bpm'online

Bpm'online
Development Guide

Simplify the future

Table of Contents

Getting started with the bpm’online platform 10
Architecture 10
Application infrastructure 10-14
Components 14-16
Packages, schemas, modules 16-21
Application interface and structure 21
Main menu 21-22
Sections 22-23
Section lists 23-26
Section analytics 26
Section actions 26-27
Filters 27-29
Tags 30
Record edit page 30-31
Details 31-34
Mini-page 34
Modal windows 34-35
Communication panel 35-36
Command line 36-37
Action dashboard 37
Development tools 38
How to start development 38
Development process organization 38-39
Organizing a development environment 39-41
Recommended development sequence 41-45
Development rules 45-46
How to deploy bpm'online on-site 46-57
Deploying the bpm'online "cloud" application 57-59
Create user and setup workspaces 59
Create repository in SVN server 59-62

Working with packages 62
Package structure and contents 62-65
Package dependencies. Basic application packages 65-70
Package [Custom] 70-71
Creating and installing a package for development 71-75
Committing a package to repository 75-77
Installing packages from repository 77-80
Updating package from repository 80-81
Installing packages from an application 81-85
Exporting packages from the application interface 85-87
Creating a package in the file system development mode 87-94
Transferring changes between the working environments 94
Transferring changes using schema export and import 94-96
Transferring changes using SVN 96-98
Transferring changes using WorkspaceConsole 98-101
Creating a custom client module schema 101-106
Creating the entity schema 106-115
Creating the [Source code] schema 115-116
Development resources 116
Built-in development tools 116-117
General information about the [Configuration] section 117
Source code and metadata viewport 117-119
Designers of configuration items 119-120
Workspace of the Object Designer 120-121
Module designer 121-124
Source code designer 124-126
Process designer workspace 126-127
User task designer workspace 127-128
Workspace of image list designer 128-129
DevExpress report designer 129-130
Setting up the report designer connection with server 130-131
Report designer workspace 131-135

Report designer features 135-139
Development in the file system 139-142
Visual Studio settings for development in the file system 142-147
Working with the server side source code in Visual Studio 147-152
Working with the client code in the file system 152-155
Working with SVN in the file system 155-158
Working with WorkspaceConsole 158
WorkspaceConsole settings 158-159
WorkspaceConsole parameters 159-163
Exporting packages from database 163-165
Saving packages to the database 165-168
Saving SVN packages 168-170
Client code debugging 170-174
Server code debugging 174-182
Bpm'online development cases 183
Section business logic 183
Creating new section 183-186
Adding an action to the list 186-188
How to add a section action: handling the selection of a single record 188-191
How to add a section action: handling the selection of several records 191-195
Handling the selection of several records. Examples 195-202
How to add a button to a section 202-205
How to highlight a record in the list in color 205-207
How to add a fixed filters block to the section 207-210
Page configuration 210
Setting the edit page fields using business rules 210-213
Adding the [FILTRATION] rule to the edit page 213-216
How to hide the edit page field by a specific condition 216-222
How to block the edit page field by a specific condition 222-224
How to make the field required by a specific condition 224-227
Business rules created in wizards 227-229
Adding an action to the edit page 229-234

Control elements 234
Adding a new field to the edit page 234-239
Adding a button to the edit page 239-241
How to add a button to an edit page in the new record add mode 241-244
How to add a button to an existing record edit page 244-249
How to add a field with an image to the edit page 249-256
How to add the color select button to the edit page 256-259
How to add multi-currency field 259-266
How to add a computed field 266-271
How to set a default value for a field 271-276
How to add the field validation 276-283
Using filtration for lookup fields. Examples 283-286
Adding an action panel 286-289
Adding a new channel to the action dashboard 289-294
Displaying contact's time zone 294-300
Adding details 300
Adding an edit page detail 300-309
Adding a detail with an editable list 309-316
Adding a detail with a lookup 316-324
Adding multiple records to a detail 324-327
Creating a custom detail with fields 327-332
Advanced settings of a custom detail with fields 332-337
Creating a detail in wizards 337-341
How to hide menu commands of the detail with list 341-342
Business processes 342
How to add auto-numbering to the edit page field 342-346
How to run a process from client module 346-350
Creating custom [User task] process element 350-360
Typical customizations 360
How to create a mini-page 360-368
Adding a mini-page to a module 368-373
Adding pop-up hints 373-380

How to modify sales pipeline calculations 380-384
How to add extra filtering to a pipeline 384-386
Adding a custom dashboard widget 386-392
The Terrasoft.AlignableContainer custom element 392-401
Adding a duplicate search rule 401-404
Junk case custom filtering 404-405
Analytics 405
How to create own macros for a MS Word printable 405-413
Working with data 413
CRUD-operations in configuration 413
The use of EntitySchemaQuery implementation on client 413-414
Building of paths to columns relative to root schema 414-415
Adding columns to a query 415-418
Getting query result 418-419
EntitySchemaQuery filters handling 420-423
CRUD-operations on server side 423
Composing add data queries 423-424
The use of EntitySchemaQuery for creation of queries in database 424-435
Composing modify data queries 435-436
Composing delete data queries 436
Web-services in configuration 436
How to create own configuration service 436-438
How to call a configuration service 438-440
Reading multilingual data with EntitySchemaQuery 440-442
Views localization 442-447
Working with the localized data via Entity 447-451
Adding a multilingual terminator to an object schema 451-453
Using the DBExecutor for working with the database 453-454
Mobile application 454-455
Mobile application debugging 455-459
How to add a custom section to a mobile application 459-472
Working with editable lists in the mobile app 472-474

Adding the [Attachments] detail to the section in mobile application 474-479
Adding a custom widget to the mobile application 479-481
Service products customization 481-482
Adding a new rule for the case deadline calculation 482-487
Adding a macro handler in email templates 487-489
Creating Web-to-Case landing pages 489-494
How to hide feed area in the agent desktop 494-496
Lending product customization 496
How to create custom verification action page 496-500
Using the EntityMapper schema 500-506
Recommended integration options 507
Authenticating external requests to bpm'online web-services 507
The AuthService.svc authentication service 507-511
Protection from CSRF attacks during integration with bpm'online 511-512
DataService web service 512-514
DataService. Adding records 514-521
DataService. Reading records 521-531
DataService. Data filtering 531-537
DataService. Using macros 537-541
DataService. Updating records 541-546
DataService. Deleting records 546-550
DataService. Batch queries 550-554
OData 554
Possibilities for the bpm'online integration over the OData protocol 554-556
Working with bpm'online objects over the OData protocol using Http request 556-564
Working with bpm'online objects over the OData protocol WCF-client 564-570
Examples of requests for filter selection 570-574
Integration of third-party sites via iframe 574-578
Platform description 579
System Settings 579
Setting user session timeout 579-581

Working with data structure 582
Configuration localizable resources 582-585
Localizable resource structure and use 585-587
Localization tables 587-588
Bound data structure 588-590
User interface 590
AMD concept Modules 590-592
Modular development principles in bpm'online 592-598
Client Modules 598-603
Client view model schemas 603-604
Mixins. The "mixins" property 604-605
Attributes. The "attributes" property 605-607
Messages. The "messages" property 607-608
Methods. The "methods" property 608-609
Rules. The "rules" property 609
Business rules. The businessRules property 609-610
Modules. The "modules" property 610-611
The "diff" array 611-613
Alias mechanism 613-616
Schema formatting requirements for compatibility with wizards 617-621
Handling a data context loss 621-625
Properties. The "properties" property 625-626
Controls 626
Controls. Introduction 626-627
Details 627-635
The [Connected entity profile] control 635-645
SourceCodeEditMixin class description and work examples. 645-648
Dashboard widgets 648-651
Charts 651-653
Metrics 653-654
Gauge 654-655
Lists 655-656

Web-page 656-657
Sales pipeline 657-658
Integration 658
Phone integration 658-661
Oktell 661-664
Webitel 664-665
Asterisk 665-667
Email integration 667
Working with email threads 667-669
Mobile application 669-670
Mobile app architecture 670-674
Mobile application manifest 674-676
Manifest. Application interface properties 676-678
Manifest. Data and business logic properties 679-681
Manifest. Application synchronization properties 681-685
Mobile application background update 685
Resolving synchronization conflicts automatically 686
Batch mode export 686-687
Getting the settings and data from the [Dashboards] section 687-689
Self-service Portal 689-690
ClientMessageBridge 690
ClientMessageBridge . Message history save mechanism 690-692
ClientMessageBridge. API description 692-693
ClientMessageBridge. The client-side WebSocket message handler 693-698
Sync Engine synchronization mechanism 698
Bpm'online synchronization with external storages 698-705
Synchronizing metadata in bpm'online 705-708
Synchronizing tasks with MS Exchange 708-710
Synchronizing email with MS Exchange 710-713
Synchronizing contacts with MS Exchange 713-716
Synchronizing appointments with MS Exchange 716-718
Data Enrichment 718

API of cloud service for enriching contacts from email 718-721
Bpm'online lending 721
Terrasoft.Configuration.EntityMapper class 721-724
DataManager class description and use cases 724-728
Feature Toggle. Mechanism of disabling functionality 728-730
The MoneyUtilsMixin mixin 730-733
The DecimalUtils module 733-738
Basic macros in the MS Word printables 738-739
Web-to-Case 739-740
Development recommendations for Right-To-Left mode 740-742

Getting started with the bpm'online platform

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

Architecture

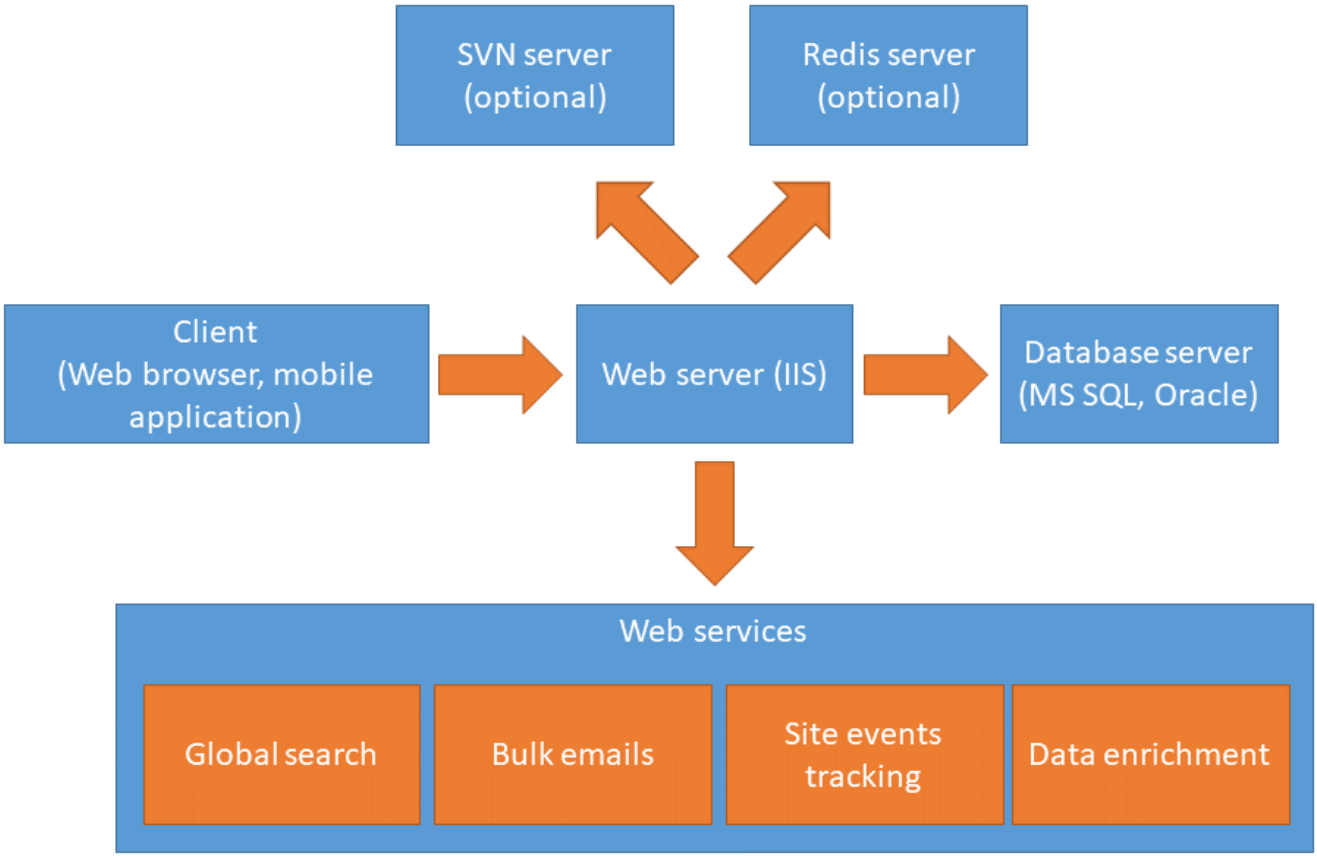
- **Application infrastructure**
- **Components**
- **Packages, schemas, modules**

Application infrastructure

Introduction

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 optional ones:

- database session server (Redis)

- version control server
- 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 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 configurations Modifications are transferred with the packages through version control system.
- 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

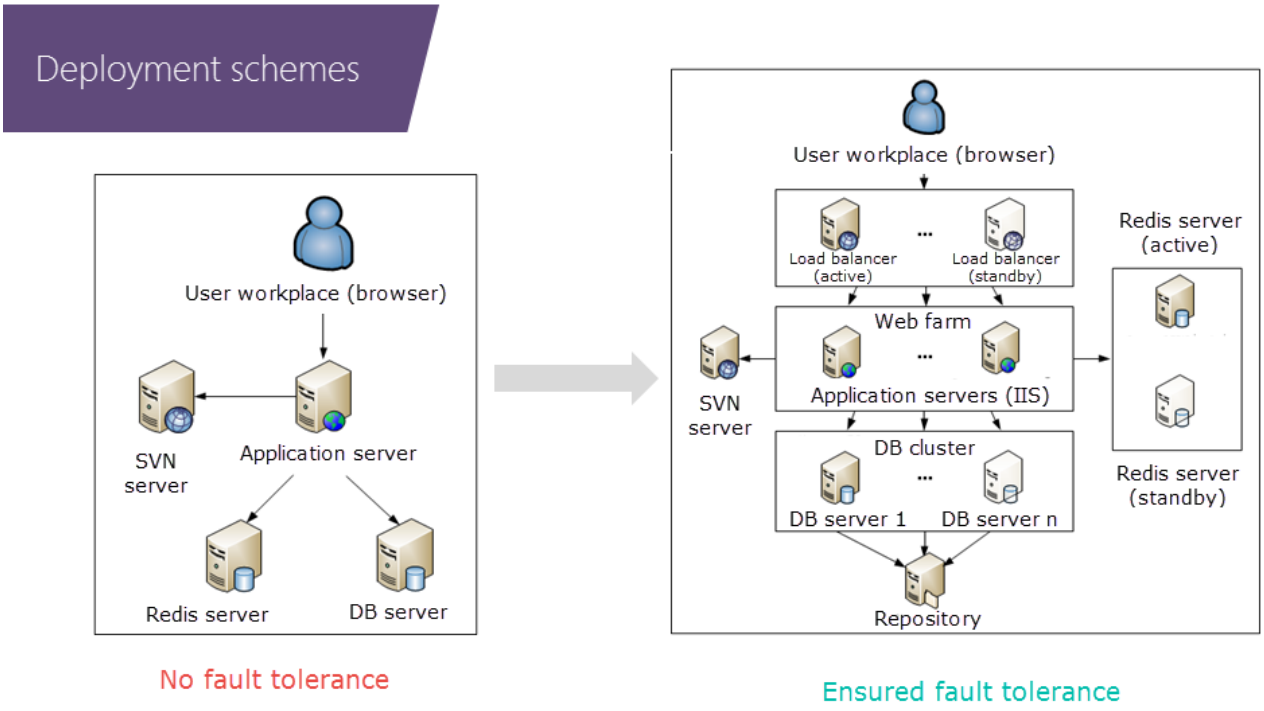
Account data enrichment service uses different search technologies to find information about accounts and their communications from the open Internet sources.

