the email in which it participated. When you select the data for enrichment, it is grouped by the Hash field and will not be duplicated.

Machine learning service



Introduction

The machine learning service (or lookup value prediction service) uses statistical analysis methods for machine learning based on historical data. For example, a history of customer communications with customer support is considered historical data in bpm'online. The message text, the date and the account category are used. The result is the [Responsible Group] field.

Bpm'online interaction with the prediction service

There are two stages of model processing in bpm'online: training and prediction.

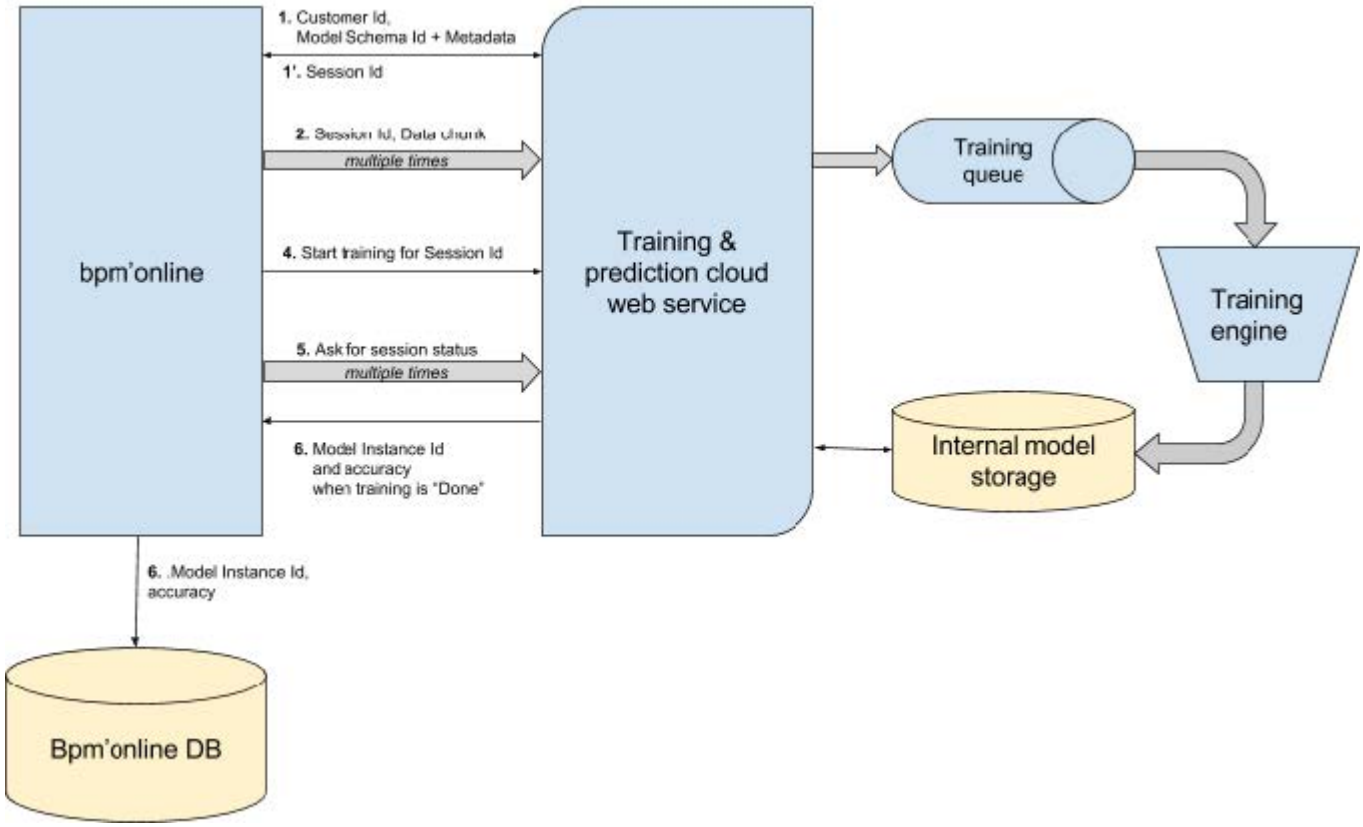
Prediction model is the algorithm which builds predictions and enables the system to automatically make decisions based on historical data.

Training

The service is “trained” at this stage (Fig. 1). Main training steps:

- Establishing a session for data transfer and training.
- Sequentially selecting a portion of data for the model and uploading it to the service.
- Requesting to include a model a training queue.
- *Training engine* processes the queue for model training, trains the model and saves its parameters to the local database.
- Bpm'online occasionally queries the service to get the model status.
- Once the model status is set to *Done*, the model is ready for prediction.

Fig. 1. Bpm’online interaction with the prediction service on the training stage

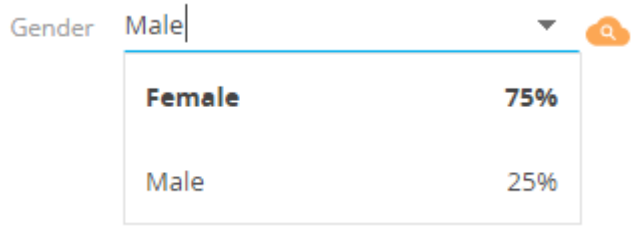


Prediction

The prediction task is performed through a call to the cloud service, indicating the Id of the model instance and the data for the prediction. The result of the service operation is a set of values with prediction probabilities, which is stored in bpm'online in the *MLPrediction* table.

If there is a prediction in the *MLPrediction* table for a particular entity record, the predicted values for the field are automatically displayed on the edit page (Fig. 2).

Fig. 2. Displaying prediction data



Bpm'online settings and data types for working with the prediction service

Bpm'online setup

The following data is provided for working with the prediction service in bpm'online.

1. The *CloudServicesAPIKey* system setting authenticates the bpm'online instance in cloud services.
2. The record in the [ML problem types] (*MLProblemTypes*) lookup with the populated [ServiceUrl] field is the address of the implemented prediction service.
3. The model records in the [ML model] (*MLModel*) lookup that contain information about the selected data for the model, the training period, the current training status, etc. For each model, the *MLProblemType* field must contain a reference to the correct record of the [ML problem types] lookup.
4. The *MLModelTrainingPeriodMinutes* system setting determines the frequency of model synchronization launch.

The MLModel lookup

The primary fields of *MLModel* lookup are given in Table 1.

Table 1. – Main *MLModel* lookup fields

Field	Data type	Purpose
Name	String	Model name
ModelInstanceUId	Unique identifier	The identifier of the current model instance.
TrainedOn	Date/time	The date/time of instance training.
TriedToTrainOn	Date/time	The date/time of last training attempt.
TrainFrequency	Integer	Model retraining frequency (days).
MetaData	String	Metadata with selection column types. Uses the following JSON format:

```
{
    inputs: [
        {
            name: "Name of the field 1 in the
data sample",
            type: "Text",
            isRequired: true
        },
        {
            name: "Name of the field 2 in the
data sample",
            type: "Lookup"
        },
        //...
    ],
    output: {
        name: "Resulting field",
        type: "Lookup",
```

```
        displayName: "Name of the column to
display"
    }
}
```

In this code:

- *inputs* – a set of incoming columns for the model.
- *output* – a column, the value of which the model should predict.

Column descriptions support the following attributes:

- *name* – field name from the *TrainingSetQuery* expression.
- *type* – data type for the training engine. Supported values:
  - "Text" – text column.
  - "Lookup" – lookup column.
  - "Boolean" – logical data type.
  - "Numeric" – numeric type.
  - "DateTime" – date and time.
- *isRequired* – mandatory field value (*true / false*). Default value – *false*.

TrainingSetQuery | String | C#-expression of the training data selection. This expression should return the *Terrasoft.Core.DB.Select* class instance. For example:

```
(Select)new Select(userConnection)
.Column("Id")
.Column("Symptoms")
.Column("CreatedOn")
.From("Case", "c")
.OrderByDesc("c", "CreatedOn")
```

> ATTENTION
>
> Select the "Unique identifier" column type in the selection expression. This column should have the *Id* name.

> ATTENTION
>
> If the selection expression contains a column for sorting, then this column must be present in the resulting selection.

You can find examples of data queries creating in the "**Creating data queries for the machine learning model**" article.

RootSchemaUId | Unique identifier | A link to an object schema for which the prediction will be executed.

Status | String | The status of model processing (data transfer, training, ready for forecasting).

InstanceMetric | Number | A quality metric for the current model instance.

MetricThreshold | Number | Lowest threshold of model quality.

PredictionEnabled | Logical | A flag that includes the prediction for this model.

TrainSessionId | Unique identifier | Current training session.

MLProblemType | Unique identifier | Machine learning problem (defines the algorithm and service url

for model training).

A set of classes for training

MLModelTrainerJob: IJobExecutor, IMLModelTrainerJob – model synchronization task

Orchestrates model processing on the side of bpm’online by launching data transfer sessions, starting trainings, and also checking the status of the models processed by the service. Instances are launched by default by the task scheduler through the standard Execute method of the IJobExecutor interface.

Public methods:

IMLModelTrainerJob.RunTrainer() is a virtual method that encapsulates the synchronization logic. The base implementation of this method performs the following actions:

1. Selecting models for training – the records are selected from MLModel based on the following filter:

- The *MetaData* and *TrainingSetQuery* fields are populated.
- The *Status* field is not in the *NotStarted*, *Done* or *Error* state (or not populated at all).
- *TrainFrequency* is more than 0.
- The *TrainFrequency* days have passed since the last training date (*TriedToTrainOn*).

For each record of this selection, the data is sent to the service with the help of the predictive model trainer (see below).

2. Selecting previously trained models and updating their status (if necessary).

The data transfer session for the selection starts for each suitable model. The data is sent in packages of 1000 records during the session. For each model, the selection size is limited to 75,000 records.

MLModelTrainer: IMLModelTrainer – the trainer of the prediction model.

Responsible for the overall processing of a single model during the training stage. Communication with the service is provided through a proxy to a predictive service (see below).

Public methods:

IMLModelTrainer.StartTrainSession() – sets the training session for the model.

IMLModelTrainer.Upload Data() – transfers the data according to the model selection in packages of 1000 records. The selection is limited to 75,000 records.

IMLModelTrainer.BeginTraining() – indicates the completion of data transfer and informs the service about the need to put the model in the training queue.

IMLModelTrainer.UpdateModelState – requests the service for the current state of the model and updates the Status (if necessary).

If the training was successful (*Status* contains the *Done* value), the service returns the metadata for the trained instance, particularly the accuracy of the resulting instance. If the precision is greater than or equal to the lower threshold (*MetricThreshold*), the ID of the new instance is written in the *ModelInstanceUId* field.

MLServiceProxy: IMLServiceProxy – proxy to the prediction service

A wrapper class for http requests to a prediction service.

Public methods:

IMLServiceProxy.UploadData() – sends a data package for the training session.

MLServiceProxy.BeginTraining() – calls the service for setting up training in the queue

IMLServiceProxy.GetTrainingSessionInfo() – requests the current state from the service for the training session.

IMLServiceProxy.SafeClassify(Guid modelInstanceUId, Dictionary data) – calls the prediction service of the field value for a single set of field values for the previously trained model instance. In the *Dictionary data* parameter, the

field name is passed as the key, which must match the name specified in the *MetaData* field of the model lookup. If the result is successful, the method returns a list of values with the *ClassificationResult* type.

Basic properties of the *ClassificationResult* type:

- *Value* – field value.
- *Probability* – the probability of a given value in the [0:1] range. The sum of the probabilities for one list of results is close to 1 (values of about 0 can be omitted).
- *Significance* - the level of importance of this prediction. This is a string enumeration with the following options:
    - High - this field value has a distinct advantage over other values from the list. Only one element in the prediction list can have this level.
    - Medium - the value of the field is close to several other high values in the list. For example, two values in the list have a probability of 0.41 and 0.39, and all the others are significantly smaller.
    - None - irrelevant values with low probabilities.

Expanding the training model logic

The above chain of classes calls and creates instances of each other through the IOC of the *Terrasoft.Core.Factories.ClassFactory* container.

If you need to replace the logic of any component, you need to implement the appropriate interface. When you start the application, you must bind the interface in your own implementation.

Interfaces for logic expansion:

IMLModelTrainerJob – the implementation of this interface will enable you to change the set of models for training.

IMLModelTrainer – responsible for the logic of loading data for training and updating the status of models.

IMLServiceProxy - the implementation of this interface will enable yo to execute queries to arbitrary predictive services.

Auxiliary classes for forecasting

Auxiliary (utility) classes for forecasting enable you to implement two basic cases:

1. Prediction at the time of creating or updating an entity record on the server.
2. Prediction when the entity is changed on the edit page.

While predicting on the bpm'online’s server side, a business process is created that responds to the entity creation/change signal, reads a set of fields, and calls the prediction service. If you get the correct result, it stores the set of field values with probabilities in the *MLPrediction* table. If necessary, the business process writes a separate value (for example, with the highest probability) to the corresponding field of the entity.

To call the prediction from the edit page, do the following:

- Extend the edit page.
- Develop a logic for changing the fields used for the prediction.
- Call the bpm'online web-service to perform the communication logic with the prediction service while preserving the results.
- The result of the call is displayed on the edit page in the predicted field.

As an example, consider expanding the *ContactPageV2* page of the pre-installed *ML* package.

LookupMLPredictor

A utility class that helps to predict the value of a field based on a particular model for a particular entity.

Public methods:

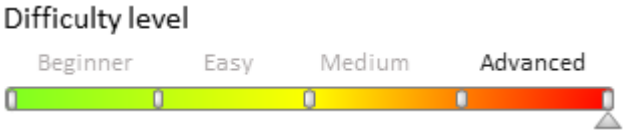
TryLoadModelDataForPrediction() – loads and checks the model from the *MLModel* table (using the *Id*). Returns *true* if the model is trained and the *PredictionEnabled* flag is set for it.

PredictAndSaveResults() – prepares the data for the prediction service, calls it and saves the results in *MLPrediction*. Possible method parameters are listed in table 2.

Table 2. – Main PredictAndSaveResults() method parameters

Name	Description
string schemaName	The name of the schema of the target entity for which the prediction is performed.
Guid entityId	Id of the entity record.
string targetColumnName	Name of the predicted field.
Dictionary inputColumnPathMap	A set of correspondences between the columns of the entity (or paths to linked columns) and fields in the model's metadata. Example: *new Dictionary { { "Symptoms", "Symptoms" }, { "CreatedOn", "CreatedOn" }, { "Account.Industry", "IndustryId" } };*
Func<IEnumerable<ClassificationResult>, ClassificationResult> valueSelectorFunc	This parameter is optional. A delegate connected to a method that enables you to specify which value from the predicted list will be written to the predicted field. By default, only the value for which the *Significance* property is set to "High" will be recorded in the field.

Creating data queries for the machine learning model



Introduction

Use the *Terrasoft.Core.DB.Select* class instance for queries of training data or data for predicting machine learning service (see "**Machine learning service**" and "**How to implement custom prediction model**"). It is dynamically imported by the *Terrasoft.Configuration.ML.QueryInterpreter*.