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

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.

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).

Cloud deployment

In the cloud deployment option, the application is deployed on the Amazon servers. 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.

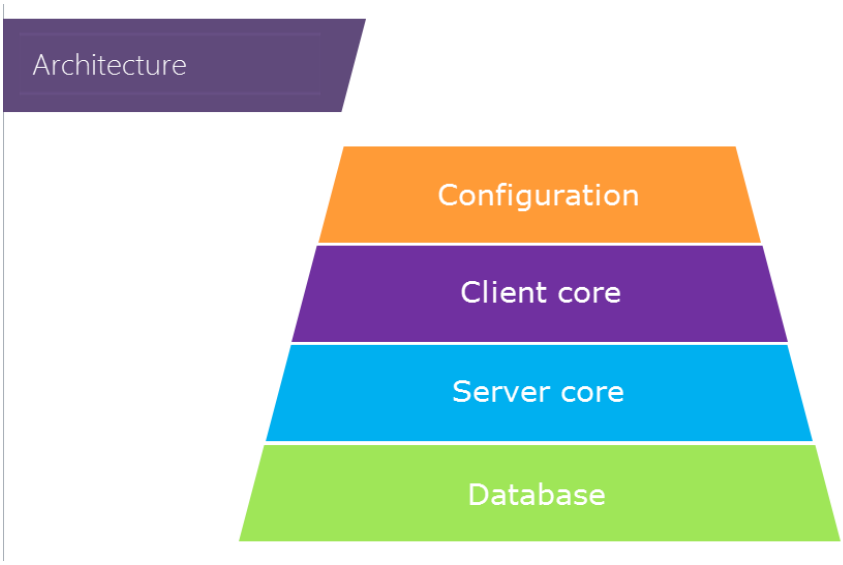
Advantages of the Cloud deployment:

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

Components

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

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.

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 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 library 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:

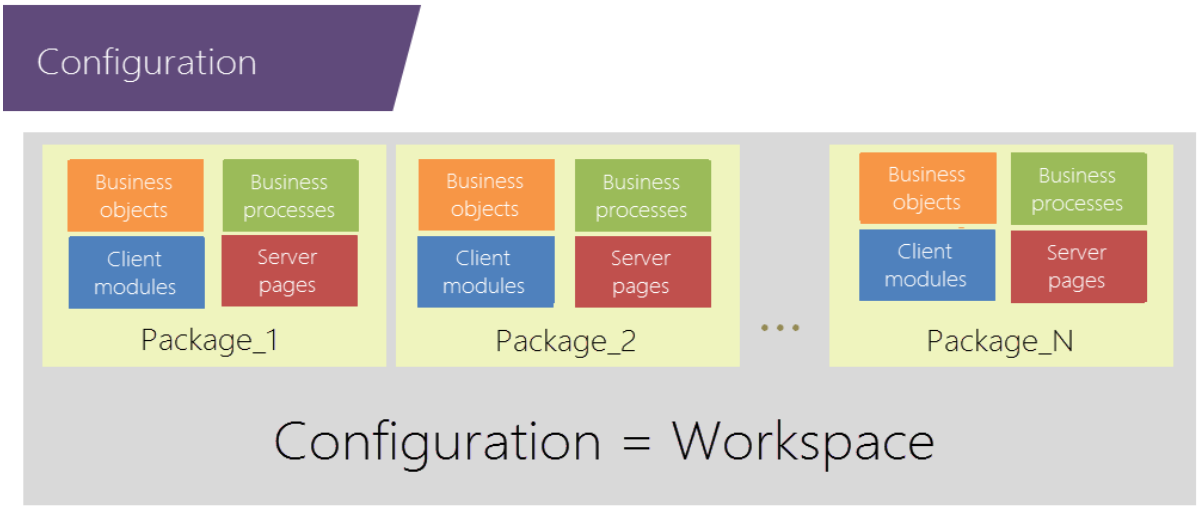
- business objects – entities that store data and connect a database table to a class on the server side;
- business processes – customizable elements that are visual algorithms of user activities;
- client modules;
- server pages, which are sometimes used as record edit pages in the lookups (this is a legacy functionality).

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

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 and are installed by default.
- Custom packages are packages created by system users.

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.

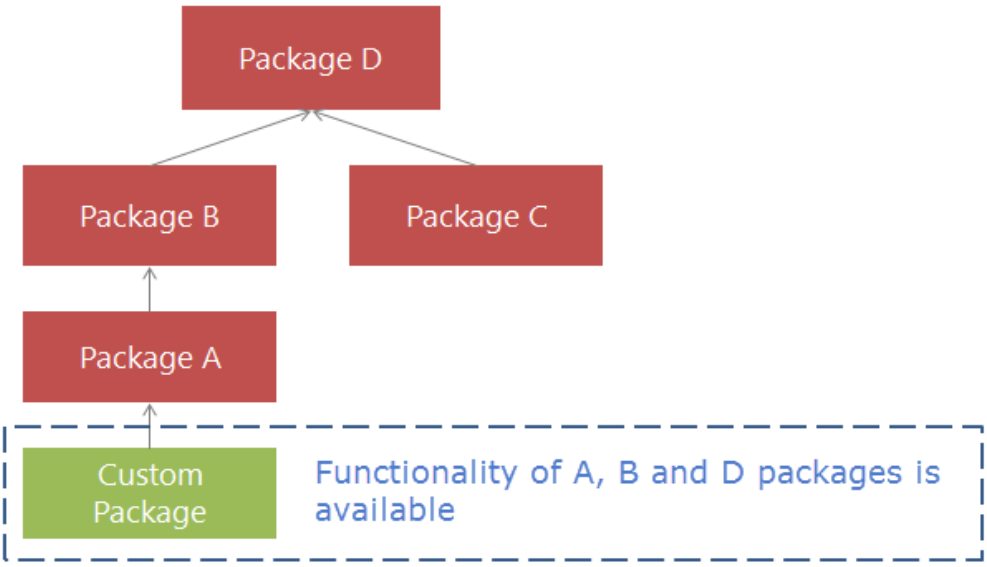
> *NOTE*
>
> The replacement of a single base element can be implemented in several custom packages. The final implementation of a replacing element in a compiled configuration is determined by the hierarchy of all packages that contain replacing elements for the base package.

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 DevExpress 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.

For more information about schema and manager types, please refer to the “Platform description” section. 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.

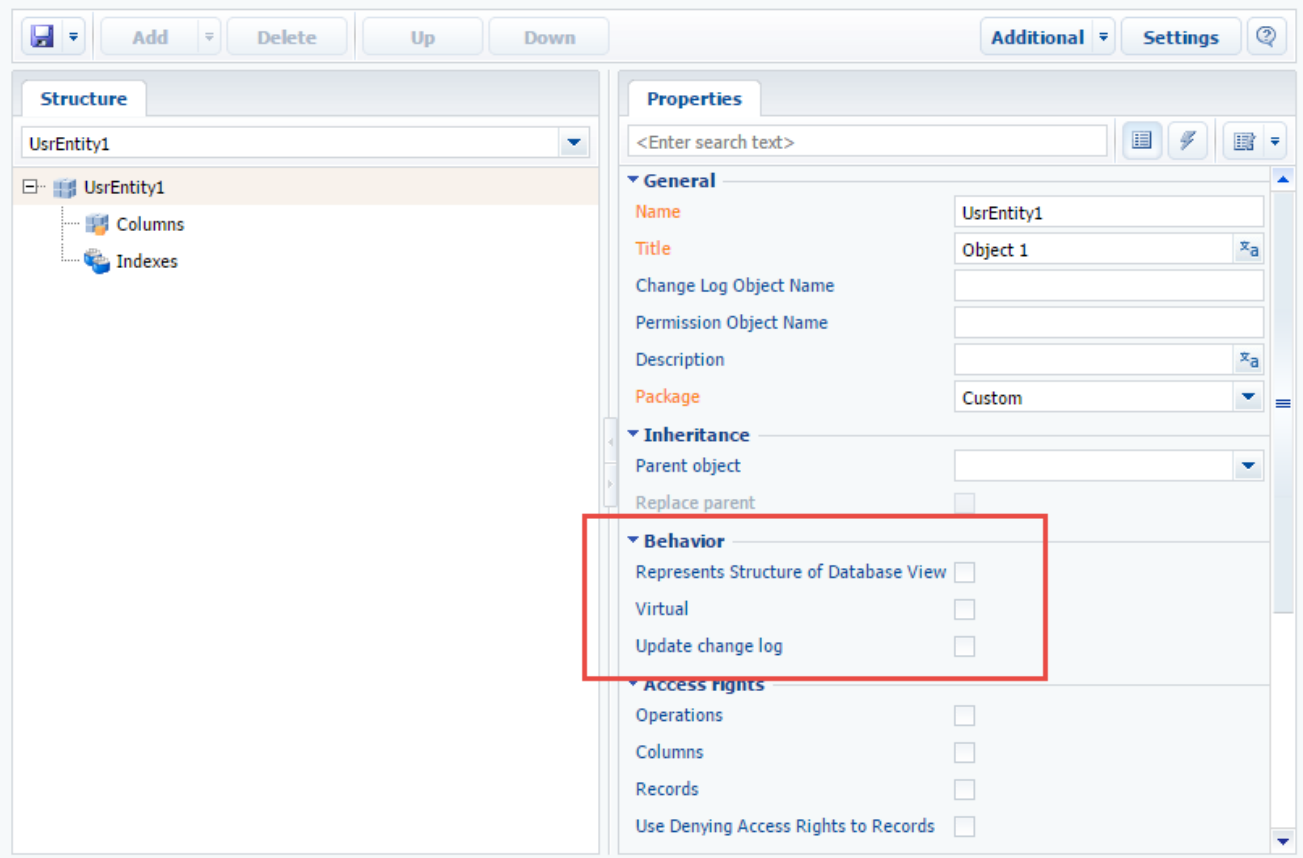
Custom objects are objects created as part of configurations saved in custom packages.

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.

More details on the bpm'online data model can be found in the "Platform description" section.

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

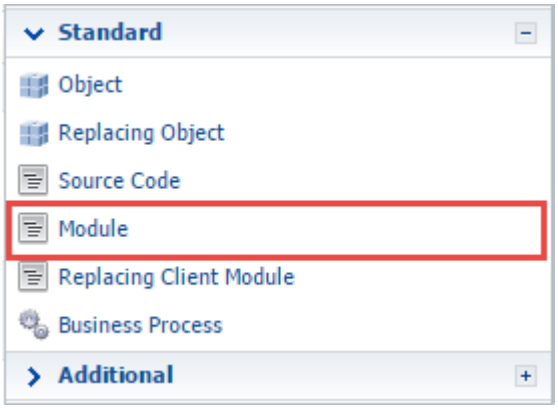
The following client module types are used in bpm'online:

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.

To create a non-visual module, go to the [Configuration] section and execute the menu command Add > Module (Fig. 3).

Fig. 3. Creating a non-visual module



2) View schema (visual module)

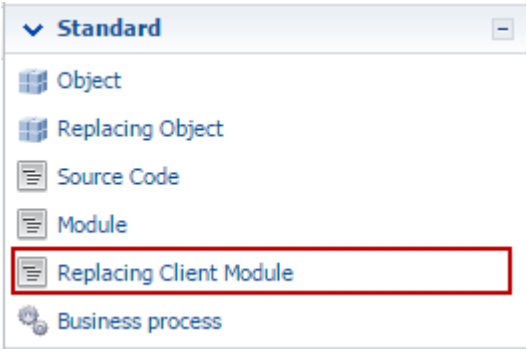
Visual modules implement the *View models* (ViewModel) according to MVVM template. These modules encapsulate data that is displayed in GUI control elements, as well as methods for working with them. Examples of visual modules are section, detail and page modules.

3) Extension module (replacing client module)

This type of module is designed for extending the functionality of base modules. The mechanism for working with replacing client modules will be covered in a separate article.

To create a replacing client module, in the [Configuration] section execute the menu commands Add > Replacing Client Module (Fig. 4).