> ATTENTION
>
> The *QueryInterpreter* interpreter does not allow the use of lambda expressions.

Use the provided *userConnection* variable as an argument of the *Terrasoft.Core.UserConnection* type in the *Select* constructor when building query expression. The column with the "Id" alias (the unique id of the target object instance) is a required in the query expression.

The *Select* expression can be complex. Use the following practices to simplify it:

- Dynamic adding of types for the interpreter.
- Using local variables.
- Using the *Terrasoft.Configuration.QueryExtensions* utility class.

Dynamic adding of types for the interpreter

You can dynamically add types for the interpreter. For this the *QueryInterpreter* class provides the *RegsiterConfigurationType* and *RegisterType* methods. You can use them directly in the expression. For example, instead of direct using the type id:

```
new Select(userConnection)
        .Column("Id")
        .Column("Body")
```

```
    .From("Activity")
    .Where("TypeId").IsEqual(Column.Const("E2831DEC-CFC0-DF11-B00F-001D60E938C6"));
```

you can use the name of a constant from dynamically registered enumeration:

```
RegisterConfigurationType("ActivityConsts");
new Select(userConnection)
        .Column("Id")
        .Column("Body")
    .From("Activity")
    .Where("TypeId").IsEqual(Column.Const(ActivityConsts.EmailTypeUId));
```

Using local variables

You can use local variables to avoid code duplication and more convenient structuring. Constraint: the type of the variable must be statically calculated and defined by the *var* word.

For example, the query with repetitive use of delegates:

```
new Select(userConnection)
        .Column("Id")
        .Column("Body")
    .From("Activity")
    .Where("CreatedOn").IsGreater(Func.DateAddMonth(-1, Func.CurrentDateTime()))
    .And("StartDate").IsGreater(Func.DateAddMonth(-1, Func.CurrentDateTime()));
```

you can write in a following way:

```
var monthAgo = Func.DateAddMonth(-1, Func.CurrentDateTime());

new Select(userConnection)
        .Column("Id")
        .Column("Body")
    .From("Activity")
    .Where("StartDate").IsGreater(monthAgo)
    .And("ModifiedOn").IsGreater(monthAgo);
```

Using the Terrasoft.Configuration.QueryExtensions utility class

The *Terrasoft.Configuration.QueryExtensions* utility class provides several extending methods for the *Terrasoft.Core.DB.Select*. This enables to build more compact queries.

As the *object sourceColumn* argument you can use following types (they will be transformed to the Terrasoft.Core.DB.QueryColumnExpression) for all extending methods:

- *System.String* – the name of the column in the "TableAlias.ColumnName as ColumnAlias" format (where the TableAlias and ColumnAlias are optional) or "*" – all columns.
- *Terrasoft.Core.DB.QueryColumnExpression* – will be added without changes.
- *Terrasoft.Core.DB.IQueryColumnExpressionConvertible* – will be converted.
- *Terrasoft.Core.DB.Select* – will be considered as subquery.

ATTENTION

An exception will be thrown if the type is not supported.

Terrasoft.Configuration.QueryExtensions use cases

1. The public static Select Cols(this Select select, params object[] sourceColumns) method

Adds specified columns or subexpressions to the query.

Using the Cols() extension method, instead of the following expression:

```
new Select(userConnection)
        .Column("L", "Id")
        .Column("L", "QualifyStatusId")
        .Column("L", "LeadTypeId")
        .Column("L", "LeadSourceId")
        .Column("L", "LeadMediumId").As("LeadChannel")
        .Column("L", "BusinesPhone").As("KnownBusinessPhone")
    .From("Lead").As("L");
```

you can write:

```
new Select(userConnection).Cols(
        "L.Id",
        "L.QualifyStatusId",
        "L.LeadTypeId",
        "L.LeadSourceId",
        "L.LeadMediumId AS LeadChannel",
        "L.BusinesPhone AS KnownBusinessPhone")
    .From("Lead").As("L");
```

2. The public static Select Count(this Select select, object sourceColumn) method

Adds an aggregation column to calculate the number of non-empty values to the query.

For example, instead:

```
var activitiesCount = new Select(userConnection)
    .Column(Func.Count(Column.Asterisk()))
    .From("Activity")
```

you can write:

```
var activitiesCount = new Select(userConnection)
    .Count("*") // You can also specify the column name.
    .From("Activity")
```

3. The public static Select Coalesce(this Select select, params object[] sourceColumns) method

Adds a column with the function of determining the first value not equal to NULL to the query.

For example, instead:

```
new Select(userConnection)
    .Cols("L.Id")
    .Column(Func.Coalesce(
            Column.SourceColumn("L", "CountryId"),
            Column.SourceColumn("L", "CountryId"),
            Column.SourceColumn("L", "CountryId")))
        .As("CountryId")
    .From("Lead").As("L")
        .LeftOuterJoin("Contact").As("C").On("L", "QualifiedContactId").IsEqual("C",
"Id")
         .LeftOuterJoin("Account").As("A").On("L", "QualifiedAccountId").IsEqual("A",
"Id");
```

you can write:

```
new Select(userConnection)
        .Cols("L.Id")
        .Coalesce("L.CountryId", "C.CountryId", "A.CountryId").As("CountryId")
    .From("Lead").As("L")
```

```
        .LeftOuterJoin("Contact").As("C").On("L", "QualifiedContactId").IsEqual("C",
"Id")
        .LeftOuterJoin("Account").As("A").On("L", "QualifiedAccountId").IsEqual("A",
"Id");
```

4. The public static Select DateDiff(this Select select, DateDiffQueryFunctionInterval interval, object startDateExpression, object endDateExpression) method

Adds a column that specifies the date difference to the query.

For example, instead:

```
new Select(_userConnection)
        .Cols("Id")
        .Column(Func.DateDiff(DateDiffQueryFunctionInterval.Day,
            Column.SourceColumn("L", "CreatedOn"),
Func.CurrentDateTime())).As("LeadAge")
    .From("Lead").As("L");
```

you can write:

```
var day = DateDiffQueryFunctionInterval.Day;
new Select(userConnection)
        .Cols("L.Id")
        .DateDiff(day, "L.CreatedOn", Func.CurrentDateTime()).As("LeadAge")
    .From("Lead").As("L");
```

5. public static Select IsNull(this Select select, object checkExpression, object replacementValue)

Adds a column with the function replacing NULL value with a replacement expression.

For example, instead:

```
new Select(userConnection).Cols("Id")
        .Column(Func.IsNull(
                Column.SourceColumn("L", "CreatedOn"),
                Column.SourceColumn("L", "ModifiedOn")))
    .From("Lead").As("L");
```

you can write:

```
new Select(userConnection).Cols("L.Id")
        .IsNull("L.CreatedOn", "L.ModifiedOn")
    .From("Lead").As("L");
```

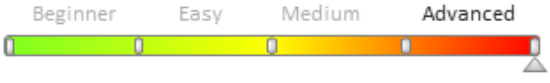
Bpm'online lending

Contents

- **Terrasoft.Configuration.EntityMapper class**

Terrasoft.Configuration.EntityMapper class

Difficulty level



Introduction

The *Terrasoft.Configuration.EntityMapper* class is a utility configuration class that stored in the [FinAppLending] package of the Lending product. *EntittyMapper* allows to map data of one *Entity* with another using rules defined in the configuration file. Using the approach of mapping the data of different entities avoids the appearance of a monotonous code.

The idea of mapping the data of different entities is implemented in the following classes:

- *EntityMapper* – implements the mapping logic.
- *EntityResult* – defines the resulting type of the mapped entity.
- *MapConfig* – a set of mapping rules.
- *DetailMapConfig* – used to set up a list of mapping rules of the details and entities connected with them.
- *RelationEntityMapConfig* – contains rules for mapping connected entities.
- *EntityFilterMap* – a filter for database query.

Terrasoft.Configuration.EntityMapper

Table 1. Methods of the Terrasoft.Configuration.EntityMapper class

Name	Parameters	Returned value	Description
public virtual EntityResult GetMappedEntity(Guid recId, MapConfig config)	*recId* – GUID recodrs in the database. *config* – an instance of the *MapConfig* class, which is a set of mapping rules.	An instance of the *EntityResult* class, which is a mapped data for two Entity objects.	Returns mapped data for two Emtity objects.
public virtual Dictionary<string, object> GetColumnsValues(Guid recordId, MapConfig config, Dictionary<string, object> result)	*recordId* – GUID recodrs in the database. *Config* – an instance of the *MapConfig* class, which is a set of mapping rules. *Result* – a dictionary of columns and their values of the mapped entity.	A dictionary of columns and their values.	Gets the main entity from the database and matches its columns and values according to the rules specified in the config object.
public virtual Dictionary<string, object> GetRelationEntityColumnsValues(List<RelationEntityMapConfig> relations, Dictionary<string, object> dictionaryToMerge, string columnName, Terrasoft.Nui.ServiceModel.DataContract.LookupColumnValue entitylookup)	*relations* – a list of rules for obtaining related records. *dictionaryToMerge* – a dictionary with columns and theis values. *columnName* – name of the parent column *entitylookup* – an object that contains name and Id of the record in the database.	A dictionary of columns and their values.	Gets the related entities from the database and matches them to the main entities.
protected virtual EntitySchemaQuery SetColumns(EntitySchemaQuery esq, Dictionary<string, string> columns)	*esq* – instance of the EntitySchemaQuery class *Columns* – a dictionary of names of the mapped	The instance of the EntitySchemaQuery class.	Sets columns for selection from the database.

	columns.		
protected EntitySchemaQuery SetFilters(EntitySchemaQuery esq, List<EntityFilterMap> filters)	*esq* – instance of the EntitySchemaQuery class *filters* – a list of the filters.	The instance of the EntitySchemaQuery class.	Sets filters for the entities to select records from the database.
protected virtual Dictionary<string, List<Dictionary<string, object>>> GetDetailsColumnsValues(Guid recId, MapConfig config, Dictionary<string, List<Dictionary<string, object>>> result)	*recId* – GUID recodrs in the database. *Config* – an instance of the *MapConfig* class, which is a set of mapping rules. *Result* – a dictionary of columns and their values of the mapped entity.	A dictionary of details, detail columns and column values.	Gets the entity from the database and matches its columns and values according to the rules specified in the config object.

Terrasoft.Configuration.EntityResult

Used as a container for returning mapped values.

Table 2. Main properties of the Terrasoft.Configuration.EntityResult class

Property	Type	Description
Columns:	*Dictionary<string, object>*	A dictionary of main entity column names and their values.
Details	*Dictionary<string, List<Dictionary<string, object>>>*	A dictionary of the detail names with the list of their columns and values.

Terrasoft.Configuration.MapConfig

Used to set a list of mapping rules

Table 3. Main properties of the Terrasoft.Configuration.MapConfig class

Property	Type	Description
SourceEntityName	*string*	Entity name in the database.
Columns:	*Dictionary<string, object>*	A dictionary with the names of columns of one entity and compared columns of another entity.
DetailsConfig	*List<DetailMapConfig>*	A list of configuration objects with rules for details.
CleanDetails	*List<string>*	A list of detail names for cleaning their values.
RelationEntities	*List<RelationEntityMapConfig>*	List of configuration objects with rules for mapping related records with the main entity.

Terrasoft.Configuration.DetailMapConfig

Used to set up a list of mapping rules of the details and entities connected with them.

Table 4. Main properties of the Terrasoft.Configuration.DetailMapConfig class

Property	Type	Description
DetailName	*string*	Detail name (Tt ensure the uniqueness of detail instance).
SourceEntityName	*string*	Entity name in the database.
Columns:	*Dictionary<string, object>*	A dictionary with the names of columns of one entity and compared columns of another entity.

Filters	*List<EntityFilterMap>*	A list of configuration objects with filtration rules for more accurate selections from the database.
RelationEntities	*List<RelationEntityMapConfig>*	List of configuration objects with rules for mapping related records with the main entity.

Terrasoft.Configuration.RelationEntityMapConfig

Contains rules for mapping connected entities.

Table 5. Main properties of the Terrasoft.Configuration.RelationEntityMapConfig class

Property	Type	Description
ParentColumnName	*string*	The name of the parent column, which, when found, will trigger the logic for obtaining and mapping the entity data.
SourceEntityName	*string*	Entity name in the database.
Columns:	*Dictionary<string, object>*	A dictionary with the names of columns of one entity and compared columns of another entity.
Filters	*List<EntityFilterMap>*	A list of configuration objects with filtration rules to refine selections from the database.
RelationEntities	*List<RelationEntityMapConfig>*	List of configuration objects with rules for mapping related records with the main entity.

Terrasoft.Configuration.EntityFilterMap

A filter for database query.

Table 5. Main properties of the Terrasoft.Configuration.EntityFilterMap class

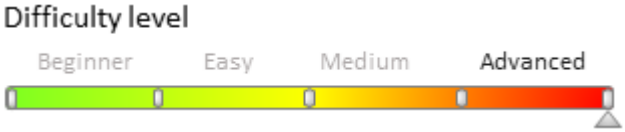
Property	Type	Description
ColumnName	*string*	The name of the column, which when found, will start the filtering logic.
Value	*object*	The value to compare to.

Bpm'online marketing

Contents

- **Campaign elements**

Campaign elements



Introduction

Marketing campaign diagrams are created in a visual campaign designer in the [Campaigns] section. The campaign diagram consists of campaign elements and transitions (flows).

Once the campaign is launched, the flow-schema of the campaign is created. The campaign elements are converted to a campaign execution chain and the start time is calculated for each element. The flow-schema can be significantly different from the visual campaign diagram in the designer.

Campaign elements can be synchronous and asynchronous.

Synchronous elements are executed according to the order specified in the flow-schema. The transition to the subsequent elements is performed once the synchronous element is executed. The execution flow is blocked and

waits for the operation to complete.

Asynchronous elements wait for the finished execution of certain external systems, resources, asynchronous services, or user reactions (e.g., clicking a link in an email).

Their position in the flow-schema is determined by their element type. The [Add from folder] and [Exit according to folder conditions] elements are executed first. These elements are used to add or remove participants from the campaign audience. Campaign participants are moving from one element to the other through the flows. If the flow has certain configured conditions, the system filters the participants based on these conditions and determines the execution time of the subsequent element.

The mechanism for planning the next campaign launch

The following is the algorithm for for planning the next campaign launch:

1. The time of the next launch of an element is determined by the configured delay.

   - The “In a day” option is selected. The date and time of the next execution of this element is calculated with the help of the following formula:

   Date and time of execution = current date and time + N minutes / hour,

   where *N* is the value of the [Number of days] field, populated by the user.

   - The “Few days” option is selected. The next execution of this element is performed with the help of the following formula:

   Date = [current date+ N days],

   where *N* is the value of the [Number of days] field, populated by the user.

   Execution time = time specified by the user.

   - The “No, execute after the previous one” option is selected. The next execution of this element is performed at the time of the next launch of the campaign.

2. According to the variant described in paragraph 1, the launch time for each element of the campaign scheme is calculated.

3. Upon comparing all values, the closest launch time selected and set as the campaign launch time.

4. Forming a list of elements, which will be executed upon next launch. The list contains all elements, the launch time of which is the same as the campaign launch time.

Main campaign element classes

JavaScript classes

The base element schema class is *ProcessFlowElementSchema*. The *CampaignBaseCommunicationSchema* is the parent class for all elements in the [Communications] group. The *CampaignBaseAudienceSchema* is the parent class for the [Audience] group of elements.

When creating an element in a new group of elements, it is recommended to implement the base schema of the element first, and then inherit each element from it.

Each schema corresponds to the schema of the element properties edit page. The base edit page schema is *BaseCampaignSchemaElementPage*. Each new element page extends the base page.

The *CampaignSchemaManager* class manages the schemas of elements available in the system. It inherits the main functionality of the *BaseSchemaManager* class.

C# classes

Simple element classes

CampaignSchemaElement – base class. All other elements are inherited from this class.

SequenceFlowElement – base class for the [Sequence flow] element.

ConditionSequenceFlowElement – base class for the [Condition flow] element.

EmailConditionalTransitionElement – transition element class by response.

AddCampaignParticipantElement – add audience (participants) element class.

ExitFromCampaignElement – the class of the audience exit element.

MarketingEmailElement – the class of the Email element.

Executable element classes

CampaignProcessFlowElement – base class. All other executable elements are inherited from this class.

AddCampaignAudienceElement – audience element class.

ExcludeCampaignAudienceElement – the class of the audience exit element.

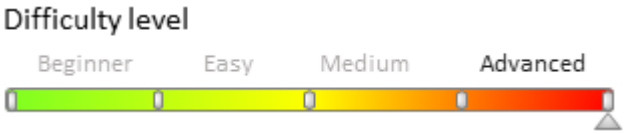
BulkEmailCampaignElement – the class of the Email element.

Bpm'online service

Contents

- **PortalMessagePublisherExtensions mixin. Portal messages in SectionActionDashboard**

PortalMessagePublisherExtensions mixin. Portal messages in SectionActionDashboard



Introduction

A mixin is a class designed to extend the functions of other classes. Mixins are separately created classes with additional functionality. Learn more about mixins in the “**Mixins. The "mixins" property** article.

The *PortalMessagePublisherExtensions* mixin is used for the extension of the *SectionActionDashboard* schema (and its derived schemas). It allows you to extend the configuration of the *SectionActionDashboard* tabs with the *PortalMessageTab* portal message tab and add the corresponding *Portal* message portal. The mixin is implemented in the *PortalMessagePublisher* package and is available in the *ServiceEnterprise* product (or in the bundles that include this product).

Methods

Name	Description
extendTabsConfig(config) : Object	Extends the configuration of the *SectionActionDashboard* tabs with the *PortalMessageTab* portal messages tab. Returns the augmented object (*Object*) of the *SectionActionDashboard* tab configuration. The *config* parameter (*Object*) – *SectionActionDashboard* tab configuration object.
extendSectionPublishers(publishers) : Array	Adds a portal channel (*Portal*) to the message publisher collection. Returns the augmented collection of message publishers (*Array*). The *publishers (Array)* parameter is the collection of message publishers.

Use case

```
define("CaseSectionActionsDashboard", ["PortalMessagePublisherExtensions"],
function() {
    return {
        mixins: {
            /**
             * @class PortalMessagePublisherExtensions extends tabs and publishers
configs.
             */
            PortalMessagePublisherExtensions:
"Terrasoft.PortalMessagePublisherExtensions"
        },
        methods: {
            /**
             * @inheritdoc Terrasoft.SectionActionsDashboard#getExtendedConfig
             * @overridden
             */
            getExtendedConfig: function() {
                // Getting the tab configuration object from the parent method.
                var config = this.callParent(arguments);
                // Calling the mixin method, adding a portal tab configuration.

this.mixins.PortalMessagePublisherExtensions.extendTabsConfig.call(this, config)
                // Returns the extended configuration object.
                return config;
            },

            /**
             * @inheritdoc Terrasoft.SectionActionsDashboard#getSectionPublishers
             * @overridden
             */
            getSectionPublishers: function() {
                // Getting a collection of message publishers from the parent method.
                var publishers = this.callParent(arguments);
                // Calling the mixin method, adding a portal channel.

this.mixins.PortalMessagePublisherExtensions.extendSectionPublishers.call(this,
publishers);
                // Returns the extended collection of message publishers.
                return publishers;
            }
    },
        diff: /**SCHEMA_DIFF*/[
            {
                "operation": "insert",
                "name": "PortalMessageTab",
                "parentName": "Tabs",
                "propertyName": "tabs",
                "values": {
                    "items": []
                }
            },
            {
                "operation": "insert",
                "name": "PortalMessageTabContainer",
                "parentName": "PortalMessageTab",
                "propertyName": "items",
                "values": {
                    "itemType": this.Terrasoft.ViewItemType.CONTAINER,
                    "classes": {
```

```
                        "wrapClassName": ["portal-message-content"]
                    },
                    "items": []
                }
            },
            {
                "operation": "insert",
                "name": "PortalMessageModule",
                "parentName": "PortalMessageTab",
                "propertyName": "items",
                "values": {
                    "classes": {
                        "wrapClassName": ["portal-message-module", "message-module"]
                    },
                    "itemType": this.Terrasoft.ViewItemType.MODULE,
                    "moduleName": "PortalMessagePublisherModule",
                    "afterrender": {
                        "bindTo": "onMessageModuleRendered"
                    },
                    "afterrerender": {
                        "bindTo": "onMessageModuleRendered"
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

DataManager class description and use cases

Difficulty level

Introduction

Sometimes It may be necessary to create, modify and delete entity data without saving these changes to the database in the process of working with the client part of the application. Saving changes to the database must take place when the save method is explicitly called. These functions are implemented in the *DataManager* and *DataManagerItem* classes.

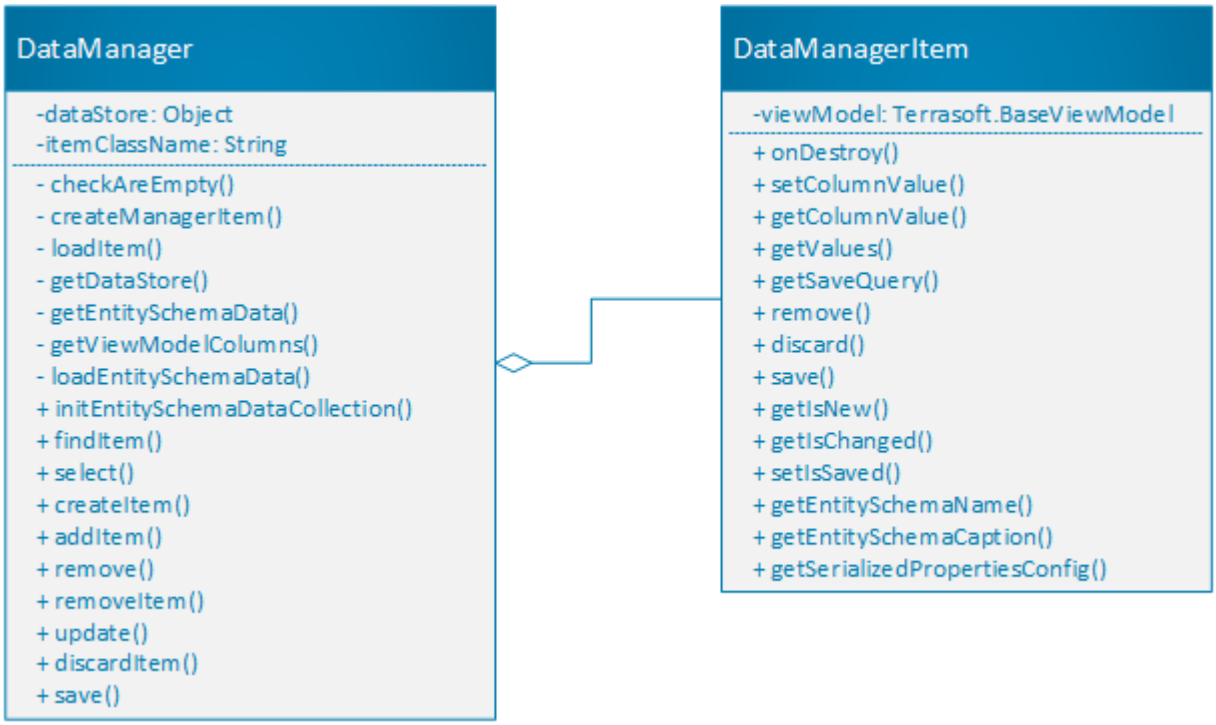
The *DataManager* class is a singleton available through the *Terrasoft* global object. This class provides the *dataStore* repository. The contents of one or more database tables can be loaded into the repository. Example:

```
dataStore: {
SysModule: sysModuleCollection,
SysModuleEntity: sysModuleEntityCollection
}
```

sysModuleCollection and *sysModuleEntityCollection* are the data collections of the *DataManagerItem* type of the *SysModule* and *SysModuleEntity* schemas. Each collection record is a record of the corresponding database table.

DataManager and *DataManagerItem* class diagram is available on Fig. 1.

Fig. 1 Class diagram



Base properties and methods

Base properties and methods of the *DataManager* class are available in Table 1 and Table 2. " article.

Table 1. The *DataManager* class properties

Name	Type	Description
dataStore	*Object*	The data collection repository.
itemClassName	*String*	Name of the record class. Has the "Terrasoft.DataManagerItem" value.

Table 2. Main methods of the *DataManager* class

Name	Parameters	Description
select	*config {Object}* – configuration object; *callback {Function}* – callback function; *scope {Object}* – the callback function context.	If there is no data with the *config.entitySchemaName* name in the *dataStore*, then the method forms and executes the request to the database and returns the received data, or the method will return the data collection from the *dataStore*.
createItem	*config {Object} – configuration object;* *callback {Function}* – callback function; *scope {Object}* – the callback function context.	Creates a new record of the *config.entitySchemaName* type with the *config.columnValues* colomn values.
addItem	*item {Terrasoft.DataManagerItem}* – added record.	Adds the *item* record to the schema data collection.
findItem	*entitySchemaName {String}* – data collection name; *id {String}* — record Id.	Returns the record of the schema data collection with the *entitySchemaName* name and *id* Id.

remove	*item {Terrasoft.DataManagerItem}* – deleted record.	Sets the *isDeleted* flag for the *item* record. The record will be deleted from the database after saving the changes.
removeItem	*item {Terrasoft.DataManagerItem}* – deleted record.	Deletes the record from the schema data collection.
update	*config {Object}* – *configuration object;* *callback {Function}* – callback function; *scope {Object}* – the callback function context.	Updates the record with the *config.primaryColumnValue* primary column value by the *config.columnValues* values.
discardItem	*item {Terrasoft.DataManagerItem}* – a record with the canceled changes.	Cancels changes for the *item* record made in current working session with the *DataManger* object.
save	*config {Object}* – *configuration object;* *callback {Function}* – callback function; *scope {Object}* – the callback function context.	Saves the schema data collections specified in the *config.entitySchemaNames* to the database.

Base properties and methods of the *DataManagerItem* class are available in Table 3 and Table 4. " article.

Table 3. The *DataManagerItem* class properties

Name	Type	Description
viewModel	*Terrasoft.BaseViewMode*	Object projection of the record in the database.

Table 4. Main methods of the *DataManagerItem* class

Name	Parameters	Description
setColumnValue	*columnName {String}* – column name; *columnValue {String}* – column value.	Sets the new *columnValue* value for the *columnName* column.
getColumnValue	*columnName {String}* – column name;	Returns the value of the *columnName* column.
getValues	No.	Returns values of all record columns.
remove	No.	Sets *isDeleted* flag to the record.
discard	No.	Cancels changes for the record made in current working session with the *DataManger* object.
save	No.	Saves changes in the database.
getIsNew	No.	Returns the flag that the record is new.
getIsChanged	No.	Returns the flag that the record was modified.

Examples

Getting records from the [Contact] table:

```
// Definition of the configuration object.
var config = {
    //Entity Schema Name.
    entitySchemaName: "Contact",
    // Remove duplicates in the resulting dataset.
    isDistinct: true
```

```
};
// Receiving data.
Terrasoft.DataManager.select(config, function (collection) {
    // Saving received records to local storage.
    collection.each(function (item) {
        Terrasoft.DataManager.addItem(item);
    });
}, this);
```

Adding new record to the *DataManager* object:

```
// Definition of the configuration object.
var config = {
    // Entity Schema Name.
    entitySchemaName: "Contact",
    // Column values.
    columnValues: {
        Id: "00000000-0000-0000-0000-000000000001",
        Name: "Name1"
    }
};
// Creating a new record.
Terrasoft.DataManager.createItem(config, function (item) {
    Terrasoft.DataManager.addItem(item);
}, this);
```

Getting the record and changing the column value:

```
// Getting a record.
var item = Terrasoft.DataManager.findItem("Contact",
     "00000000-0000-0000-0000-000000000001");
// Setting a new value for "Name2" to the [Name] column.
item.setColumnValue("Name", "Name2");
```

Deleting the record from the *DataManager* object:

```
// Definition of the configuration object.
var config = {
    // Entity Schema Name.
    entitySchemaName: "Contact",
    // Primary column value.
    primaryColumnValue: "00000000-0000-0000-0000-000000000001"
};
// Sets the isDeleted attribute for item.
Terrasoft.DataManager.remove(config, function () {
}, this);
```

Cancels changes made in current working session with the *DataManger* object.

```
// Getting a record.
var item = Terrasoft.DataManager.findItem("Contact",
     "00000000-0000-0000-0000-000000000001");
// Undo changes for writing.
Terrasoft.DataManager.discardItem(item);
```

Saves changes in the database.