Fig. 4. Creating a replacing client module



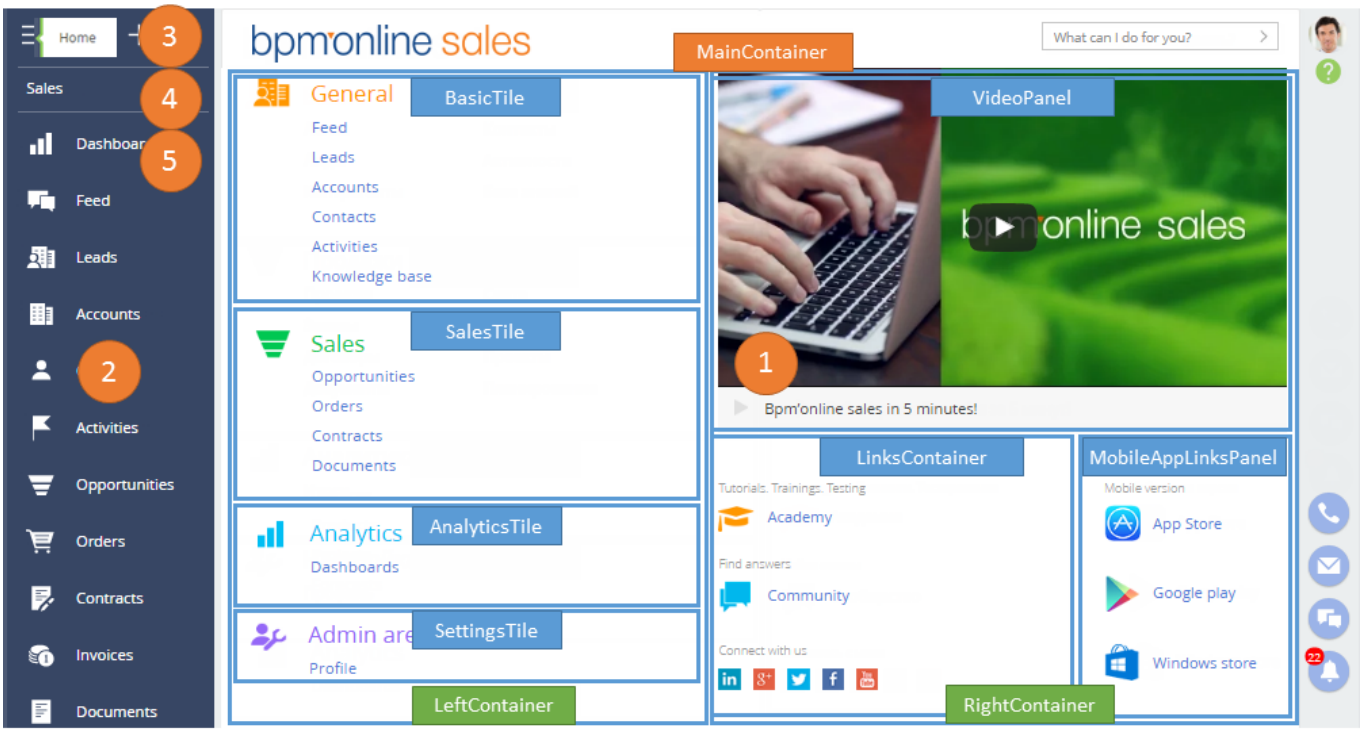
Application interface and structure

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

Main menu

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*.

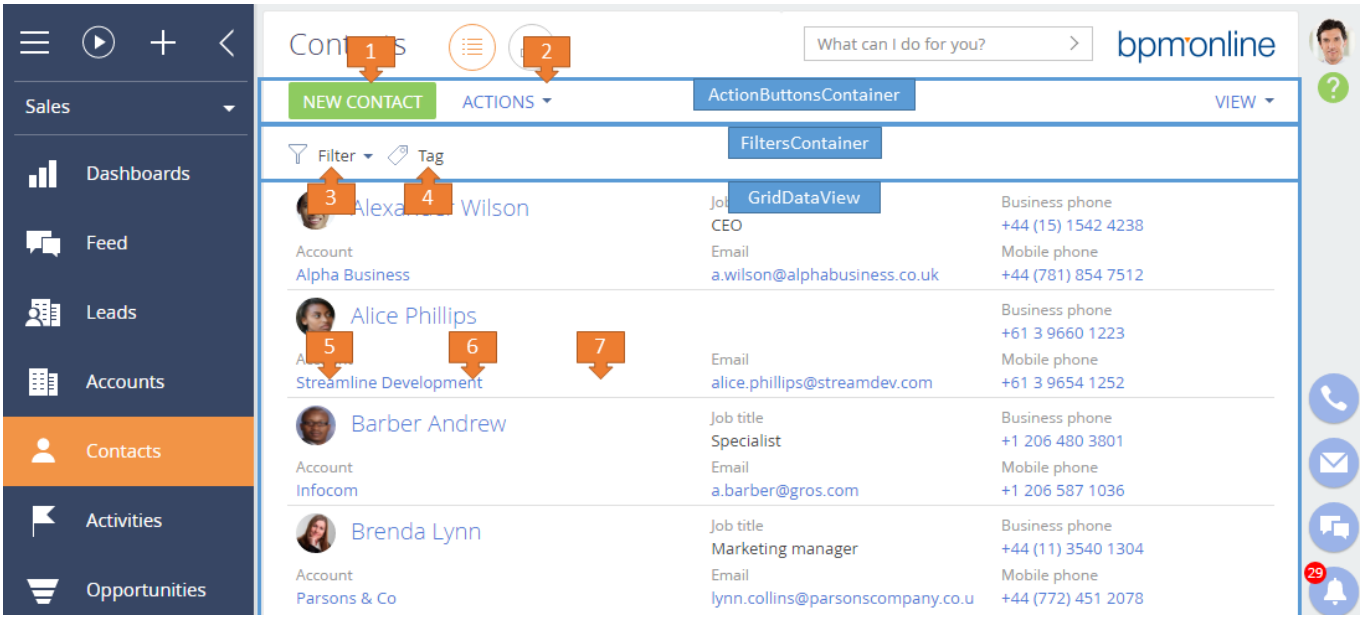
The element composition of the main menu UI depends on the product. All elements are placed in corresponding containers that are set up in the base or inherited schema of the main menu. The primary containers of the SalesEnterprise product include:

- **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

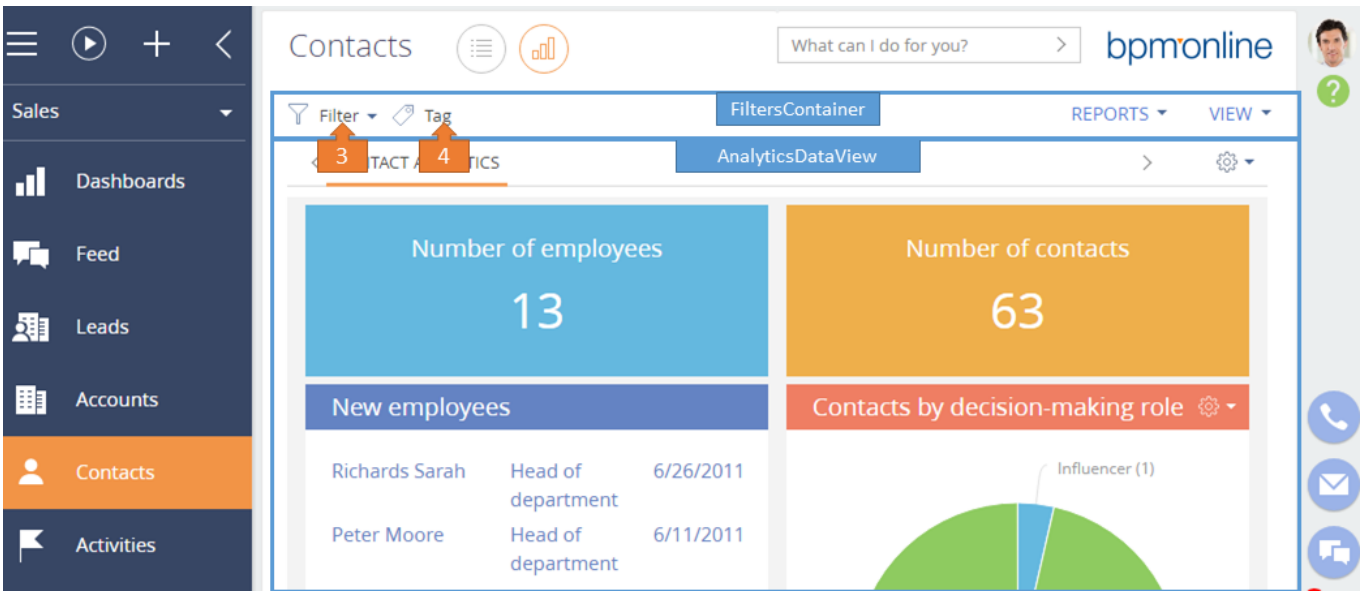
The "Section" element is displayed in the workspace of the user interface after selecting the appropriate menu item in the sidebar or on the main application page (Fig. 1).

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

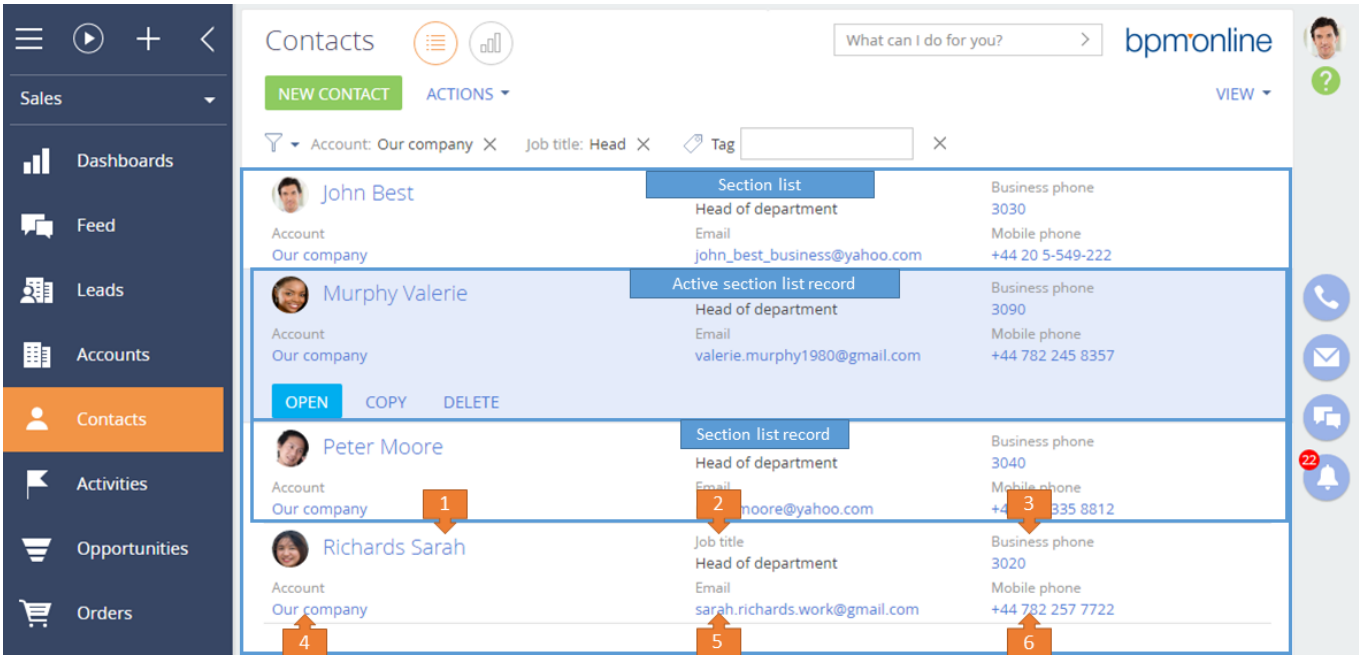
Section list is a list of records that can be displayed in one of two views.

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)

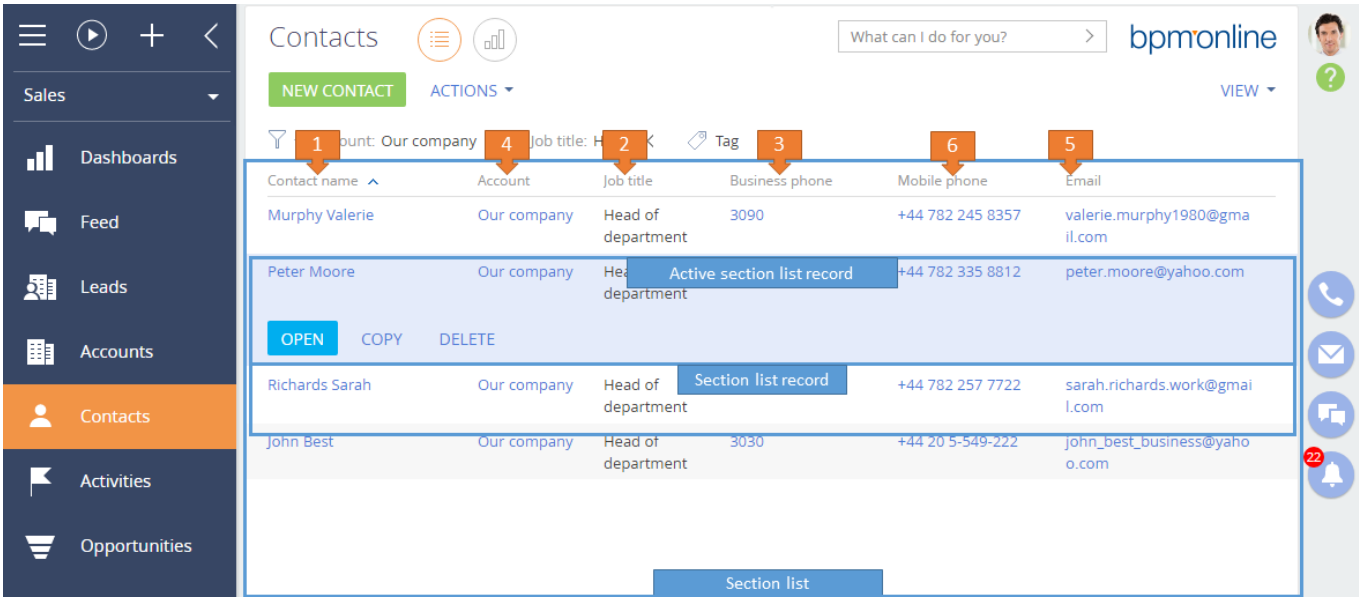
- Account (4)
- Email (5)
- Mobile phone (6)