```
// Definition of the configuration object.
var config = {
    // Entity Schema Name.
    entitySchemaNames: ["Contact"]
};
// Saving changes to the database.
```

```
Terrasoft.DataManager.save(config, function () {
}, this);
```

Feature Toggle. Mechanism of enabling and disabling functions

Difficulty level

Introduction

Feature toggle is a software development technique that provides support for connecting additional functionality in a running application. This allows to use continuous integration, keep the application working and hide the functionality that is under development process.

The main idea is that there is a block of additional functionality (often not fully implemented) in the source code and conditional operator that defines if the functionality connected.

Mechanism of enabling and disabling functions

The FeaturesPage page is used to add, enable and disable functions. The page address is:

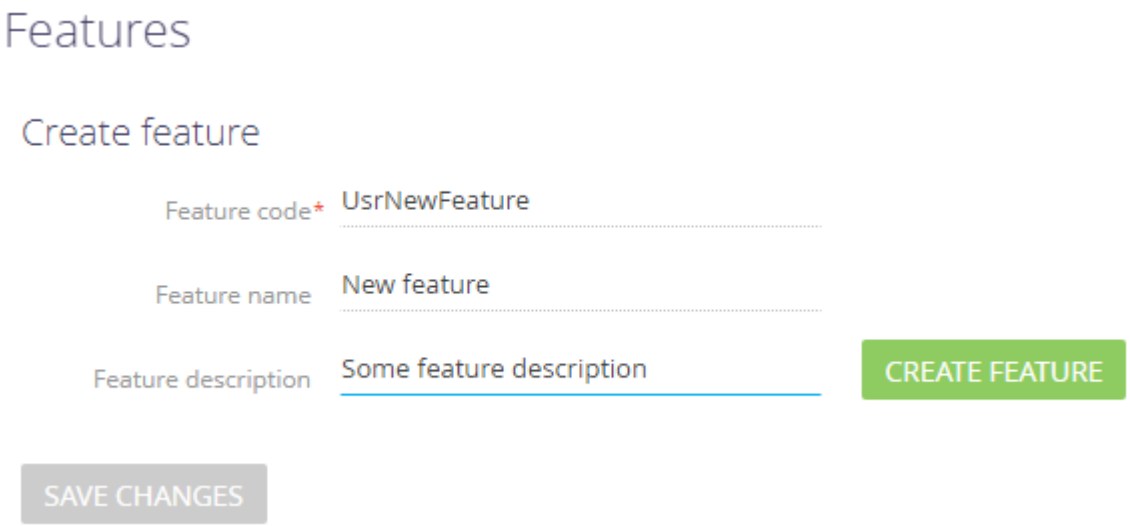
```
[Application address]/0/Nui/ViewModule.aspx#BaseSchemaModuleV2/FeaturesPage
```

Example:

```
http://mybpmonline.com/0/Nui/ViewModule.aspx#BaseSchemaModuleV2/FeaturesPage
```

To add new functions specify its code, name and description and click the [Create feature] button (Fig. 1).

Fig. 1. Interface of adding new feature



Use corresponding checkbox to enable or disable new features (Fig. 2.1). To apply changes click the [Save changes] button (Fig. 2.2).

Fig. 2. Enable/disable feature

Storing the functionality datain the database

A list of functionality available for enabling/disabling is stored in the *Feature* table of the application database, Table is empty by default. Main *Feature* table fields are given in table 1.

Table 1. Main *Feature* table fields

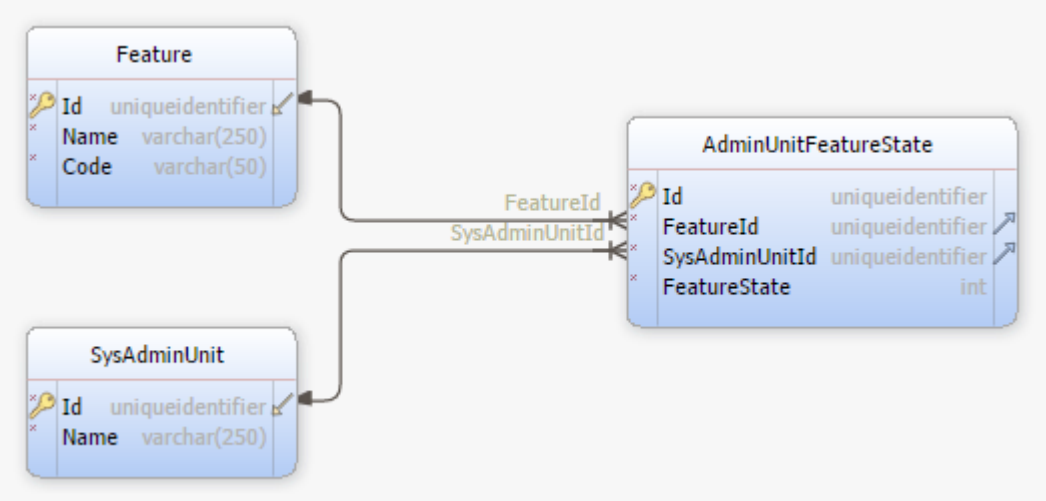
Name	Type	Description
Id	*uniqueidentifier*	Unique Id of the record
Name	*varchar(250)*	Functionality name.
Code	*varchar(50)*	Functionality code.

Information about functionality state (enabled/disabled) stored in the *FeatureState* field of the *AdminUnitFeatureState* table (Fig.1). The *AdminUnitFeatureState* table binds the *Feature* and *SysAdminUnit* tables where users and system user groups are defined. Main *AdminUnitFeatureState* table fields are given in table 2.

Table 2. Main *AdminUnitFeatureState* table fields

Name	Type	Description
Id	*uniqueidentifier*	Unique Id of the record
FeatureId	*uniqueidentifier*	Unique Id of the functionality record.
SysAdminUnitId	*uniqueidentifier*	Unique Id of the user record.
FeatureState	*int*	Functionality state. 1 – enabled, 0 – disabled.

Fig. 1 Diagram of table relationships



Defining the new functionality in the source code.

To implement the new functionality to the source code it should be defined in the block of the conditional operator

that will check the state of the functionality connection (FeatureState).

Client side JavaScript

A conditional template for defining additional functionality in the source code:

```
// The method defining the additional functionality.
someMethod: function() {
    // Functionality connection check.
    if (Terrasoft.Features.getIsEnabled("functionality code")) {
        // Implementation of additional functionality.
        ...
    }
    // Method Implementation
    ...
}
```

The *getIsFeatureEnabled* method is implemented in the *BaseSchemaViewModel* base schema view model. Therefore, the *Terrasoft.Features.getIsEnabled* method can be replaced with *this.getIsFeatureEnabled("functionality code")*.

Refresh the browser page after connecting the new functionality to enable it in the client code and load it in the browser.

Server side C#

A set of extending methods of the *UserConnection* class was implemented to use the *Feature toggle* in the source code schemas on the server side in the *Terrasoft.Configuration.FeatureUtilities* class. A list of the extended methods is given in the Table 3. The *FeatureState* functionality states are enumerated in the same class.

Table 2. Main methods of the *DataManager* class

Methods.	Parameters	Description
int GetFeatureState(this UserConnection source, string code)	*code* – functionality code.	Returns functionality state.
Dictionary <string, int> GetFeatureStates(this UserConnection source)	No.	Returns the state of all functionality.
void SetFeatureState(this UserConnection userConnection, string code, int state, bool forAllUsers = false)	*code* – functionality code; *state* – functionality state (0/1); *forAllUsers* – a flag of enabling the functionality for all users.	Returns functionality state.
void CreateFeature(this UserConnection source, string code, string name, string description)	*code* – functionality code; *name* – functionality name; *Description* – functionality description.	Creates new functionality.
bool GetIsFeatureEnabled(this UserConnection source, string code)	*code* – functionality code.	Checks if the functionality connected.

A conditional template for defining additional functionality in the source code:

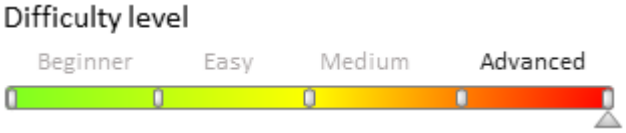
...

```
// A namespace in which the ability to switch additional
// functionality is defined.
using Terrasoft.Configuration;
…
// The method in which additional functionality will be defined.
public void AnyMethod() {
    // Check if functionality is enabled.
    if (UserConnection.GetIsFeatureEnabled("functionality code")) {
        // Implementation of additional functionality.
    }
    // Method implementation.
    ...
}
```

Setting the value of functionality state is executed by call of the SetFeatureState method:

```
UserConnection.SetFeatureState("functionality code", FeatureState);
```

The MoneyUtilsMixin mixin



Introduction

The *MoneyUtilsMixin* mixin contains the general logic of cash transactions.

Methods

The getCurrencyDivision method

The *getCurrencyDivision* method is used to get the denomination (multiplicity) of a currency. Returns the denomination (multiplicity) of the currency. The type of returned value is "*Number*".

> ⚠ NOTE
>
> The denomination (multiplicity) of the currency is the amount of currency for which the calculation of the exchange rate will be made. For example: x1, x10, x100, etc.

Method format: *this.getCurrencyDivision([config]);*

Possible properties of the configuration object that are passed as a parameter are listed in Table 1.

Table 1. 1. Properties of the parameter object of the *getCurrencyDivision* method

Name	Type	Description	Default values
config	*Object*	Object with additional parameter properties.	
config.currencyAttribute (optional)	*String*	The name of the view model attribute containing the currency object.	Currency
config.currencyDivisionProp (optional)	*Number*	The name of the currency object property containing the denomination (multiplicity) of the currency. If this property is specified it will be returned.	Division
config.currencyDivision	*Number*	The value of the denomination (multiplicity)	

(optional)	of the currency. If this property is specified it will be returned.

The recalculatePrimaryValue method

The *recalculatePrimaryValue* method calculates the value of the specified attribute in the base currency.

> NOTE
>
> "Base" currency is the currency that defines exchange rate for all other currencies. Base currency is defined in the [Base currency] system setting.

Method format: *this.recalculatePrimaryValue(attribute [, config]);*

Possible method parameters are listed in table 2.

Table 2. Parameters of the *recalculatePrimaryValue* method

Name	Type	Description	Default values
attribute	*String*	The name of the view model attribute, for which the value in the base currency must be recalculated.	
config (optional)	*Object*	Object with additional parameter properties. May include parameters for the *getCurrencyDivision* method.	
config.modelInstance (optional)	*Terrasoft.BaseModel*	A view model for which the recalculation will be performed.	this
config.primaryValueAttribute (optional)	*String*	The name of the attribute containing the value in the base currency.	"Primary" + attribute
config.currencyRateAttribute (optional)	*String*	The name of the attribute containing the currency exchange rate value.	CurrencyRate

The recalculateValue method

The *recalculateValue* method calculates the value of the specified attribute according to the base currency.

Method format: *this.recalculateValue(attribute [, config]);*

Possible method parameters are listed in table 3.

Table 3. Parameters of the *recalculateValue* method

Name	Type	Description
attribute	*String*	The name of the attribute, the value in the base currency must be recalculated for this attribute.
config (optional)	*Object*	Object with additional parameters. It can include parameters for the *recalculatePrimaryValue* method with parameters for the *getCurrencyDivision* method.

The getPercentage method

The *getPercentage* method calculates which percentage does a part of a total number make. Returns the percentage. The type of returned value is "*Number*".

Method format: *this.getPercentage(amount, part);*

Possible method parameters are listed in table 4.

Table 4. Parameters of the *getPercentage* method

Name	Type	Description
amount	*Number*	Total number.
part	*Number*	Part of the total for which the percentage value must be calculated.

Use cases

```
// Returns 20.
this.getPercentage(10, 2);
// Returns 100.
this.getPercentage(10, 10);
// Returns 0.0001.
this.getPercentage(100, 0.0001);
```

The getPercentagePart method

The *getPercentagePart* method calculates which number ("part") makes the specified percentage from a total number. Returns the part of a number. The type of returned value is "*Number*".

Method format: *this.getPercentagePart(amount, percent);*

Possible method parameters are listed in table 5.

Table 5. Parameters of the *getPercentagePart* method

Name	Type	Description
amount	*Number*	Total number.
percent	*Number*	Percentage.

Use cases

```
 // Returns 2.
this.getPercentagePart(10, 0.2);
```

The getIncludedPercentagePart method

The *getIncludedPercentagePart* method divides the total number into two parts. One of the parts is calculated as the percentage of the second part. Returns the part that was calculated as the percentage. The type of returned value is "*Number*".

Method format: *this.getIncludedPercentagePart(amount, percent);*

Possible method parameters are listed in table 6.

Table 6. Parameters of the getIncludedPercentagePart method

Name	Type	Description
amount	*Number*	Total number.
percent	*Number*	Percentage.

Use cases

```
// Returns 1, because 1 is a 10% of 10, 10 + 1 = 11.
this.getIncludedPercentagePart(11, 10);
```

The roundMoney method

The *roundMoney* method rounds the value with the precision specified in the "Terrasoft.data.constants.MONEY_PRECISION". Banking rounding is used. The type of returned value is "*Number*".

Method format: *this.roundMoney(amount);*

Possible method parameters are listed in table 7.

Table 7. Parameters of the roundMoney method

Name	Type	Description
amount	*Number*	Total number.

Use cases

```
// If Terrasoft.data.constants.MONEY_PRECISION = 4 (by default):
// Returns 0.1235.
this.roundMoney(0.123456789);
// Returns 0.1234.
this.roundMoney(0.123449);
```

The roundValue method

The *roundValue* method rounds the value with the precision specified in the configuration object or in the "Terrasoft.data.constants.MONEY_PRECISION". Banking rounding is used. The type of returned value is "*Number*".

Method format: *this.roundValue(amount [,config]);*

Possible method parameters are listed in table 8.

Table 8. Parameters of the roundValue method

Name	Type	Description
amount	*Number*	Total number.
config (optional)	*Object*	Object with additional parameters.
config.targetColumnName (optional)	*String*	The name of the view model column for which the calculation is made. The precision of this column will be used. If the column does not have the *precision* property, then the precision from the "Terrasoft.data.constants.MONEY_PRECISION" will be used.
config.decimalPlaces (optional)	*Number*	Precision (number of decimals) for rounding the value.

Use cases

```
// Returns 0.12.
this.roundValue(0.123456789, {decimalPlaces: 2});
// If the "SomeColumnName" column has a property
//  precision = 4, Returns 0.1235.
this.roundValue(0.123456789, {targetColumnName: "SomeColumnName"});
```

The getMoneyCalculator method

The *getMoneyCalculator* method returns the *DecimalUtils* object that is configured with the "Terrasoft.data.constants.MONEY_PRECISION" precision. The type of returned value is "*DecimalUtils*".

For more information about the *DecimalUtils* object, see the "**The DecimalUtils module**" article.

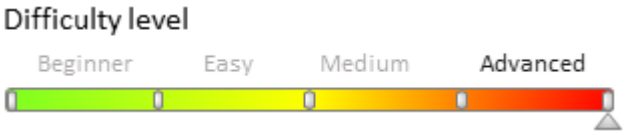
Method format: *this.getMoneyCalculator();*

Use cases

```
var calculator = this.getMoneyCalculator();
// Returns 3.
calculator.add(1, 2);
// Returns 0.3.
```

```
calculator.add(0.1, 0.2);
```

The DecimalUtils module



Introduction

Certain errors cay occur when using JavaScript to perform floating point calculations. For example:

```
var a = 0.1 + 0.2; // a = 0.30000000000000004.
var b = 0.3 - 0.1; // b = 0.19999999999999998.
```

The *DecimalUtils* module was created to avoid these errors.

It is designed to perform highly accurate mathematical operations and generates pseudo-random numbers. The *Terrasoft.DecimalUtils* class is defined in this module. It contains all methods used for mathematical operations.

Constructor

The *Terrasoft.DecimalUtils* class instance is created using the *Ext.create()* method of the global *Ext* object.

Constructor format *var decimalUtils = Ext.create("Terrasoft.DecimalUtils" [, config]);*

The optional *config* parameter is the configuration object of the constructor. Properties of the *config* object are described in table 1.

Table 1. Properties of the DecimalUtils constructor configuration object

Name	Type	Description	Default value
decimalPlaces (optional)	*Number*	The number of fractional part digits. Banker's rounding is applied to all calculations.	4
precision (optional)	*Number*	The number of significant figures for internal calculations. It includes both integer and fractional parts. For example, the number 123456789.123456789 contains 18 significant digits - 9 integer digits and 9 fractional digits.	24

Methods

The add() method

The *add* method calculates the sum of two numbers. Returning value type – *Number*.

Method format: *decimalUtils.add(a, b);*

Possible method parameters are listed in table 2.

Table 2. Add() method parameters

Name	Type	Description
a	*Number*	Addend
b	*Number*	Addend

Use case:

```
// Crearing an object.
```

```
var decimalUtils = Ext.create("Terrasoft.DecimalUtils");
// The add() method returns 0.3.
decimalUtils.add(0.1, 0.2);
```

The subtract() method

The *subtract()* method subtracts two numbers, i.e. calculates the difference. Returning value type – *Number*.

Method format: *decimalUtils.subtract(a, b);*

Possible method parameters are listed in table 3.

Table 3. Parameters of the subtract() method

Name	Type	Description
a	*Number*	Minuend
b	*Number*	Subtrahend

Use case:

```
// Crearing an object.
var decimalUtils = Ext.create("Terrasoft.DecimalUtils");
// The subtract() method returns 0.2.
decimalUtils.subtract(0.3, 0.1);
```

The multiply() method

The *multiply()* method calculates the multiplication of two numbers. Returning value type – *Number*.

Method format: *decimalUtils.multiply(a, b);*

Possible method parameters are listed in table 4.

Table 4. Parameters of the multiply() method

Name	Type	Description
a	*Number*	Multiplier
b	*Number*	Multiplier

Use case:

```
// Crearing an object.
var decimalUtils = Ext.create("Terrasoft.DecimalUtils");
// The multiply() method returns 0.03.
decimalUtils.multiply(0.3, 0.1);
```

The divide() method

The *divide()* method divides two numbers. Returning value type – *Number*.

Method format: *decimalUtils.divide(a, b);*

Possible method parameters are listed in table 5.

Table 5. Parameters of the divide() method

Name	Type	Description
a	*Number*	Dividend
b	*Number*	Divisor

Use case:

```
// Crearing an object.
var decimalUtils = Ext.create("Terrasoft.DecimalUtils");
```

```
// The divide() method returns 3.
decimalUtils.divide(0.3, 0.1);
```

The evaluate() method

The *evaluate()* method evaluates the result of the passed expression. Returning value type – *Number*.

Method format: *decimalUtils.evaluate(expression);*

Possible method parameters are listed in table 6.

Table 6. Parameters of the evaluate() method

Name	Type	Description
expression	*Object*	An object with mathematical operation properties (*add*, *subtract*, *multiply*, *divide*). Its values are arrays containing the *Number* value types or objects with subexpressions. It has a recursive structure.

Use case:

```
// Crearing an object.
var decimalUtils = Ext.create("Terrasoft.DecimalUtils");
// A configuration object is passed as an argument.
decimalUtils.evaluate({
    // Calculating the sum. The summand values are passed in the array.
    add: [ // 1 + 7 = 8.
        // The first summand is a nested expression.
        {
            // Calculating the subtract. The values are passed in the array.
            subtract: [ // 4 - 3 = 1.
                // The minuend is a nested expression. It contains the multiplication
operation.
               { multiply: [2, 2] }, // 2 * 2 = 4.
               3
            ]
        },
       // The second summand is a nested expression.
       {
           // Calculating the subtract. The values are passed in the array.
           subtract: [ // 5 - (-2) = 7.
                // The minuend is a nested expression. It contains the multiplication
operation.
              { divide: [10, 2] }, // 10 ÷ 2 = 5.
              -2
           ]
       }
    ]
});// Returns 8.
```

> **Note**
>
> An array of operation arguments can contain an arbitrary number of elements. The operation is applied to the elements from left to right.

An example of using an addition operation with four arguments:

```
var decimalUtils = Ext.create("Terrasoft.DecimalUtils");
decimalUtils.evaluate({
  add: [1, 2, 3, 4]
});
```

Calculation sequence:

```
1 + 2 = 3;
3 + 3 = 6;
6 + 4 = 10;
```

An example of using an addition operation with four arguments:

```
var decimalUtils = Ext.create("Terrasoft.DecimalUtils");
decimalUtils.evaluate({
  subtract: [10, 5, 3, 1]
});
```

Calculation sequence:

```
10 - 5 = 5;
5 - 3 = 2;
2 - 1 = 1;
```

The toDecimalPlaces() method

The *toDecimalPlaces()* method rounds the number with the specified precision. This precision is specified by the *decimalPlaces* property of the configuration object passed to the constructor (see example below). Banker's rounding is applied to these calculations. Returning value type – *Number*.

Method format: *decimalUtils.toDecimalPlaces(number);*

Possible metric parameters are listed in table 7.

Table 7. Parameters of the toDecimalPlaces() method

Name	Type	Description
number	*Number*	The number to be rounded.

Use case:

```
var decimalUtils =  Ext.create("Terrasoft.DecimalUtils", {
    decimalPlaces: 1
});
decimalUtils.toDecimalPlaces(1.15);         // Returns 1.2
```

The roundValue() method

The *roundValue()* method rounds the number with the precision which is passed in the configuration object. If the precision has not been passed, the method rounds uses the precision of the *decimalPlaces* specified in the constructor configuration object. Banker's rounding is applied to these calculations. Returning value type – *Number*.

Method format: *decimalUtils.roundValue(number [, config]);*

Possible method parameters are listed in table 8.

Table 8. Parameters of the roundValue() method

Name	Type	Description
number	*Number*	The number to be rounded.
config (optional)	*Object*	An object with additional parameters.
config.decimalPlaces (optional)	*Number*	The number of fractional part digits.

Use case:

```
var decimalUtils =  Ext.create("Terrasoft.DecimalUtils", {
    decimalPlaces: 1
});
decimalUtils.roundValue(1.123456, {decimalPlaces: 4});// Returns 1.1235.
```

```
decimalUtils.roundValue(1.123456);// Returns 1.1.
```

The random() method

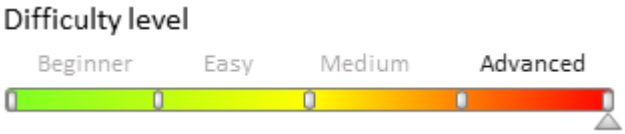
The *random()* method generates a pseudo-random number from 0 to 1 (not including 0 and 1) with the number of decimal places equal to *decimalPlaces*. Returning value type – *Number*.

Method format: *var randomValue = decimalUtils.random();*

Use case

```
var decimalUtils = Ext.create("Terrasoft.DecimalUtils");
// Returns a random number, e.g. 0.35.
var randomValue = decimalUtils.random();
```

Basic macros in the MS Word printables



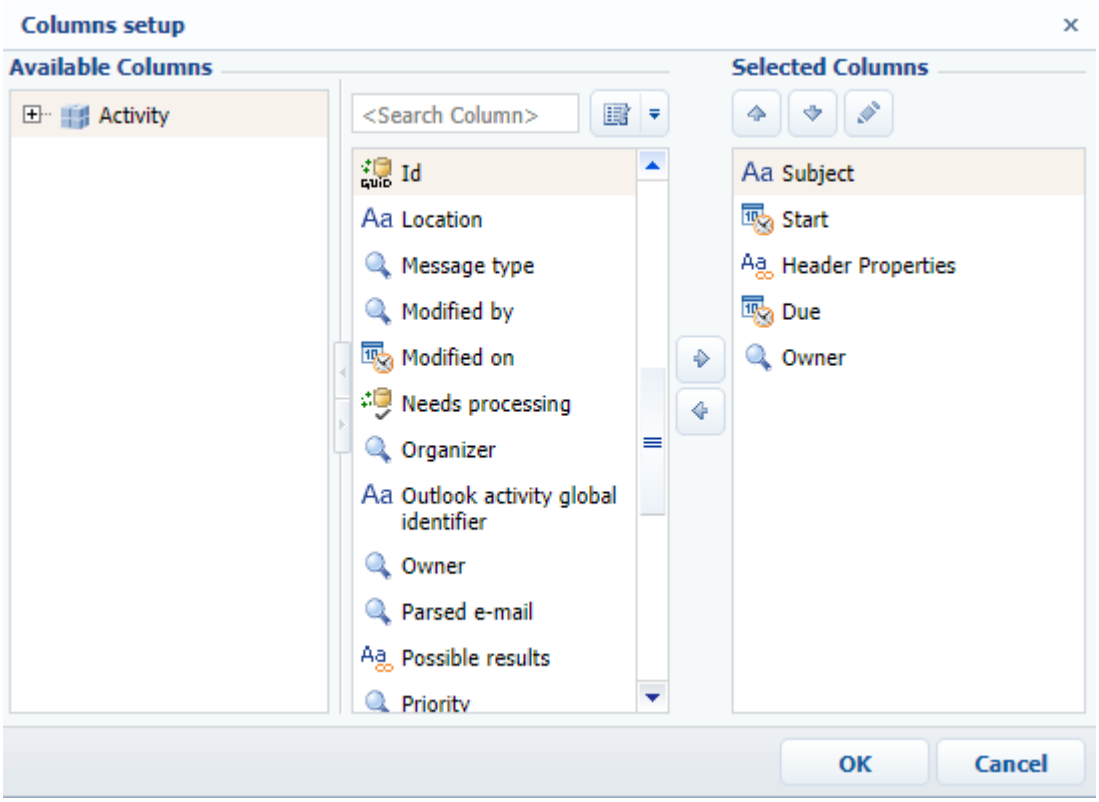
Introduction

A printables can be configured using bpm'online MS Word Report Designer standard tools. More information about MS Word printables can be found in the “The MS Word printables setup” article. Use the macros to implement the specific tasks of printables configuring. A process of the custom macro creation is described in the “**How to create macros for a custom report in Word**” article.

Adding macro

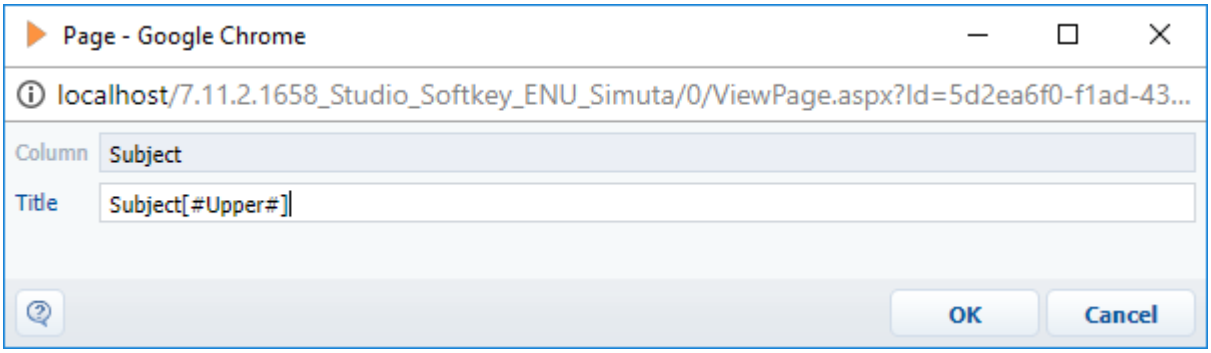
You can add the required macro on the stage of column configuration. For this, select the required column in the [Selected Columns] field (fig. 1).

Fig. 1. — Printable columns list



After that, click the column edit button  and add the macro in the opened window (Fig. 2).

Fig. 2. — Adding macro



Macro recording format

A following format is used for the MS Word printables macros:

```
Column name [#Macro name|Arguments#]
```

The [#Date#] macro

Converts date according to the specified format. If the data format is not specified, then the values will be converted to the default format ("MM-dd-yy"). More information about formats can be found in the MSDN documentation. Argument is optional.

Example:

```
ColumnName[#Date|MM-dd-yy#]
```

When the value "12/30/2016 11:48:24 AM” is entered, the macro will return the “12-30-16” as a result.

The [#Lower#] macro

Converts a string to the lowercase. This macro is used without an arguments.

Example:

```
The ColumnName[#Lower#]
```

When the value "ExamPle" is entered, the macro will return the "example” as a result.

The [#Upper#] macro

Converts a string to the uppercase. If the argument "FirstChar" is passed, only the first character will be converted to the uppercase. Argument is optional.

Examples:

```
ColumnName[#Upper#]
```

When the value "example" is entered, the macro will return the "EXAMPLE” as a result.

```
ColumnName[#Upper|FirstChar#]
```

When the value "example" is entered, the macro will return the "Eample” as a result.

The [#NumberDigit#] macro

Converts a fractional number to a thousand-digit number. A “space” character is the delimiter by default. Arguments are optional.

Examples:

```
ColumnName[#NumberDigit#]
```

When the value "345566777888.567" is entered, the macro will return the "345 566 777 888.567” as a result.

```
ColumnName[#NumberDigit|,#]
```

When the value "345566777888.567" is entered, the macro will return the "345,566,777,888.567” as a result.

> NOTE
>
> If the fractional part is zero, it will not be displayed. For example, if the input vaule is "345566777888.000", the macro will return "345,566,777,888" as a result.

The [#NumberRU#] macro

Converts a number to a text string. The *Cent* argument returns the fractional part of the number without converting it to string. The *Decimal* argument converts the fractional part of the number to string. Arguments are optional.

```
ColumnName[#NumberRU#]
```

When the value "456" is entered, the macro will return the "четыреста пятьдесят шесть” as a result.

When the value "456.78" is entered, the macro will return the "четыреста пятьдесят шесть” as a result. If arguments are not specified, then only decimal part of the number is converted (see *Decimal* argument example).

```
ColumnName[#NumberRU|Cent#]
```

When the value "123.45" is entered, the macro will return the "45” as a result.

When the value "123" is entered, the macro will return the "00” as a result.

```
ColumnName[#NumberRU|Decimal#]
```

When the value "123.45" is entered, the macro will return the "семьсот семьдесят семь целых семьдесят семь сотых” as a result.

The [#Boolean#] macro

Converts boolean value to a user type value. Arguments are required. The *CheckBox* argument converts the entered value to the checkbox element ("☑"/"☐"). Text arguments must match the "Yes, No” format.

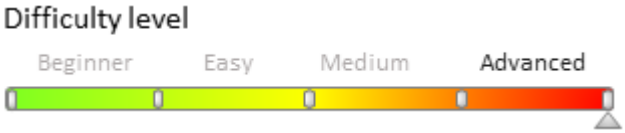
```
ColumnName[#Boolean|CheckBox#]
```

If the column has the “*true*” value, the macro will return "☑" as a result.