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

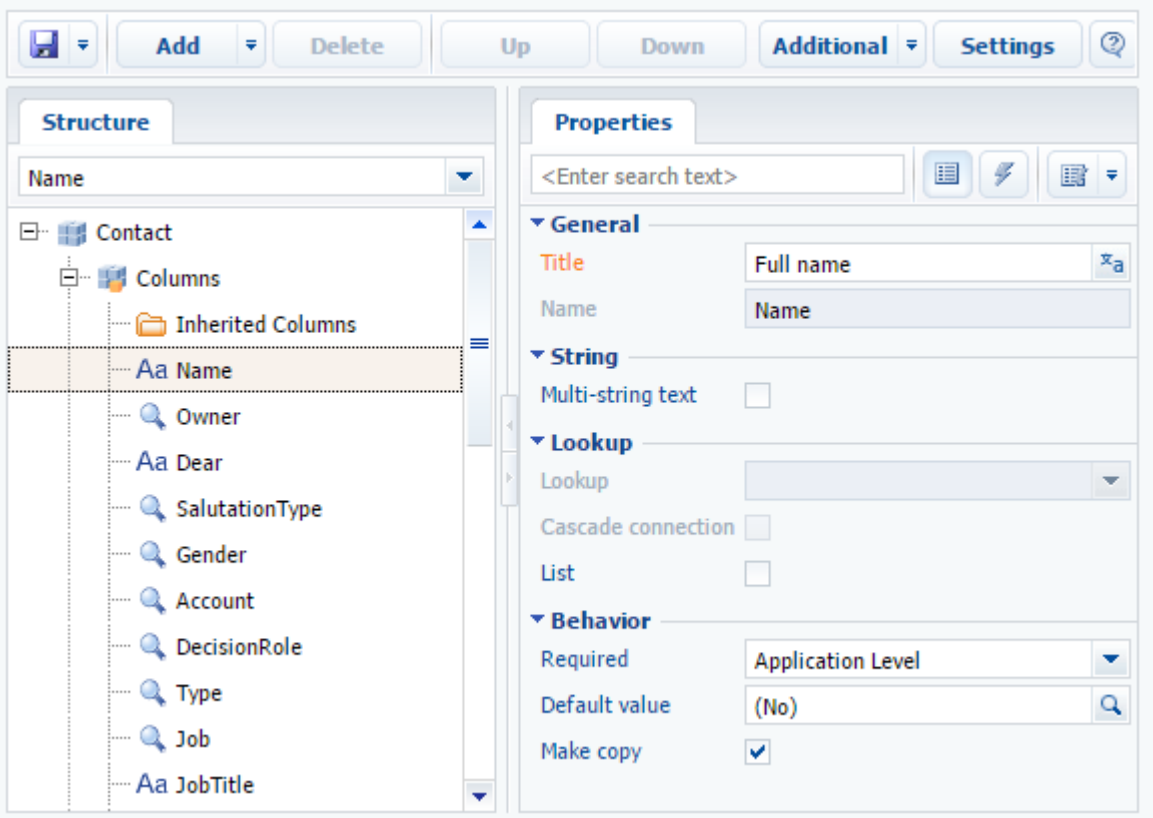


Fig. 4. Contact table

	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.

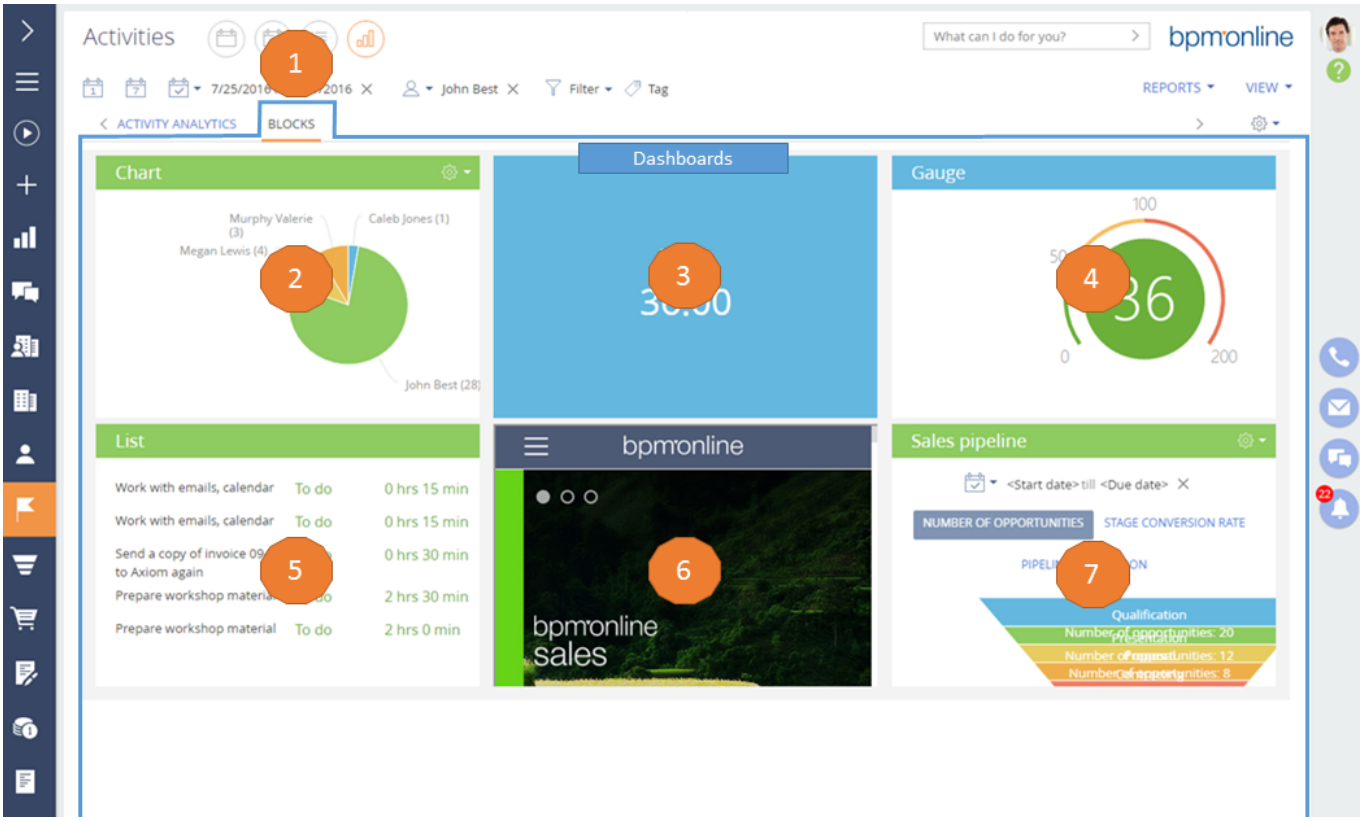
The first way is to use the section wizard. Create a replacing *Contact* object In the current custom package, which will inherit all the columns of the base *Contact* object from the *Base* package, to which a new custom column will be added. More information about the section wizard can be found in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

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.

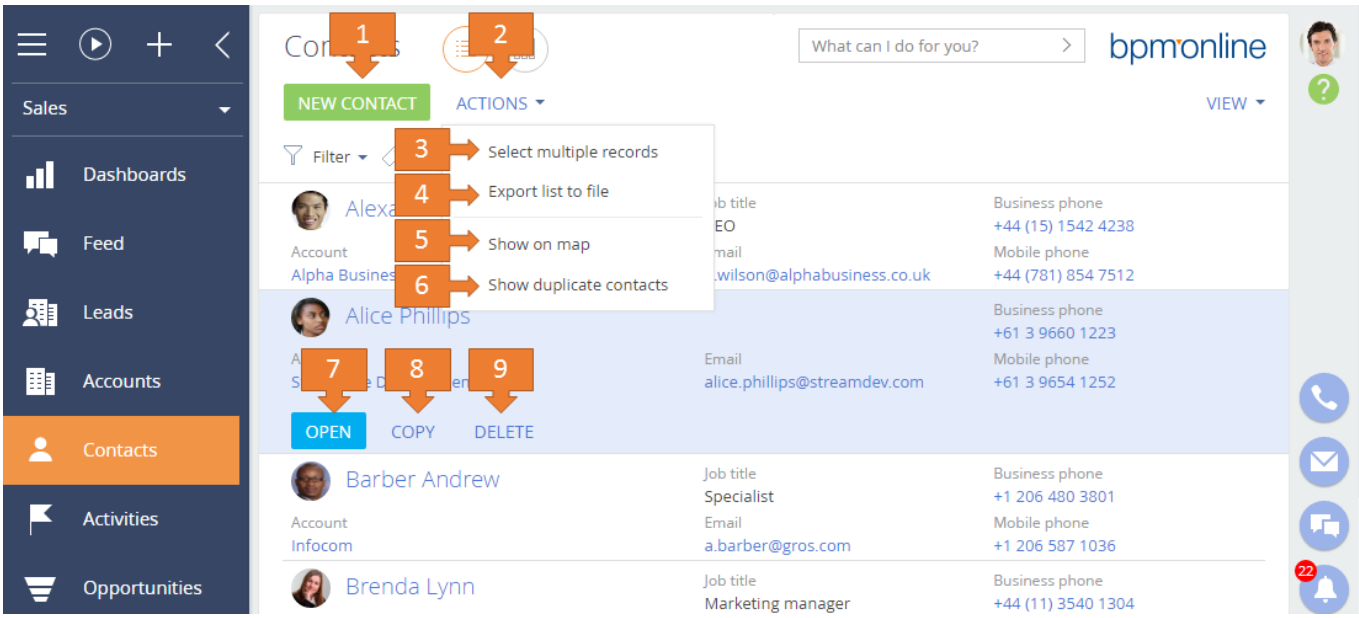
The widget dashboard component displays additional custom widgets.

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

Section actions

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".

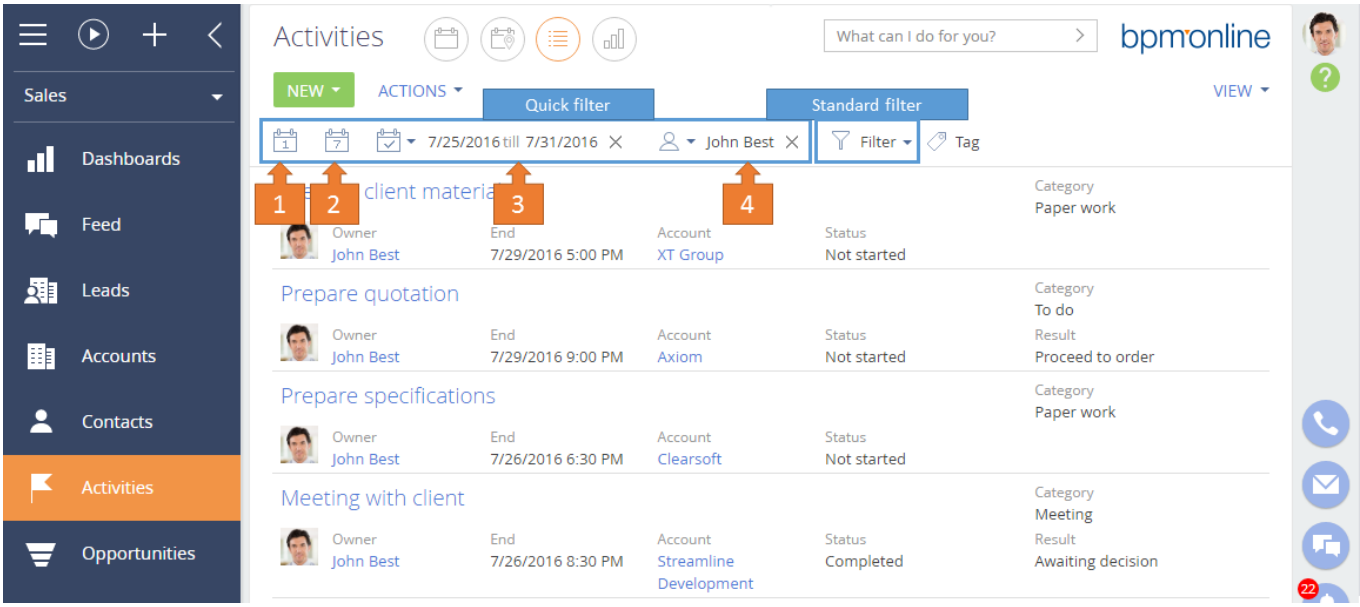
You can also create custom actions in bpm'online. Learn how to add custom actions in the "**Adding an action to the list**" article.

Filters

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



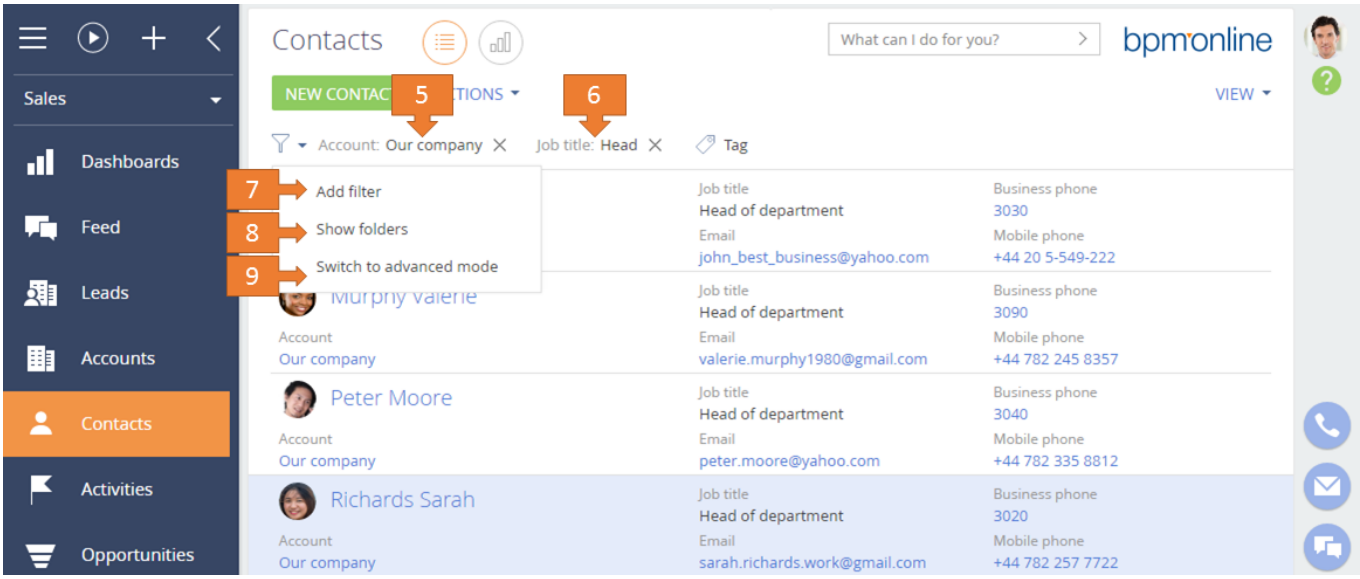
Quick filter is used to filter data based on most frequently used parameters. For example, activities of a single employee for a specified period of time are most often viewed in the [Activities] section. The following quick filters are designed exactly for this purpose (Pic 1):

- **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

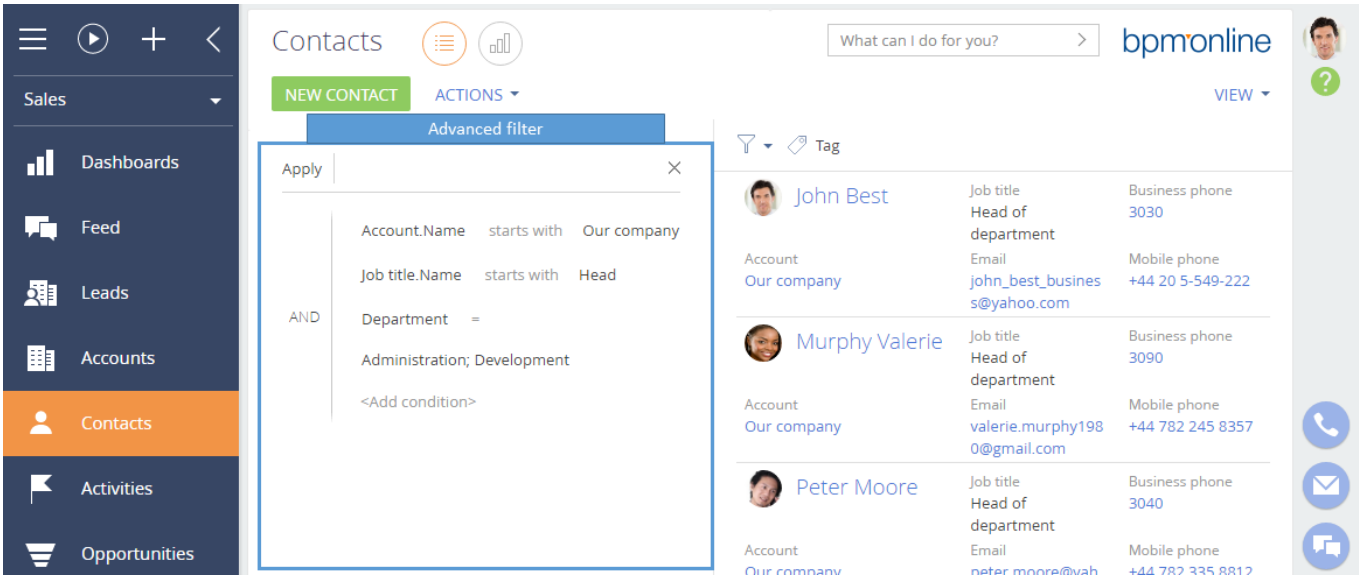


You can set up standard filters by running the [Set condition] (7) command in the "Filter" menu. More detailed

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



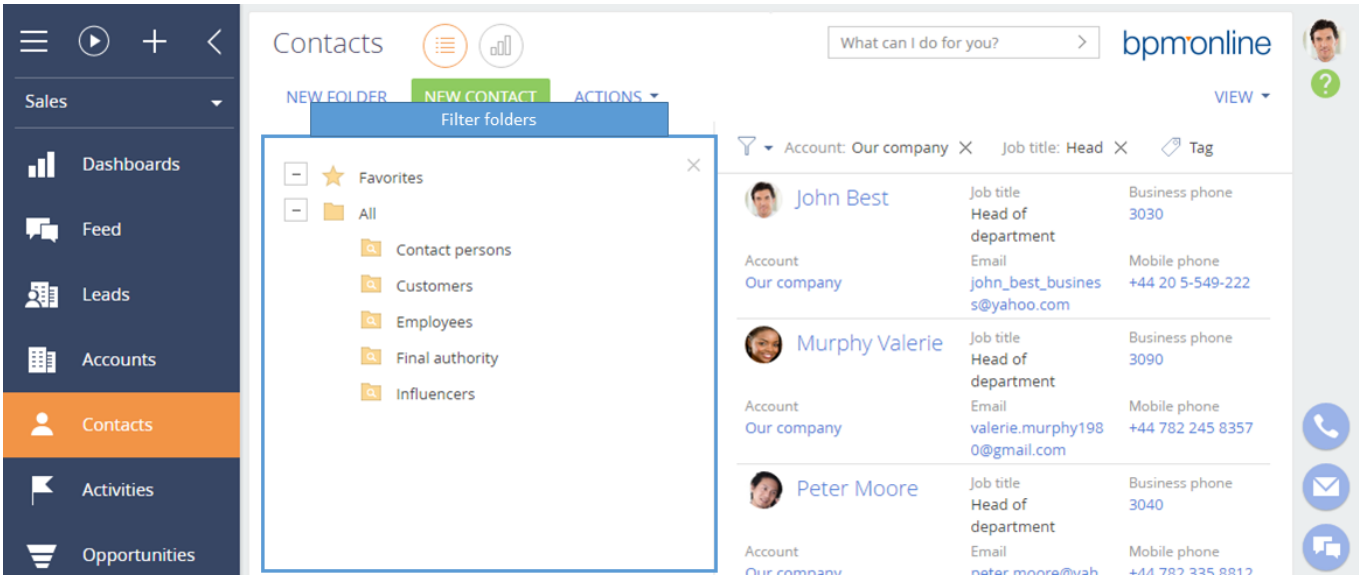
To set up the advanced filter, you must run the [Switch to extended mode] command (8, Fig. 2). More detailed information about advanced filter setup is available in the "Advanced filter" article.

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



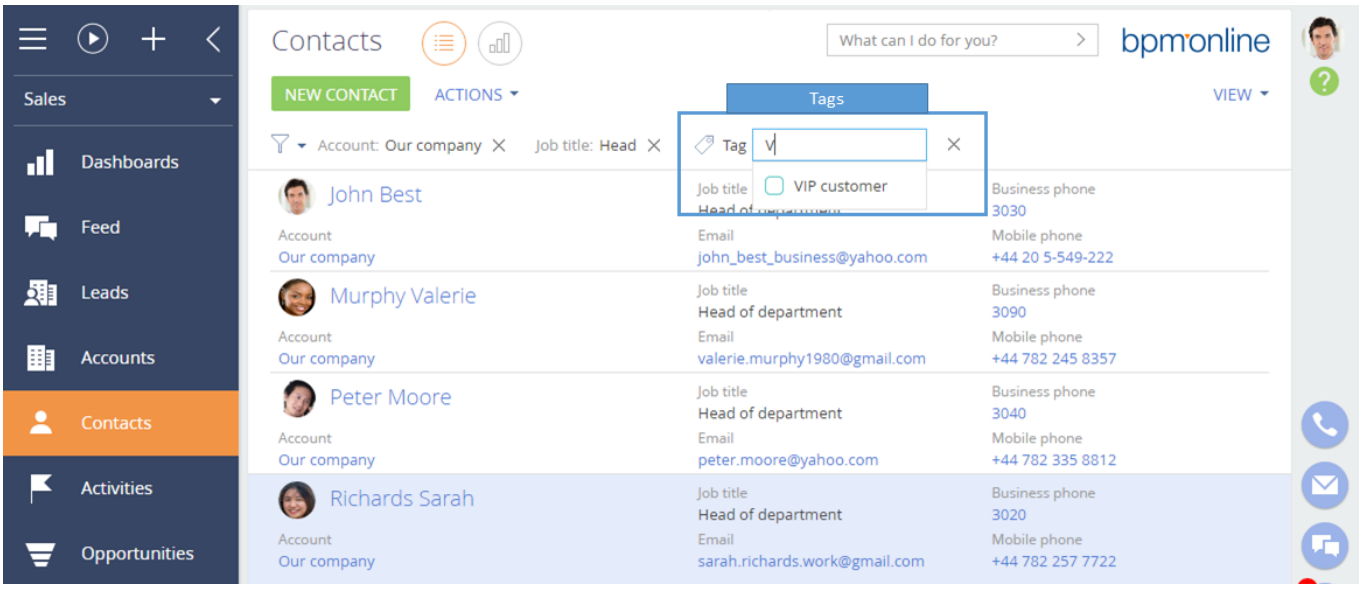
More detailed information about filter folders is available in the "Working with folders" article.

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

Tags

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:

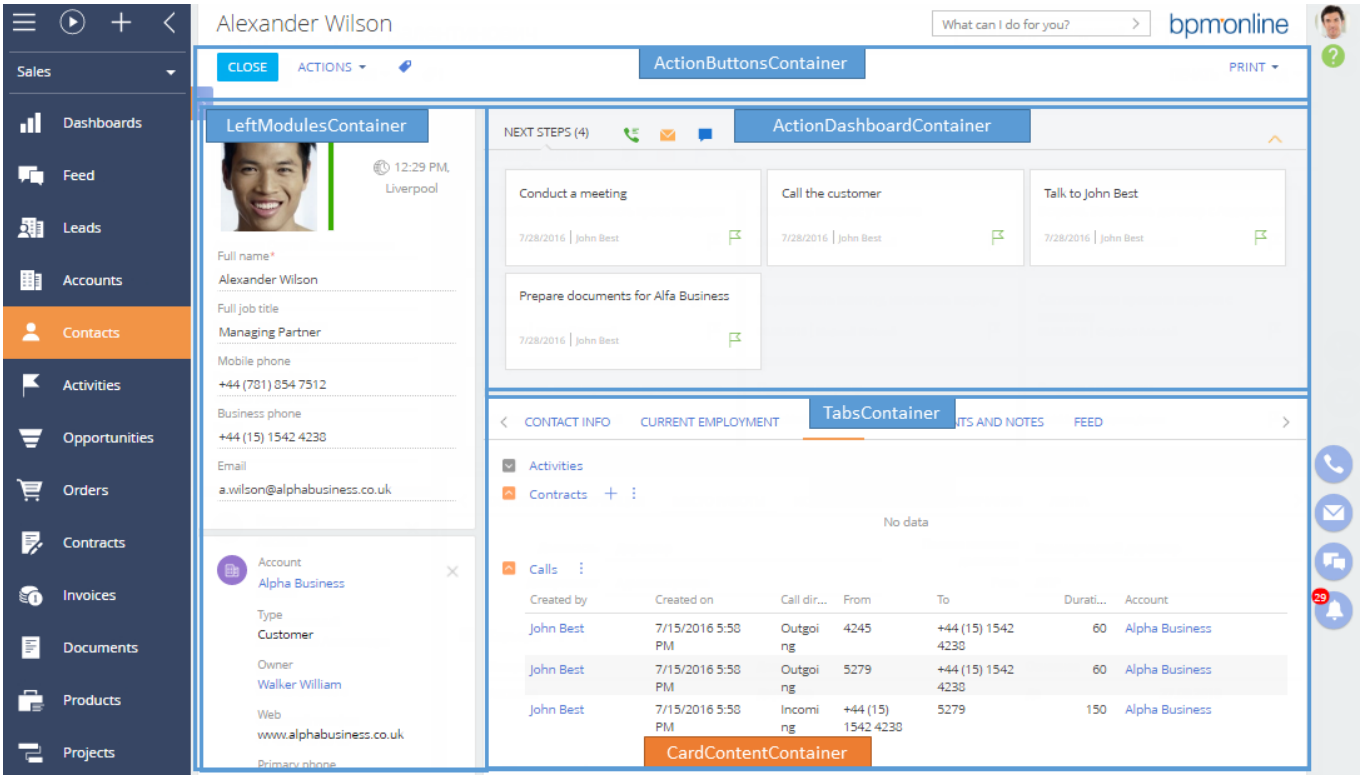
- **Personal** tags — can be seen and used only by users who created them. Neither system administrators nor managers can see the personal tags of employees. Personal tags are displayed in green.
- **Corporate** tags — displayed for all employees of the company. Any employee can set or clear a corporate 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

p>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.

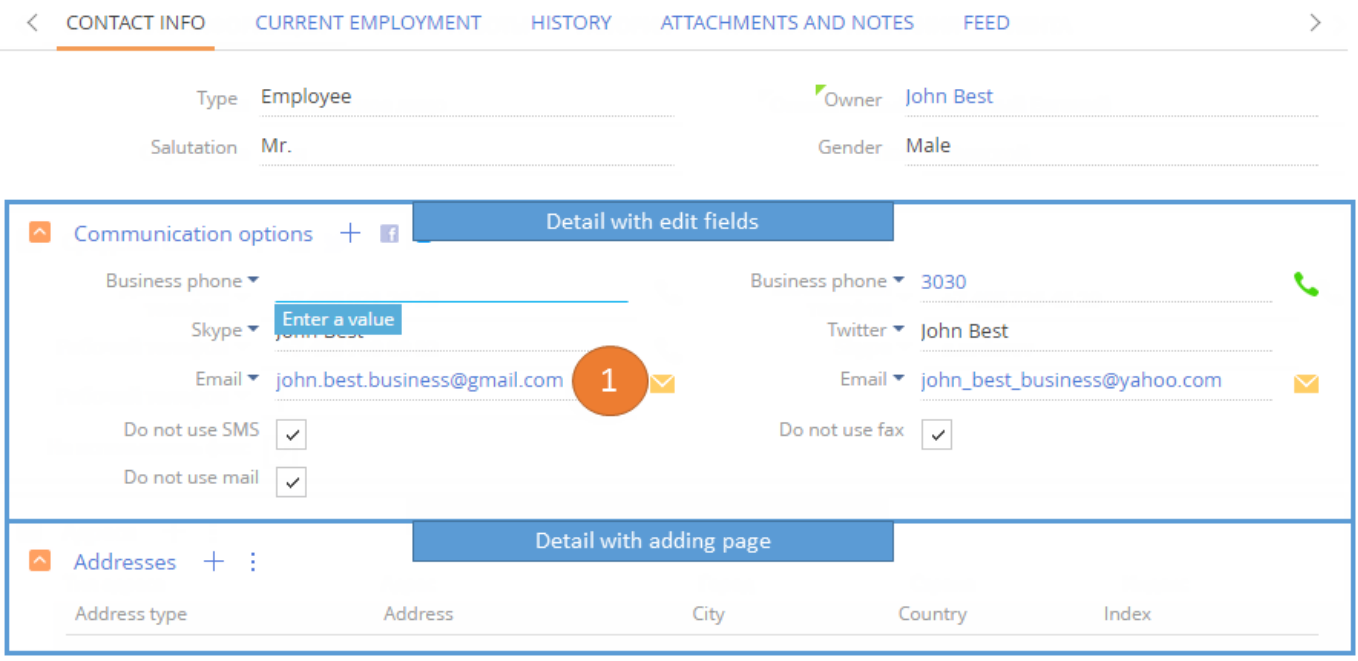
Details

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.