```
ColumnName[#Boolean|Yes,No#]
```

If the column has the “*true*” value, the macro will return "Yes” as a result.

Web-to-Case



Introduction

Web-to-Case functionality implements the ability to create cases in the bpm'online by filling the required form fields embedded in a third-party site - landing.

The *ProductCore* package depends on the *WebForms* package, that contains Web-to-Case functionality. This means that landings can be used in all products. Pre-configured base functionality is implemented in the service enterprise, customer center, marketing products and all bundles that these products are part of.

More information about landings can be found in the [Landings] section articles of the corresponding products (such as bpm'online marketing).

Web-to-Case configuration can be done in the system interface. To implement generated JavaScript to a third-party site, you need the basic Web development skills.

The Web-to-Case base functionality allows to configure the following features without programming (using minor improvements on a third-party site):

- The form interface and styles.
- List of the additionally passed fields.
- List of default values for the fields that are not displayed in the form.
- The list of domains from which the case registration for each landing will be possible.
- The address to which the user will be redirected after submitting the form.
- JavaScript event handlers of successful/unsuccessful case registration.
- Additional landings, that can be configured in different way. That makes it possible to distinguish cases created from different sites.

You can modify the project to set up a preliminary handler of case registration through the Web-to-Case with the data validation, correction, creation of related entities and etc. The automatic creation of contact for the registered case is configured in the bpm'online base configuration in the handler of case registration through the Web form.

The logic of the automatic filling of case fields.

In the process of case registration through the Web form, the following fields are recommended for filling: [Name], [Email], [Phone], [Case subject]. The [Case subject] value will be passed to the new case.

The bpm'online will identify the contact by [Name], [Email] and [Phone] fields. The search is performed in a following way:

1. If contact fields matches the [Name], [Email] and [Phone] fields from the filled form, they will be added to the created case.
2. If contact fields matches only the [Name] and [Email] fields from the filled form, they will be added to the created case.
3. If contact fields matches only the [Email] field from the filled form, it will be added to the created case.
4. Otherwise, a new contact is created and the [Name], [Email] and [Phone] fields will be filled in. The created contact is added to the registered case.

If more than one contact are found, then the first contact will be used as contact of the case. Also the case registration date (*RegisteredOn* column) will be automatically filled with the current date and time.

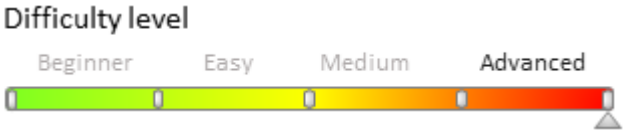
Recommendations for the execution of project solutions

If you need to customize the Web-to-Case, use its base functionality as an example.

To execute the project solution:

1. Create a page schema that is inherited from the *CaseGeneratedWebFormPageV2*. The page should not be a replacement page.
2. Add a record of the new type of landing to the *LandingType* table and localization to the *SysLandingTypeLcz* table.
3. Register the typed page created in the first step (the value of the type is new).
4. If you need preliminary processing of the form data before saving the record in the database, you need to create a class that implements the *IGeneratedWebFormPreProcessHandler* interface. This class is a preliminary handler for case registration. Implement the *Execute()* method. This method is the entry point to the handler. Additional actions are implemented in this method. You can take the *WebFormCasePreProcessHandler* schema as an example.
5. If you need to perform actions after saving the record in the database, you need to create a class that implements the *IGeneratedWebFormPreProcessHandler* interface. This class is a preliminary handler for case registration. Implement the *Execute()* method and perform necessary actions.
6. If you created the registration handlers of the case, register them in the *WebFormProcessHandlers* table. Use an existing record as an example of registration.
7. Edit the script template that forms the configuration JavaScript object of the landing, and place it in the *ScriptTemplate* localized string of the created page. Specify the similar script for all localizations used. You can find an example of the script in the *CaseGeneratedWebFormPageV2* schema.
8. Bind all created data to the package.

Separate query pool



Introduction

Some heavy database requests (DB) can fully occupy database server resources for a long time and thus make it difficult or impossible to work for other users. Among these requests are:

- Incomplete queries in dynamic groups, dashboard blocks.
- Complex analytical samples in dashboard blocks.

To solve this problem it is necessary to limit the resources allocated by the database server for processing Select-requests, or to transfer them to a separate query pool. This will reduce their impact on the work of other users and parts of the system.

NOTE

Only Select-requests can be transfered to the separate query pool and only if they are not part of the transaction.

Separate query pool implementation

MS SQL Server enables you to limit the allocated resources using the built-in Resource Governor tool. However, its ranking capabilities are based on information about the connection, and not a specific request. Bpm'online uses connections from a single pool for all queries, and since all connections are the same they are not available for ranking.

To separate the light and potentially heavy queries, the ability to send requests through a special connection in which the suffix "_Limited" is appended to the App (or Application Name) property of the connection string is added.

For example, specifying the “App = bpmonline” property in the connection string of the ConnectionStrings.config file will result in it being changed to “bpmonline_Limited” in the separate query pool connection. If the App (or Application Name) property is not specified in ConnectionStrings.config, the followinf default value is set for the shared connection: ".Net SqlClient DataProvider", and ".Net SqlClient DataProvider_Limited” in the separate query pool connection.

An example of connection string configuration with the App user property:

```
<add name="db" connectionString="App=bpmonline; Data Source=dbserver\mssql2016;
Initial Catalog=BpmonlineSolution; Persist Security Info=True;
MultipleActiveResultSets=True; Integrated Security=SSPI; Pooling = true; Max Pool
Size = 100; Async = true; Connection Timeout=500" />
```

Thus, when loading dashboards or filtering sections with dynamic groups, the application creates additional database connections that differ from the basic “_Limited” suffix.

Separating the pools will allow database administrators to regulate the allocation of resources to requests from the marked connection.

ATTENTION

No resource restriction occurs in this case. The application only provides an opportunity to use a mark for the ranking of connections in Resource Governor. Please note that the work of Resource Governor is difficult to see on an unloaded server with “short” requests. The effect is noticeable when working with a fully loaded database, and when the “heavy” request is being processed for a long time.

Enabling the separate query pool functionality

To enable the separate query pool functionality, set the *true* value for the *UseQueryKinds* setting in the application’s .\Terrasoft.WebApp\Web.config file.

```
<add key="UseQueryKinds" value="true" />
```

As a result, requests from dashboards and dynamic groups will be sent to connections marked with the “_Limited” suffix.

Resource Governor configuration example

Group and pool configuration is performed using an SQL-script. For example:

```
ALTER RESOURCE POOL poolLimited
WITH
(
    MAX_CPU_PERCENT = 20,
    MIN_CPU_PERCENT = 0
    -- REQUEST_MAX_MEMORY_GRANT_PERCENT = value
    -- REQUEST_MAX_CPU_TIME_SEC = value
```

```
    -- REQUEST_MEMORY_GRANT_TIMEOUT_SEC = value
    -- MAX_DOP = value
    -- GROUP_MAX_REQUESTS = value
);
GO
--- Create a workload group for off-hours processing
--- and configure the relative importance.
CREATE WORKLOAD GROUP groupLimited
WITH
(
    IMPORTANCE = LOW
)
USING poolLimited
GO
ALTER RESOURCE GOVERNOR RECONFIGURE;
GO
```

Learn more about the configuration in the Resource Governor documentation.

For each new connection, the classifier function is used, which returns the name of the group. For example:

```
USE [master]
GO

ALTER FUNCTION [dbo].[fnProtoClassifier]()
    RETURNS sysname
    WITH SCHEMABINDING
AS
BEGIN
    IF(app_name() like '%_Limited')
        BEGIN
        RETURN N'groupLimited'
    END
RETURN N'default'
END;
```

Use case

To execute a query in a separate query pool, get a special *DBExecutor* for it, passing the value of *Limited* from the *QueryKind* enumerator as an optional parameter. Learn more about the *DBExecutor* in the corresponding **Using the DBExecutor for working with the database**. In the following example, *QueryKind* is the argument of the *EnsureDBConnection()* method, the value of which comes in the custom *EntitySchemaQuery* request (*ESQ*-request), and is set to the server *ESQ*-request and then to the *Select*-request.

```
using (DBExecutor executor = userConnection.EnsureDBConnection(QueryKind)) {
   // ...
};
```

Calling *EnsureDBConnection(QueryKind.General)* is equivalent to calling *EnsureDBConnection()* without *QueryKind*.

Thus, if you set the instance of the *Terrasoft.EntitySchemaQuery* class to the *QueryKind.Limited* attribute in the client application, this value will be passed to the server and a special *DBExecutor* (using the marked connection to the database) will be provided.

An example of setting the *QueryKind.Limited* characteristic of a client *ESQ*-request in the *ChartModule* schema:

```
...
getChartDataESQ: function() {
    return this.Ext.create("Terrasoft.EntitySchemaQuery", {
        rootSchema: this.entitySchema,
        queryKind: Terrasoft.QueryKind.LIMITED
    });
```

```
},

...
```

ATTENTION

If there are nested calls to userConnection.EnsureDBConnection(QueryKind) in your code, make sure that you use the same QueryKind value at all nesting levels.

Development recommendations for Right-To-Left mode

Difficulty level

The bpm'online supports displaying text information in Right-To-Left (RTL) mode. Follow these recommendations while developing new functionality to avoid errors in data displaying.

1. Less-code must be written correctly. For example, there must be a semicolon after the value of the CSS property.

2. Avoid defining styles in JavaScript code. If the styles are required, then for the *margin*, *padding*, *border*, *float* and *text-align* styles it is necessary to provide behavior for RTL-mode. For this, use the *Terrasoft.getIsRtlMode()* method:

```
var borderColorCSS = Terrasoft.getIsRtlMode() ? "border-right-color" :  "border-left-color";
```

3. If the CSS style is required only for the RTL mode, it must be “wrapped” with the *html* tag with the attribute dir="rtl":

```
html[dir="rtl"] {
  .links-container label {
    text-align: left;
  }
}
```

4. To rotate the image around the axes, use the rotateY(180g) and scaleX(-1) less-functions:

```
html[dir="rtl"] {
  .links-container img {
    transform: rotateY(180g);
  }
}
...
html[dir="rtl"] {
  .links-container img {
    transform: scaleX(-1);
  }
}
```

5. For the *margin*, *padding* and *border* you cannot use contraction (for example, *margin: 1px 2px 0 0;*). An example of correct use of styles:

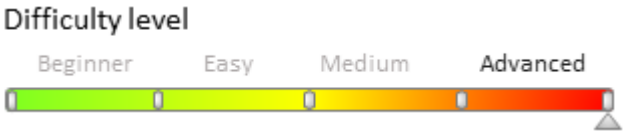
```
html[dir="rtl"] {
  .myclass {
    margin-top: 1px;
    margin-right: 2px;
    margin-bottom: 0;
    margin-left: 0;
    padding-top: 1px;
    padding-right: 0;
```

```
    padding-bottom: 1px;
    padding-left: 0;
    border-top-width: 2px;
    border-right-width: 10px;
    border-bottom-width: 4px;
    border-left-width: 20px;
  }
}
```

6. The *transform: translate(-50%, -50%)* CSS property is used in Left-to-Right mode and in the RTL mode you should set the value of the *transform: translate(50%, -50%)* property as follows:

```
html[dir="rtl"] {
  .links-container img {
    transform: translate(50%, -50%);
  }
}
```

Client static content in the file system



Introduction

Before the version 7.11, at the request of client content (.js, .css files), the application server generated the content dynamically, based on the current structure of package connections and schema dependencies. Generated data were cached and sent to client application.

Starting with version 7.11 all client content is preliminary generated in special application folder ie. it becomes static. When requesting client content, the IIS searches for requested content in this folder and sends it to the client application. Thus, the overall performance of the application is increased and the server load is reduced.

Advantages and disadvantages

Advantages and disadvantages of using the client static content are given in the Table 1.

Table 1. Advantages and disadvantages of using the client static content

Advantages	Disadvantages
Dynamic generation of client content	
No need to pre-generate client content	Processor overload when computing the hierarchy of packages, schemas, and content generation Database overload when for getting the hierarchy of packages, schemas, and content generation Memory consumption for caching client content
Usage of preliminary generated client content	
Minimum CPU load (CPU) Missing database queries Client content is cached by IIS	Need to pre-generate client content

Generating static client content

Client content is generated in the specific folder (.\Terrasoft.WebApp\conf). In contains .js files with schema source code, .css files of styles and .js files of resources of all cultures of the application.

ATTENTION

Starting with version 7.11.1 the .\Terrasoft.WebApp\conf folder also contains images.

ATTENTION

The application's IIS pool user requires modify permission (reading and writing of files and subfolders and deletion of the folder) to the .\Terrasoft.WebApp\conf directory. Without the write permission bpm'online application will not be able to generate static content.

The IIS pool user name is set in the [Identity] property. You can access this property through the [Advanced Settings] menu command on the [Application Pools] tab of the IIS Manager.

The actions to start generation of client content

Primary or secondary generation of static client content starts when the following actions are performed:

- Saving a schema through client schema designer and client objects designer.
- Saving through section wizard and detail wizard.
- Installing and deleting applications from Marketplace and zip archive.
- Applying translation.
- The [Compile all items] and [Compile modified items] actions in the [Configuration] section.

ATTENTION

When deleting schemas and packages from the [Configuration] section you need to perform [Compile all items] and [Compile modified items] actions.

When installing and updating schemas and packages from the SVN you need to perform [Compile all items] action.

NOTE

Only the [Compile all items] action performs full regeneration of client static content. Other actions lead only to regeneration of modified schemas.

Generation of client content with the WorkspaceConsole utility

The *BuildConfiguration* operation was added to the *WorkspaceConsole* utility and this operation performs generation of client content. Operation parameters are listed in table 2.

Table 2. Parameters of the BuildConfiguration operation

Parameter	Details
workspaceName	Workspace name by default (*Default*).
destinationPath	Folder to which the static content will be generated
webApplicationPath	Path to the web applcation from which the information about connection to database will be read. This parameter is optional. If this value has not been indicated, the connection will be established to the database specified in the connection string of the *Terrasoft.Tools.WorkspaceConsole.config* file. If the value is specified, the connection will be established with the database from the

	ConnectionStrings.config file of the web application.
force	If the value is set to *true*, the generation of the content will be performed for all schemas. If the value is set to *false*, the generation will be performed only for modified schemas. This parameter is optional. The value is *false* by default.

Use cases:

```
Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -
workspaceName=Default -destinationPath="C:\WebApplication\BPMOnline\Terrasoft.WebApp"
-force=true -logPath=C:\wc\log

Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -
workspaceName=Default -webApplicationPath="C:\WebApplication\BPMOnline" -
destinationPath="C:\WebApplication\BPMOnline\Terrasoft.WebApp" -force=true -
logPath=C:\wc\log
```

Compatibility with the development in the file system mode

Currently, the development in the file system is no compatible with getting client content from preliminary generated files. For the correct work of the development in the file system you need to disable getting static client content from the file system. Set the "false" for the *UseStaticFileContent* flag in the Web.config file to disable this functions.