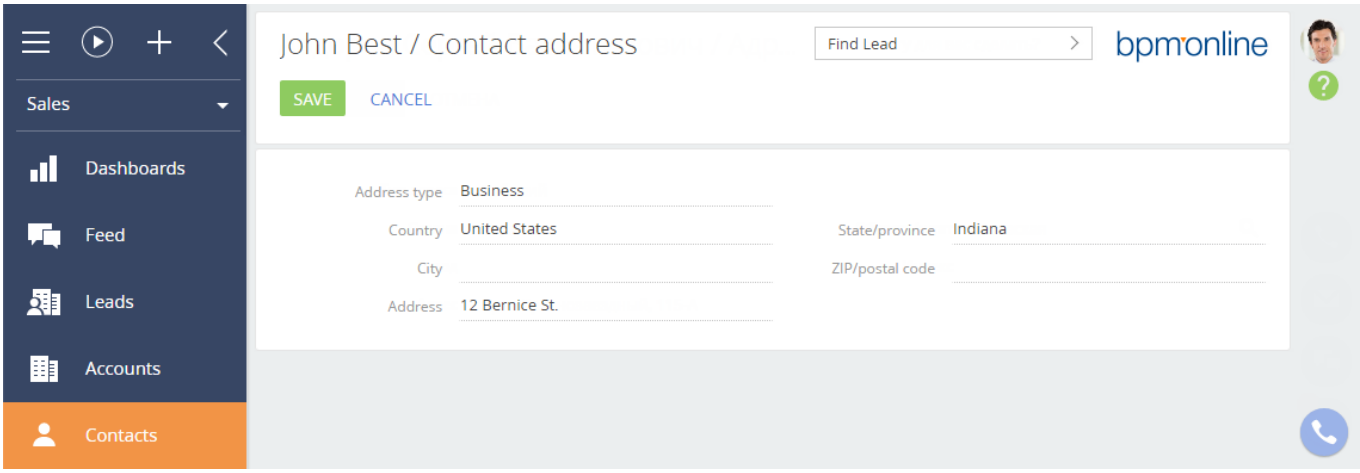
A *detail with edit fields* — data are filled in and edited in the detail data fields (Fig. 1). If necessary, you can add a new field to a detail (1). For example, the [Contact communication options] detail.

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



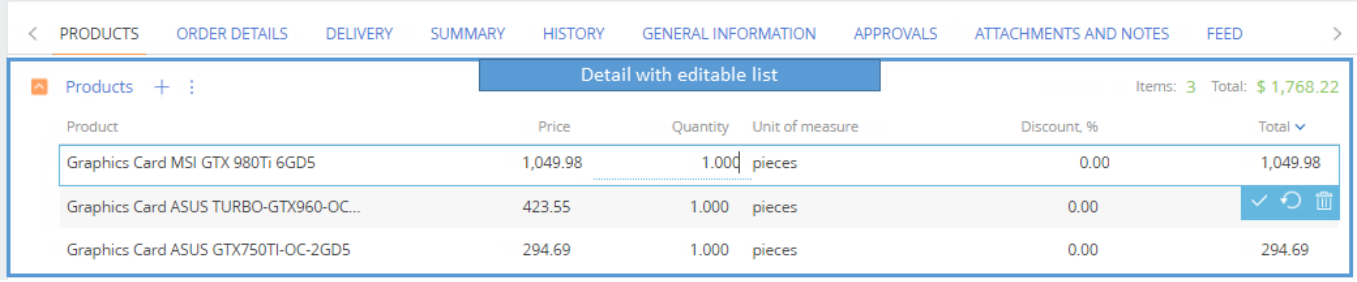
A *detail with adding page* — data are entered and edited on the detail edit page. For example, the [Contact address] detail (Fig. 1) — each address is entered and edited on the "Contact address" page (Fig. 2).

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

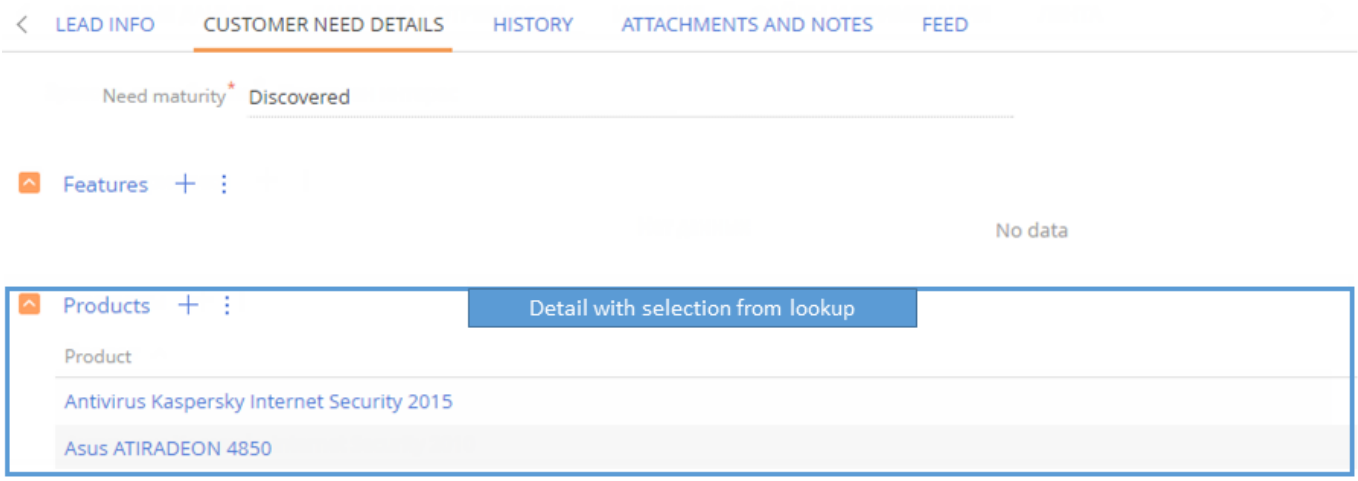
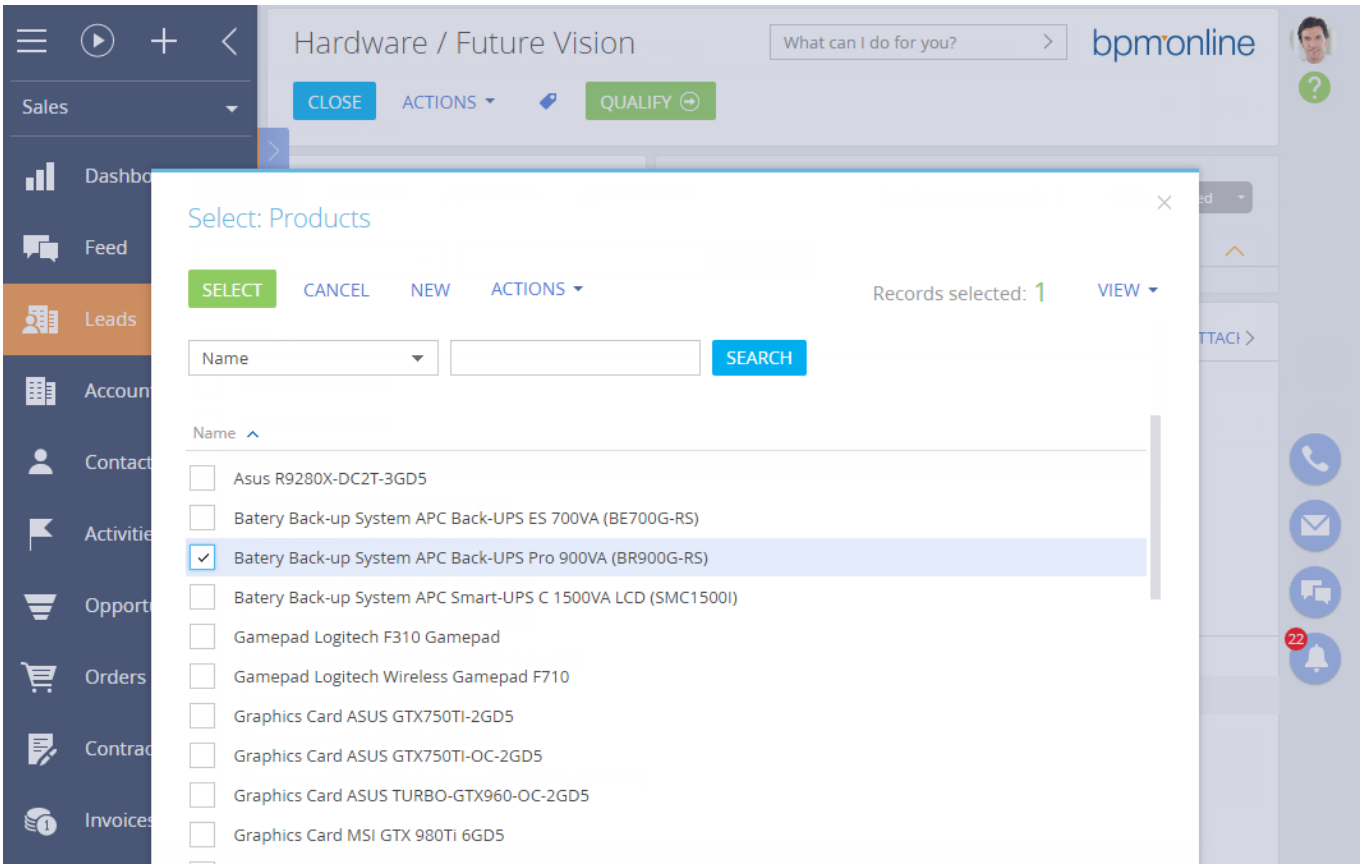


Fig. 5. Selecting products from the [Lead product] detail



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 "Detail with edit fields", "Detail with adding page", "Detail with editable list", "Detail with

selection from lookup" articles.

Mini-page

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

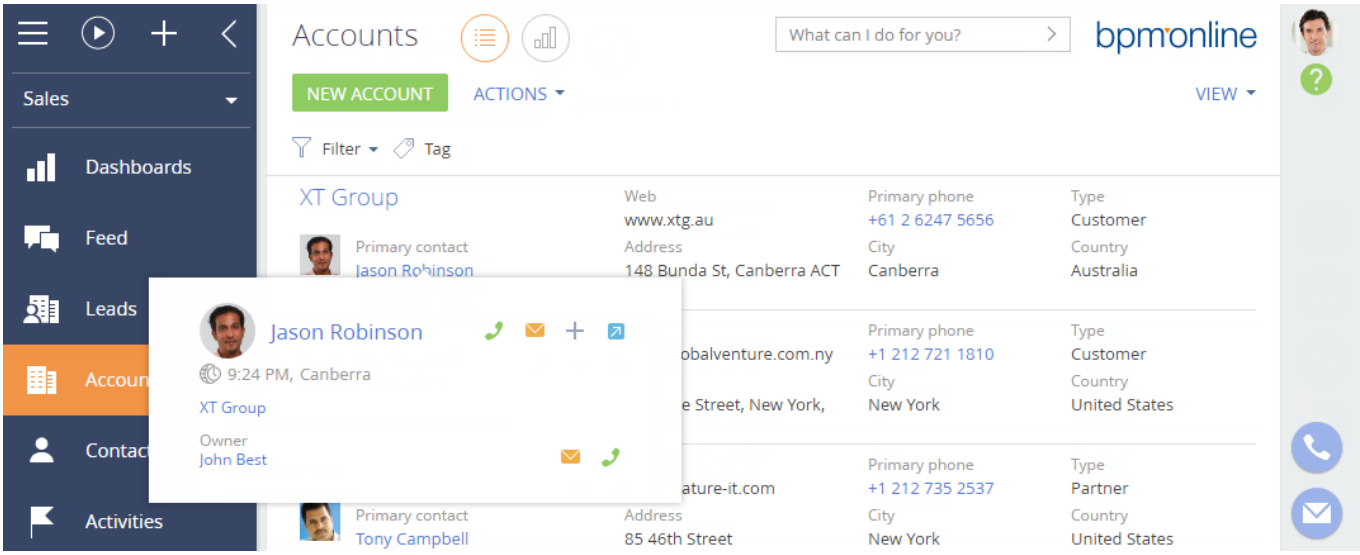
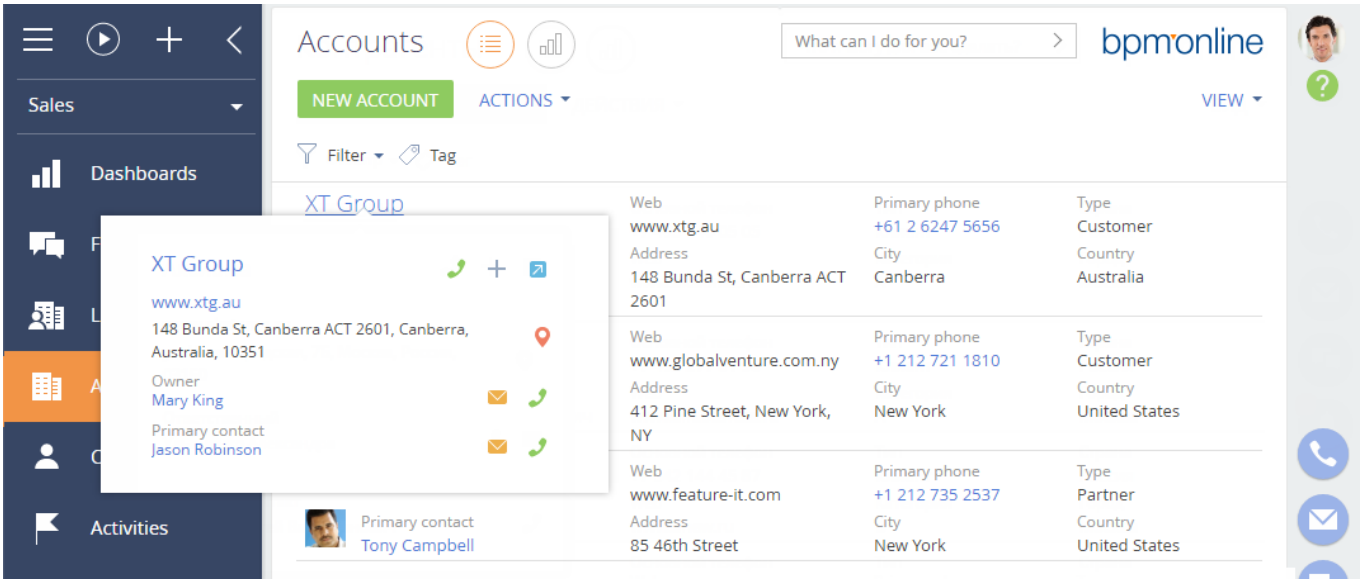


Fig. 2 Account mini-page



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.

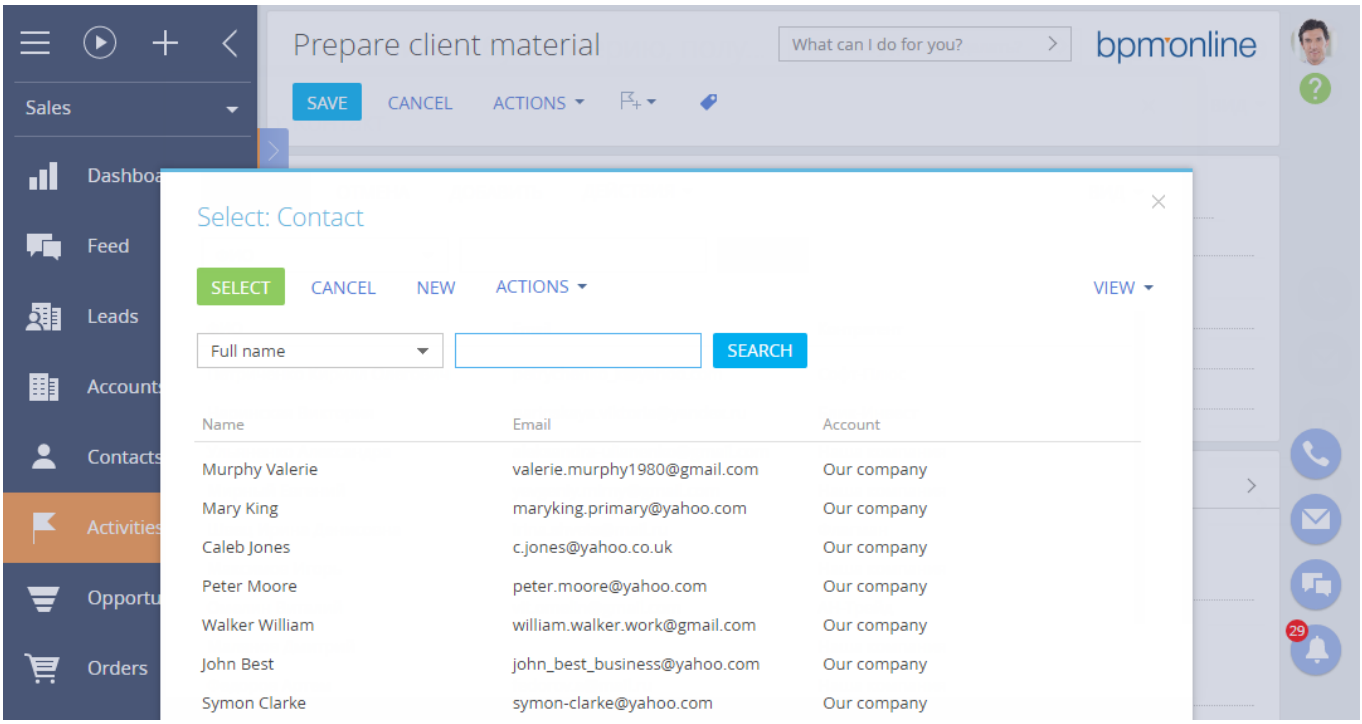
If necessary, you can create a custom mini-page. An example of creating a custom mini-page for records in the [Knowledge Base] section is described in the "**How to create a mini-page**" article.

Modal windows

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.

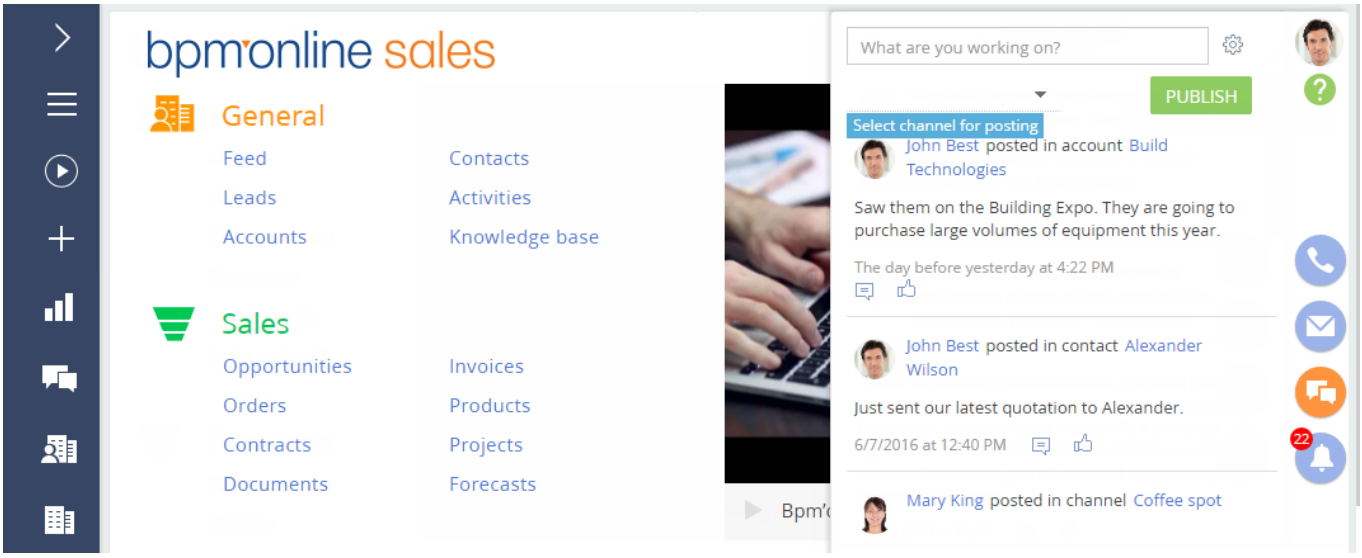
Communication panel

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



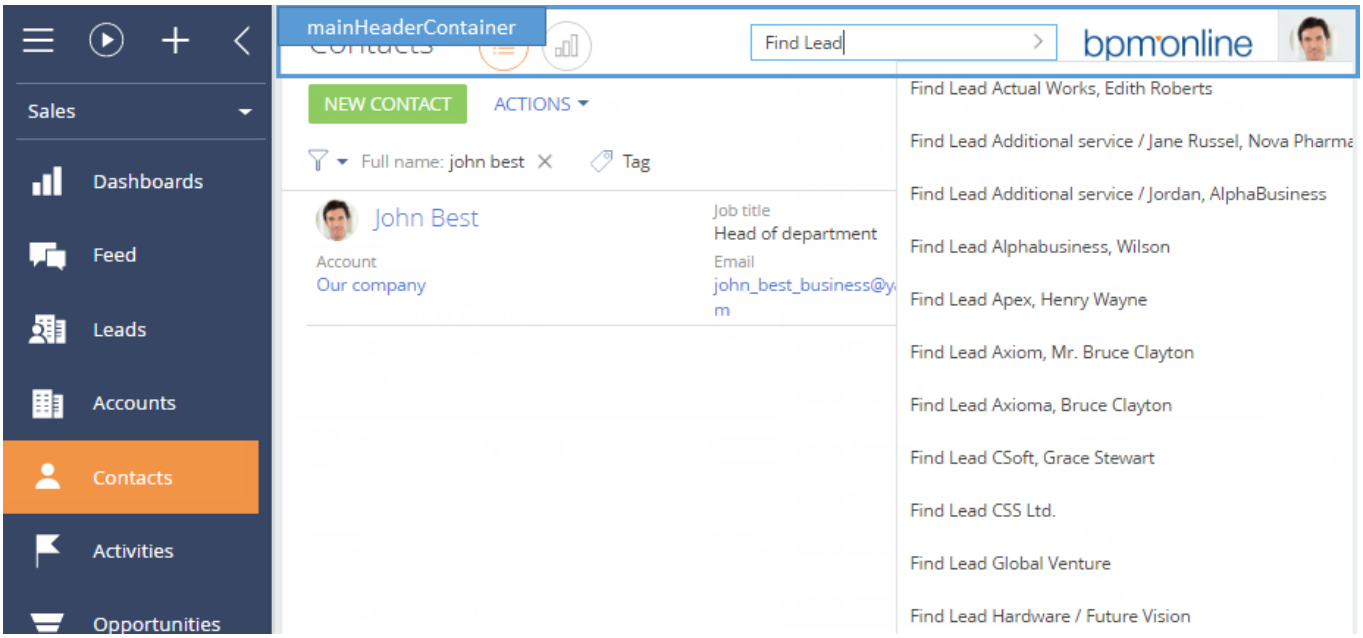
The communication panel is configured in the *CommunicationPanel* scheme of the *UIv2* package. The "Calls" tab is configured in the inherited *CommunicationPanel* scheme of the *CTIBase* package . The communication panel tab buttons (1...4) are located in the *communicationPanel* container, and the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

Command line

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).

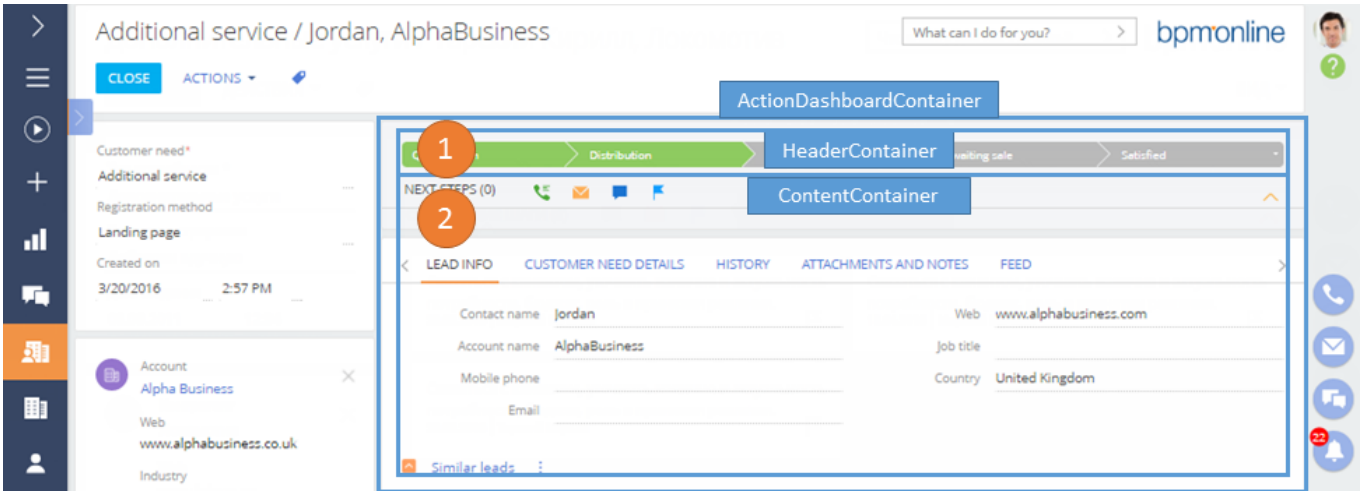
To track commands and their execution in the system, use the *CommandLineService* service. To store commands in the system, a database table is used. The structure of database table is described by the *Command* object schema. The command parameters are described by using the *CommandParams* object schema.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

Action dashboard

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.

Development tools

- **How to start development**
- **Development resources**

How to start development

- **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 user and setup workspaces**
- **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

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.

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.

The testing (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.

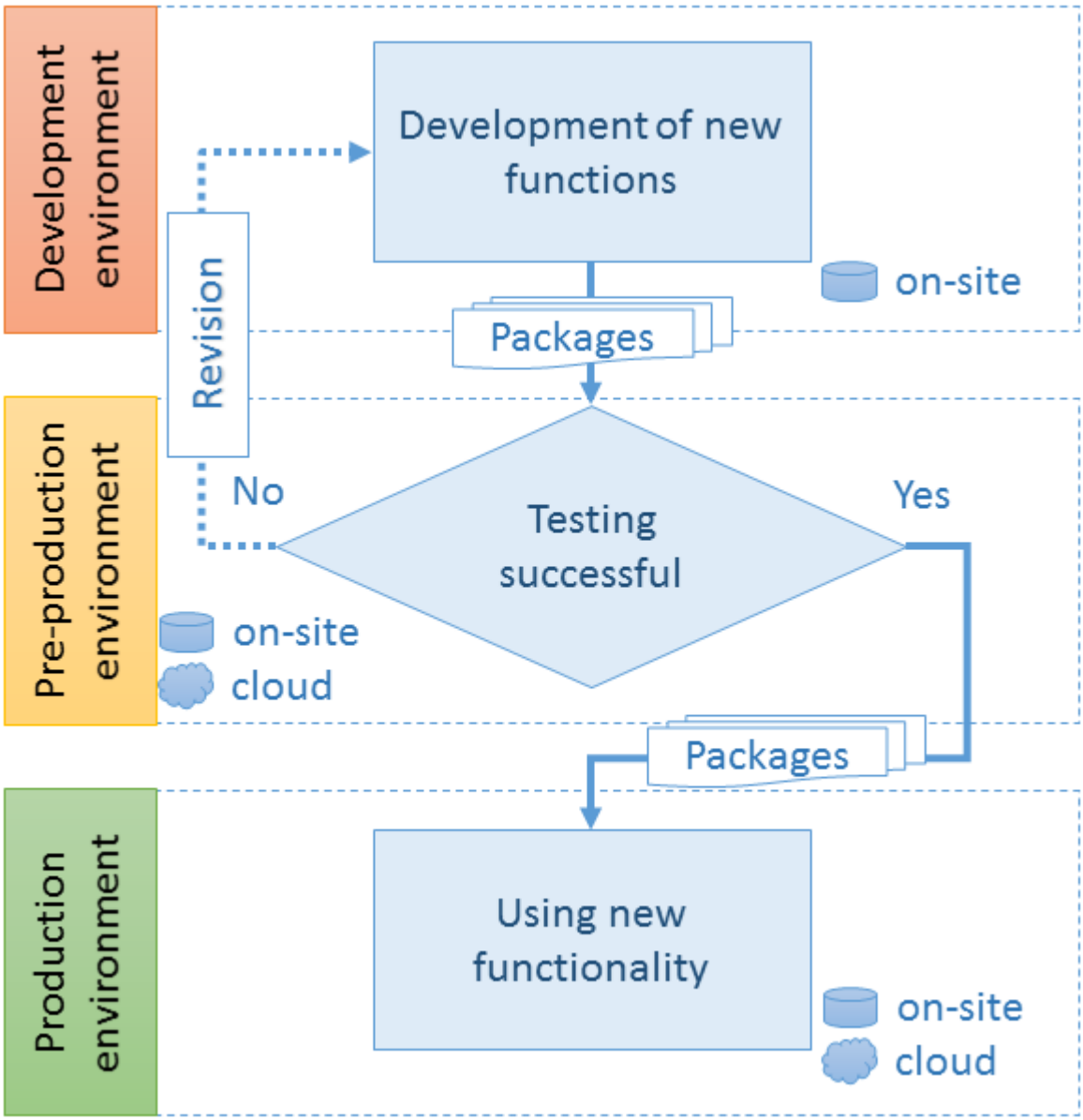
For more information about deployment options, please refer to the "**How to deploy bpm'online on-site**" and "**Deploying the bpm'online "cloud" application**".