```
<fileDesignMode enabled="true" />
...
<add key="UseStaticFileContent" value="false" />
```

Generation of client content when adding a new culture

Execute the [Compile all items] action in the [Configuration] section after adding new cultures.

> ⚠ ATTENTION
>
> If a user cannot log in to the system after adding new culture, you need to access the [Configuration] section by the http://[path to application]/0/dev path and execute the [Compile all items] action.

Changes in the parameter object that generates an image URL (version 7.11.1)

Images in the client part of bpm'online are always being requested by a browser with a specific URL specified in the *src* attribute of the *img* html-element. The *Terrasoft.ImageUrlBuilder (imagurlbuilder.js)* module with the *getUrl(config)* public method that gets the image URL is used in the URL generation. This method receives the *config* configuration JavaScript object that contains an object of parameters in the *params* property. The image URL is being generated on the basis of this object.

Till the 7.11.0 version the structure of the *params* object had the following view:

```
config: {
    params: {
        schemaName: "",
        resourceItemName: "",
        hash: ""
    }
}
```

In this code:

- *schemaName* – schema name (string)

- *resourceItemName* – image name in the bpm’online (string)
- *Hash* – image hash (string).

Starting with version 7.11.1 the *resourceItemExtension* string property that contains file extension (for example, .png) was added to the parameters list. A new structure of the *params* object:

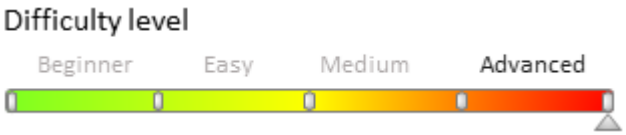
```
config: {
    params: {
        schemaName: "",
        resourceItemName: "",
        hash: "",
        resourceItemExtension: ""
    }
}
```

> ATTENTION
>
> Starting with version 7.11.1 if the *params* object is generated in the custom program code (not obtained from the resources), the *resourceItemExtension* property should be added to the object. In the opposite case, the image will be retrieved from the database, not from the static content. In the next versions, the ability to retrieve an image from the database will be disabled. Therefore, the absence of the *resourceItemExtension* property will cause errors when loading images on a page.

An example of correct generation of configuration object of parameters for getting the URL of a static image:

```
var localizableImages = {
    AddButtonImage: {
        source: 3,
        params: {
            schemaName: "ActivityMiniPage",
            resourceItemName: "AddButtonImage",
            hash: "c15d635407f524f3bbe4f1810b82d315",
            resourceItemExtension: ".png"
        }
    }
}
```

Record deactivation

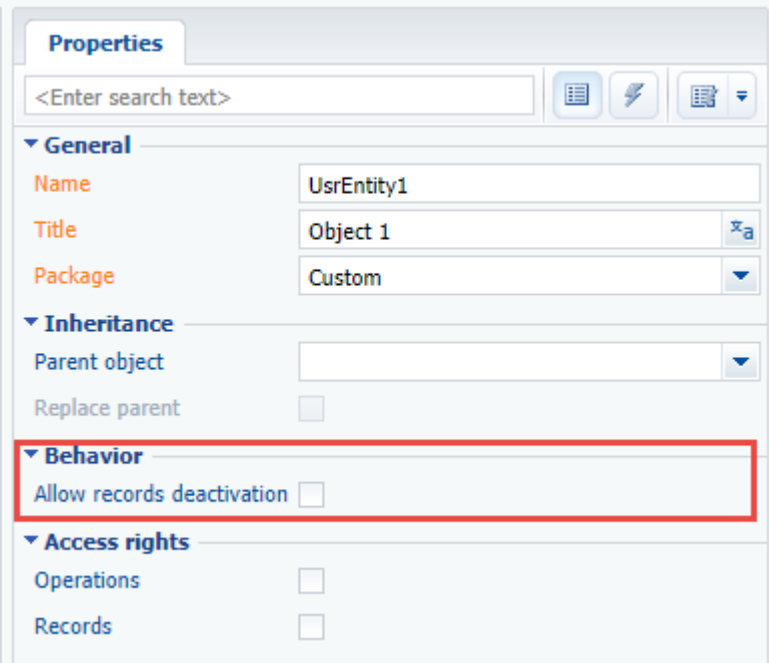


Introduction

In the bpm'online version 7.11.3 you can deactivate records of the system objects to exclude them from the business logic. It can be used if the data is outdated and will never be used. Enable this function with the [Allow record deactivation] property in the object designer (Fig. 1) and it will be enabled after object publication.

> ATTENTION
>
> In 7.11.3, these functions are disabled by default. To enable them, set the "UseRecordDeactivation" setting in the *..\Terrasoft.WebApp\Web.config* file to "true".

Fig. 1. [Allow record deactivation] property



ATTENTION

Deactivation of the records is available for all objects but automatic filtering of all records works only in drop-down lists, on the lookup selection page and in quick filters. The automatic filter is not applied on pages with lookup contents, in the advanced filters and sections.

Use case in program code

The *UseRecordDeactivation* parameter that defines enabling and disabling filtering by inactive records appeared in the *EntitySchemaQuery*. By default the parameter value is *false*. If you change the value to *true*, inactive records will be filtered from the request to select data from the object with enabled record deactivation.

Use case in client code:

```
var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
   rootSchemaName: "MyCustomLookup",
   useRecordDeactivation: true
});
```

Use case in server code:

```
var esq = new EntitySchemaQuery(userConnection.EntitySchemaManager, "ContactType") {
    UseRecordDeactivation = true
};
esq.PrimaryQueryColumn.IsAlwaysSelect = true;
var sqlQuery = esq.GetSelectQuery(userConnection).GetSqlText();
Console.WriteLine(sqlQuery);
```

The text of resulting SQL query is assigned to the sqlQuery variable. After the initialization of the *EntitySchemaQuery* instance the query will look like following:

```
SELECT
  [ContactType].[Id] [Id]
FROM [dbo].[ContactType] [ContactType] WITH(NOLOCK)
WHERE
  [ContactType].[RecordInactive] = 0
```

Monitoring of private properties overriding. The Terrasoft.PrivateMemberWatcher class

Difficulty level

Introduction

Starting with version 7.12.0 version you can override private properties via the *Terrasoft.PrivateMemberWatcher* class. When defining a custom class, this functionality checks whether the private properties or methods declared in the parent classes have been overridden. The warning is displayed in the browser console in debug mode.

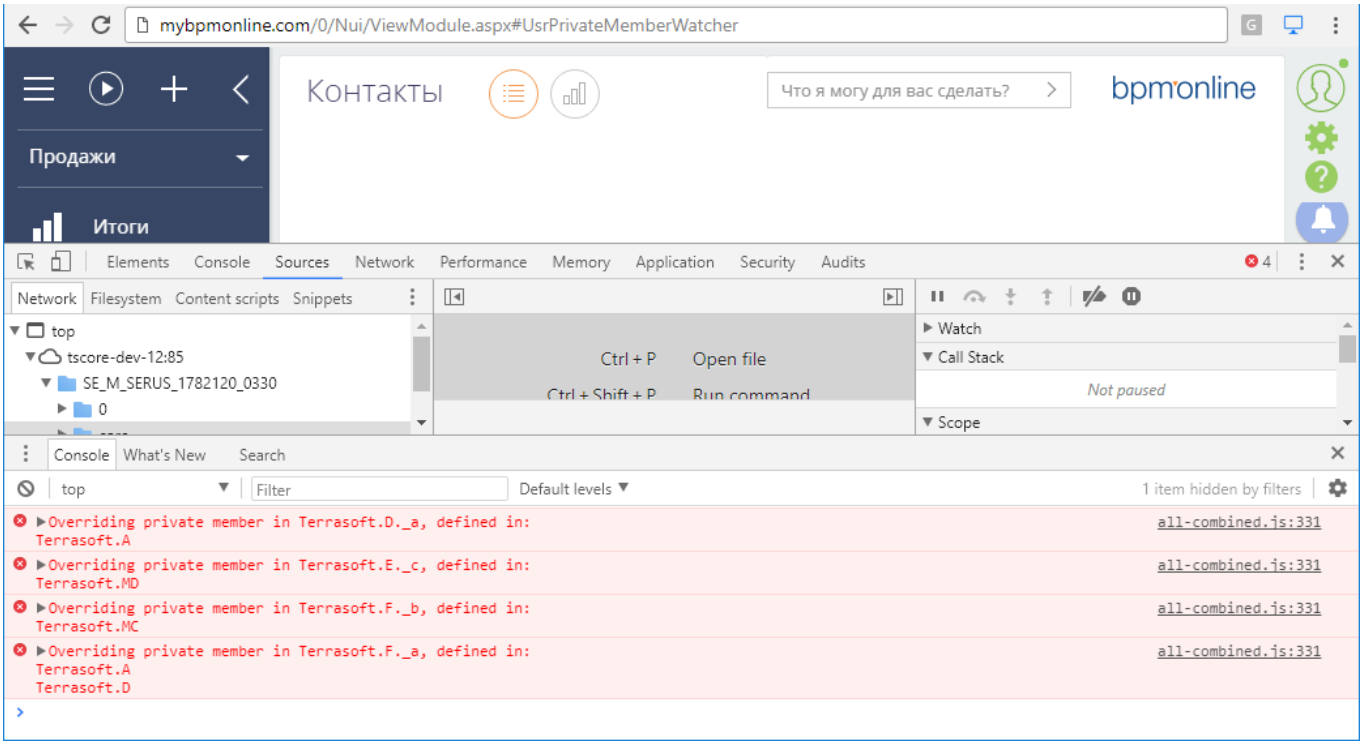
> ⚠ ATTENTION
>
> In bpm’online, the private properties and methods of the class are starting with underscore (for example, *_privateMemberName*).

For example, a module schema with the source code given below was added to a custom package:

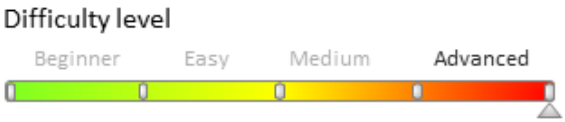
```
define("UsrPrivateMemberWatcher", [], function() {
    Ext.define("Terrasoft.A", {_a: 1});
    Ext.define("Terrasoft.B", {extend: "Terrasoft.A"});
    Ext.define("Terrasoft.MC", {_b: 1});
    Ext.define("Terrasoft.C", {extend: "Terrasoft.B", mixins: {ma: "Terrasoft.MC"}});
    Ext.define("Terrasoft.MD", {_c: 1});
    // Overriding _a property.
    Ext.define("Terrasoft.D", {extend: "Terrasoft.C", _a: 3, mixins: {mb:
"Terrasoft.MD"}});
    // Overriding _c property.
    Ext.define("Terrasoft.E", {extend: "Terrasoft.D", _c: 3});
    // Overriding _a and _b properties.
    Ext.define("Terrasoft.F", {extend: "Terrasoft.E", _b: 3, _a: 0});
});
```

Then, after loading the module via the browser address bar (see “**Client Modules**”), a number of warnings will be displayed in the console (Fig. 1).

Fig. 1. Warning message for overriding private members of the class



The [Timeline] tab



Introduction

Starting from version 7.12.0 you can use the [Timeline] tab for quick analysis of customer cooperation, opportunity, case, etc. history in bpm’online. This tab is available by default in the [Contacts], [Accounts], [Leads], [Opportunities] and [Cases] sections.

The database tables

The following tables are provided in the database for setting up the timeline:

- *TimelinePageSetting* – for setting up sections and their tiles (table 1).
- *TimelineTileSetting* – for setting up all existing and custom timeline tiles (table 2).
- *SysTimelineTileSettingLcz* – for localizing tile names (see "**Localization tables**”).

Table 1. – TimelinePageSetting table primary columns

Column	Details
Id	Record identifier.
Key	Key – the name of section page schema. For example, AccountPageV2, ContactPageV2, etc.
Data	Section timeline setup in JSON format.

Table 2. – TimelineTileSetting table primary columns

Column	Details
Id	Record identifier.
Name	Tile caption that will be displayed in the filter menu. It must have plural form, for example, “Tasks”. Localization is preformed via the *SysTimelineTileSettingLcz* table. If this field is not populated, the

	tile caption will be derived from the entity or type schema name.
Data	Section timeline setup in JSON format (table 3).
Image	The tile icon that will be displayed in the filter menu and on the left side of the tile on the [Timeline] tab.

Table 3. – Timeline tile configuration parameters in JSON format.

Column	Details	If required	Example
entityConfigKey	Tile key. It should match the *Id* in the *TimelineTileSetting* table of the corresponding existing tile that should be displayed for the entity.	No	706f803d-6a30-4bcd-88e8-36a0e722ea41
entitySchemaName:	Name of the entity object schema.	Yes	Activity
referenceColumnName	Name of the object column that will be used for selecting records.	Yes	Account
masterRecordColumnName	Name of the parent record column that will be used for selecting records.	Yes	Id
typeColumnName	Name of the type column .	No	Type
typeColumnValue	Value of the type column.	Should only be applied when typeColumnName is indicated.	fbe0acdc-cfc0-df11-b00f-001d60e938c6
viewModelClassName	The view model class name of the existing tile.	No. If the value is not populated, the *BaseTimelineItemViewModel* base class will apply.	Terrasoft.ActivityTimelineItemViewMode
viewClassName	Name of the existing tile view class.	No. If the value is not populated, the *BaseTimelineItemViewl* base class will apply.	Terrasoft.ActivityTimelineItemView
orderColumn	Column for sorting.	Yes	StartDate
authorColumnName	Column for the author.	Yes	Owner
captionColumnName	Column for the caption.	Yes, if the *messageColumnName* column is not indicated.	Title
messageColumnName	Column for messages.	Yes, if the *captionColumnName* column is not indicated.	DetailedResult
caption	Tile caption that will be displayed in the filter menu. It must have plural form, for example, “Tasks”. It is used for setting a tile caption that would differ from the one indicated in the *Name* field of the corresponding tile setting in *TimelinePageSetting*.	No	My Activity

columns	Setup array for additional tile columns.	No	
columnName	Path to the entity object column.	Yes	Result
columnAlias	Column alias in the tile model view.	Yes	ResultMessage
isSearchEnabled	Indicates the capability of text search according to the column value (for text columns only).	No	true

Adding the [Timeline] tab to the section

> ⚠ NOTES
>
> Adding the [Timeline] tab tiles to the new section is described in the "**How to create the [Timeline] tab tiles bound to custom section**" article.

To add the [Timeline] tab to the section page and display records thereon:

1. Add a new record to the *TimelinePageSetting* table.

2. Populate the corresponding columns (table 1). Indicate the section page schema name in the *Key* column. For example, if you need to add a tab to the {Accounts} section, the *Key* column value will be "AccountPageV2". The *Data* column contains the configuration of timeline tiles that are displayed on the indicated section tab in JSON format (table 3).

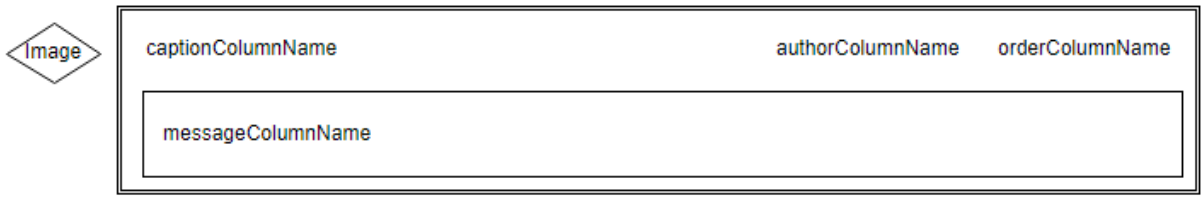
> ⚠ ATTENTION
>
> The [Timeline] tab will not be displayed on the section record edit page if the tile configuration in the *Data* column is not available or if there exist errors (for example, syntax error) in the configuration.

Usage of base tile

To start using the timeline in a section, perform base tile configuration (fig.1). The base tile compound elements:

- icon
- caption
- author
- date (sorting)
- message

Fig. 1. – The base tile element location



Example

Add the [Contract] tile to the [Accounts] section page. Sorting should be performed according to the *StartDate* column; the caption values, author and tile messages should be derived from the *Number*, *Owner* and *Notes* columns correspondingly.

Case implementation

1. Add a new record (or update an existing record) in the *TimelinePageSetting* table.

2. Set the "AccountPageV2” value for the *Key* column and populate the *Data* column with the following JSON object:

```
[
```

```
  {
    "entityConfigKey": "0ef5bd15-f3d3-4673-8af7-f2e61bc44cf0",
    "entitySchemaName": "Contract",
    "referenceColumnName": "Order",
    "orderColumnName": "StartDate",
    "authorColumnName": "Owner",
    "captionColumnName": "Number",
    "messageColumnName": "Notes",
    "caption": "My Contracts",
    "masterRecordColumnName": "Id"
  }
]
```

ATTENTION

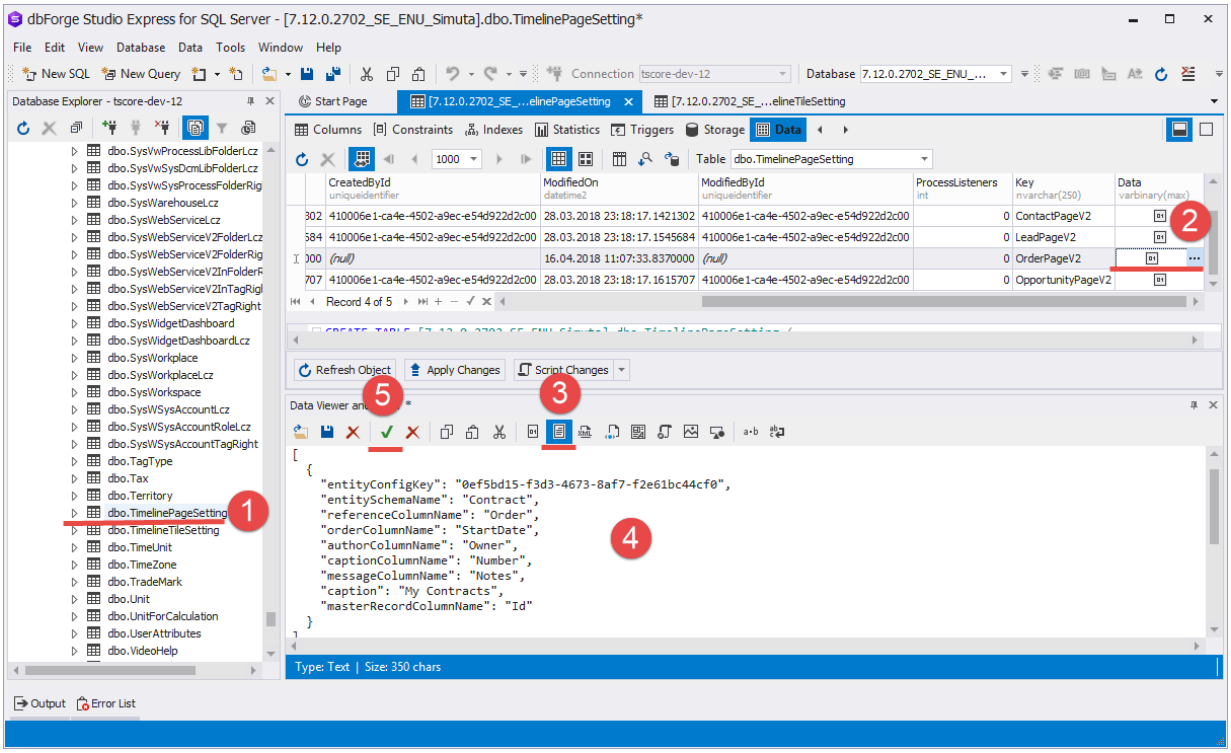
The [Orders] base tile is used in the following case. This tile has a record in the TimelineTileSettings table with the *0ef5bd15-f3d3-4673-8af7-f2e61bc44cf0* Id.

ATTENTION

As the data in the Data column are stored in the varbinary(max) form, use specific editor (such as dbForge Studio Express for SQL Server) to modify them (Fig. 2). To do this:

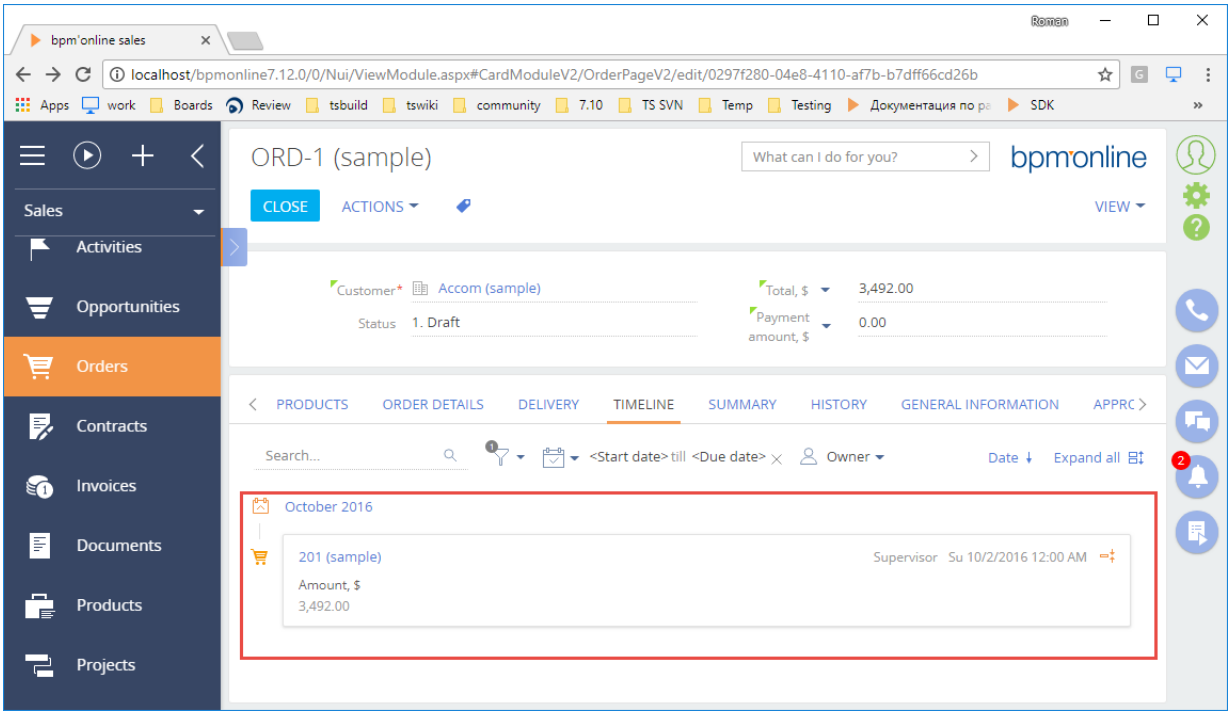
1. Select a table.
2. Select the necessary column of the record and click the edit button.
3. Enter the text data display mode in the data editor.
4. Add necessary data.
5. Save the changes in the data editor.

Fig. 2. Editing data via the dbForge Studio Express for SQL Server



The result of the base tile usage on the [Timeline] tab in the [Accounts] section is shown in fig.3

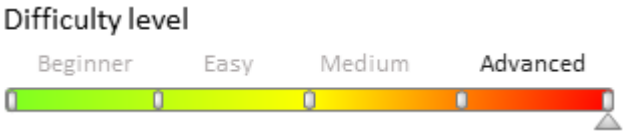
Fig. 3. – The [Timeline] tab in the [Accounts] section.



See also

- **How to create the [Timeline] tab tiles bound to custom section**

Server content in the file system



Introduction

Till the version 7.11.3 inclusive, information about tan object for the Runtime mode was stored in the specific automatically generated class, that was inherited from the *EntitySchema* class (see “**.NET class libraries of platform core (on-line documentation)**”). For example, for the [Contact] object the *ContactSchema* class is generated according to the object schema.

NOTE

Class generation was performed during compilation of the Terrasoft.Configuration.dll library, for example on clicking the [Compile all items] button in the [Configuration] section.

Starting version 7.12.0 information about the object for the Runtime mode is stored in the specific database (server content) located in the .*Terrasoft.WebApp\conf\runtime-data* folder of the deployed bpm’online application.

ATTENTION

The .*Terrasoft.WebApp\conf\runtime-data* folder should have access permission for modification (permissions for reading and writing files and sub-folders, and deleting the folder) for the user of the IIS pool in which the application is launched. Otherwise, the bpm'online will not be able to generate server content.

The IIS pool user name is set in the [Identity] property. You can access this property via the [Advanced Settings] menu on the [Application Pools] tab of IIS manager.

NOTE

For backward compatibility in version 7.12.0, object schema classes are still being generated. In the nearest versions the generation of the schema classes of the object will be disabled.

Server content generation

Primary or secondary generation of server content starts when the following actions are performed:

- Saving the schema in the object designer.
- Saving through section wizard and detail wizard.
- Installing and deleting applications from Marketplace and zip archive.
- The [Compile all items] and [Compile modified items] actions in the [Configuration] section.

ATTENTION

When deleting schemas and packages from the [Configuration] section you need to perform [Compile all items] and [Compile modified items] actions.

When installing or updating a package from SVN, you need to perform the [Compile all items] action.

NOTE

Only the [Compile all items] action performs full regeneration of client static content. Other actions lead only to regeneration of modified schemas.

Generation of client content with the WorkspaceConsole utility

Use the *BuildConfiguration* operation to generate the server content via the *WorkspaceConsole* utility. Operation parameters are listed in table 1.

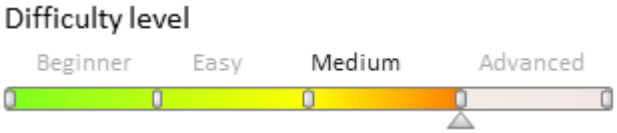
Table 1. Parameters of the BuildConfiguration operation

Parameter	Details
workspaceName	Workspace name by default (*Default*).
destinationPath	Folder to which the static content will be generated
webApplicationPath	Path to the web applcation from which the information about connection to database will be read. This parameter is optional. If this value has not been indicated, the connection will be established to the database specified in the connection string of the *Terrasoft.Tools.WorkspaceConsole.config* file. If the value is specified, the connection will be established with the database from the *ConnectionStrings.config* file of the web application.
force	If the value is set to *true*, the generation of the content will be performed for all schemas. If the value is set to *false*, the generation will be performed only for modified schemas. This parameter is optional. The value is *false* by default.

Use cases:

```
Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -
workspaceName=Default -destinationPath="C:\WebApplication\BPMOnline\Terrasoft.WebApp"
-force=true -logPath=C:\wc\log

Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -
workspaceName=Default -webApplicationPath="C:\WebApplication\BPMOnline" -
destinationPath="C:\WebApplication\BPMOnline\Terrasoft.WebApp" -force=true -
logPath=C:\wc\log
```

Logging in bpm’online. Log4net



Introduction

Logging is useful for localization of application troubles. Bpm’online enables logging for all main operations.

The Lof4net solution is used for logging. This tool enables to perform logging of parameters from different components of the application into separate log files.

Logging is performed separately for the application loader and for the *Default* configuration. To set up logging, modify the *..\Terrasoft.WebApp\log4net.config* configuration file.

Storing log data

> ATTENTION
>
> Location of the log files depends on the value of Windows system variables.

By default the loader log files are located by following path:

```
[TEMP]\BPMonline\Site_[{SiteId}]\[{ApplicationName}]\Log\[{DateTime.Today}]
```

Example:

```
C:\Windows\Temp\BPMonline\Site_1\bpmonline7121\Log\2018_05_22
```

Files with the *Default* configuration logs are located by following path:

```
[TEMP]\BPMonline\Site_[{SiteId}]\[{ApplicationName}]\[ConfigurationNumber]\Log\
[{DateTime.Today}]
```

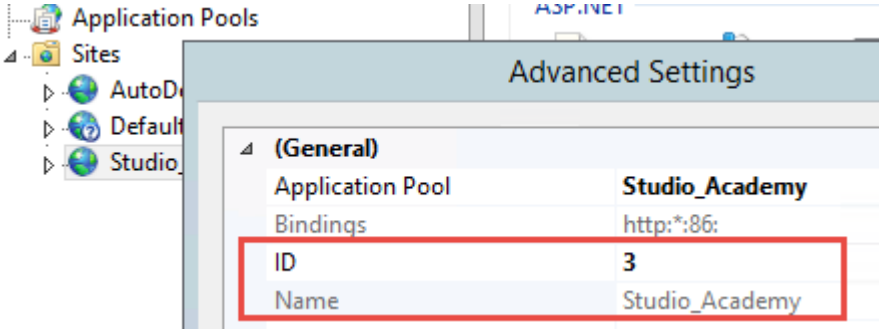
Example:

```
C:\Windows\Temp\BPMonline\Site_1\bpmonline7121\0\Log\2018_05_22
```

Variables specified in the square brackets:

- [TEMP] – the base folder. By default the *C:\Windows\Temp* folder is used by IIS and the *C:\Users\{User name}\AppData\Local\Temp* folder used by Visual Studio (IIS Express).
- [{SiteId}] – site number. For the IIS, the number is specified in the site advanced settings (Fig. 1). For the Visual Studio the number is 2.
- [{ApplicationName}] – application name (Fig. 1).
- [ConfigurationNumber] – configuration number. The number for the *Default,* configuration usually is 0.
- [{DateTime.Today}] – logging date.

Fig. 1. Advanced setting of the IIS site



Changing the logging level

By default the logging level for all bpm’online components is set to provide maximal performance for the application. Possible levels of logging in order of increasing priority:

- ALL – logging of all events. Significantly reduces application performance.
- DEBUG – logging all events at debugging.
- INFO – logging of errors, warnings and messages.
- WARN – logging of errors and warnings.
- ERROR – logging of errors.
- FATAL – logging only errors that lead to the termination of the component being logged.
- OFF – logging disabled.

Example 1. Set the maximum level of logging for all components

To do this, specify the ALL level in the <root> XML element of the *.\Terrasoft.WebApp\log4net.config* file.

```
<root>
    <level value="ALL" />
    <appender-ref ref="commonAppender" />
</root>
```

Example 2. Set logging of errors when working with SVN

To do this, specify the ERROR level in the <logger name="Svn"> XML element of the *.\Terrasoft.WebApp\log4net.config* file.

```
<logger name="Svn" >
    <level value="ERROR" />
    <appender-ref ref="SvnAppender" />
</logger>
```