> ⚠ ATTENTION
>
> The production database must never be used in development or pre-production environments. 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 a specially prepared 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 the WorkspaceConsole utility to transfer packages between applications. For more information on the WorkspaceConsole, please refer to a separate article . For more information about transferring changes between applications, please refer to the "Transferring changes on the configuration, database and package levels" article.

If you need to transfer packages with schemas that between different development environments, it is recommended to use SVN version control. For more information on working with SVN version control, please refer to the "**Create repository in SVN server**" article.

Organizing a development environment

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 testing, 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.

There are two general ways to organize a development environment:

- Develop several configurations in a single application. The development environment is based around a single deployed application with a number of configurations that share the same database.
- Develop several separate applications. The development environment is based around several applications with separate databases.

Regardless of how the development environment is organized, the applications can be deployed either locally (*on-site*), or on bpm'online servers (*cloud*), with or without the use of SVN repository. 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**" article.

Working with several configurations of a single application

ATTENTION

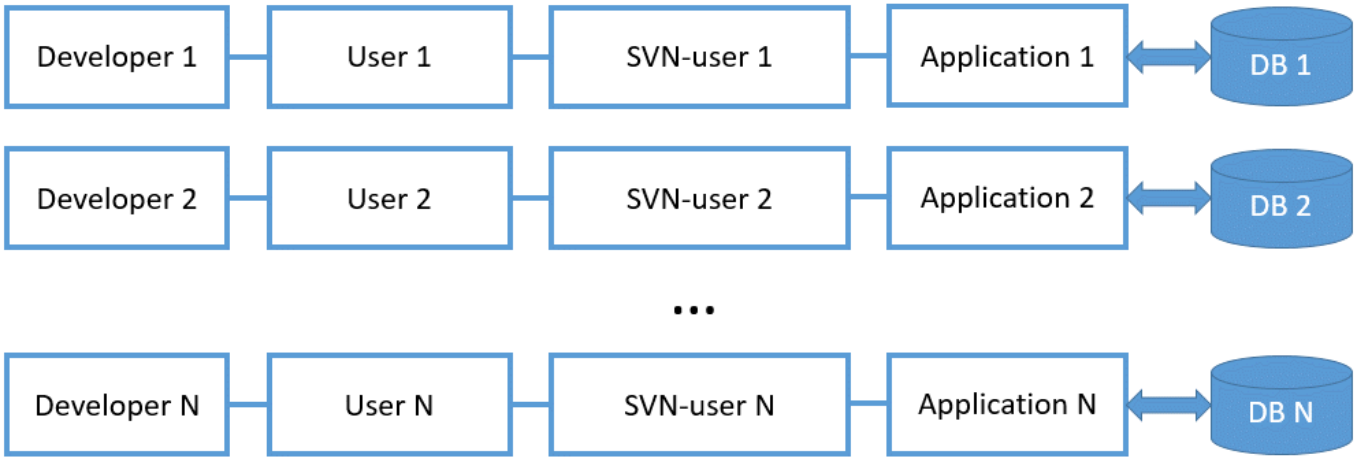
It is not recommended to perform development in several configurations of a single application. Creation of configurations and development in the production environment is strictly forbidden.

1. .

Development in separate applications

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.

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

Development RulesIntroduction

Development of complex functionality requires proper organization of the development process. Recommendations on the development process organization are covered in the **Development process organization** article. The article also contains the general development algorithm. Options for development environment setup are available in the **Organizing a development environment** article.

There are three general options for development environment deployment:

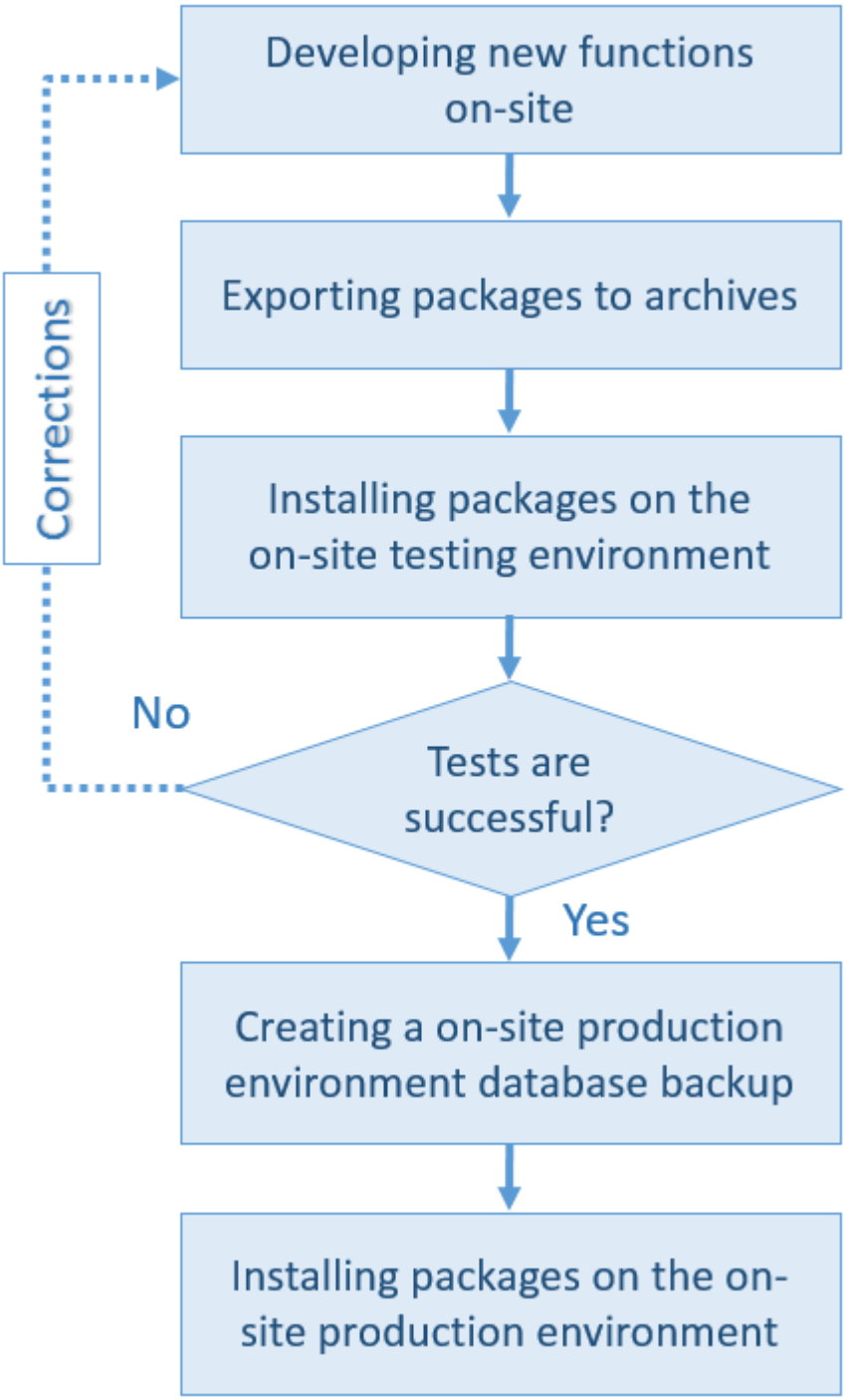
- All instances are deployed on-site.
- Development environment is deployed on-site, test and production environments are deployed on the cloud.
- All instances are deployed on the cloud.

The general procedure for developing new applications is similar, regardless of the deployment option. There are several notable differences in specifics.

All applications are deployed on-site

The general development algorithm with on-site deployment is available on Fig. 1.

Fig. 1. Development sequence for on-site deployment



All steps are performed by the dev-teams. Recommended sequence:

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 (SVN) 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 with the WorkspaceConsole

Use the *WorkspaceConsole* utility to export archived packages from the database or SVN repository to the file system. Exporting packages with the *WorkspaceConsole* utility is covered in the **Transferring changes using WorkspaceConsole**article.

> ATTENTION
>
> Transfer of changes to the test and production environments with the *WorkspaceConsole* is recommended, since all application updates are supplied to the users in the form of packages. Any errors with the new changes can be traced and resolved on the stage of loading packages to the system.

3. Loading packages to the testing environment

Use the *WorkspaceConsole* to install the packages from the file system. Installing packages with the *WorkspaceConsole* utility is covered in the **Transferring changes using WorkspaceConsole**article.

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.

5. Installing the packages on the production environment

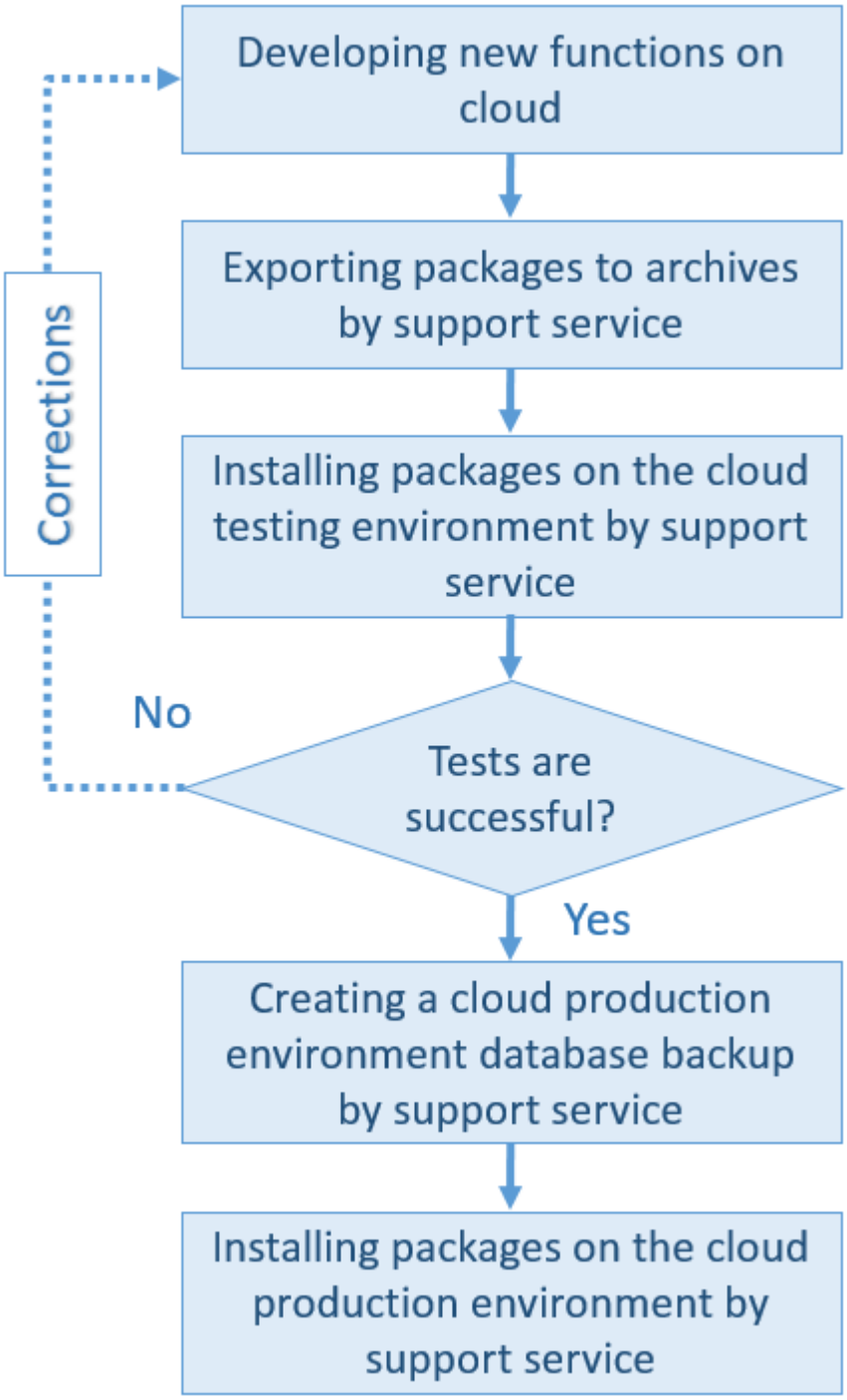
As with the testing environment (see step 3), installing the packages on the production environment is also done with the *WorkspaceConsole*.

All applications are deployed on the cloud

If the development and production environments are deployed on the bpm'online cloud, developers do not have direct access to application setup via the configuration files. Also, there is no direct access to the database and IIS. In this case all operations with the configuration files, database and IIS are performed through the bpm'online support. The application is updated by the support service as well.

Due to these limitations, most of the development process steps that include transferring changes between the working environments are performed by bpm'online support (Fig. 2).

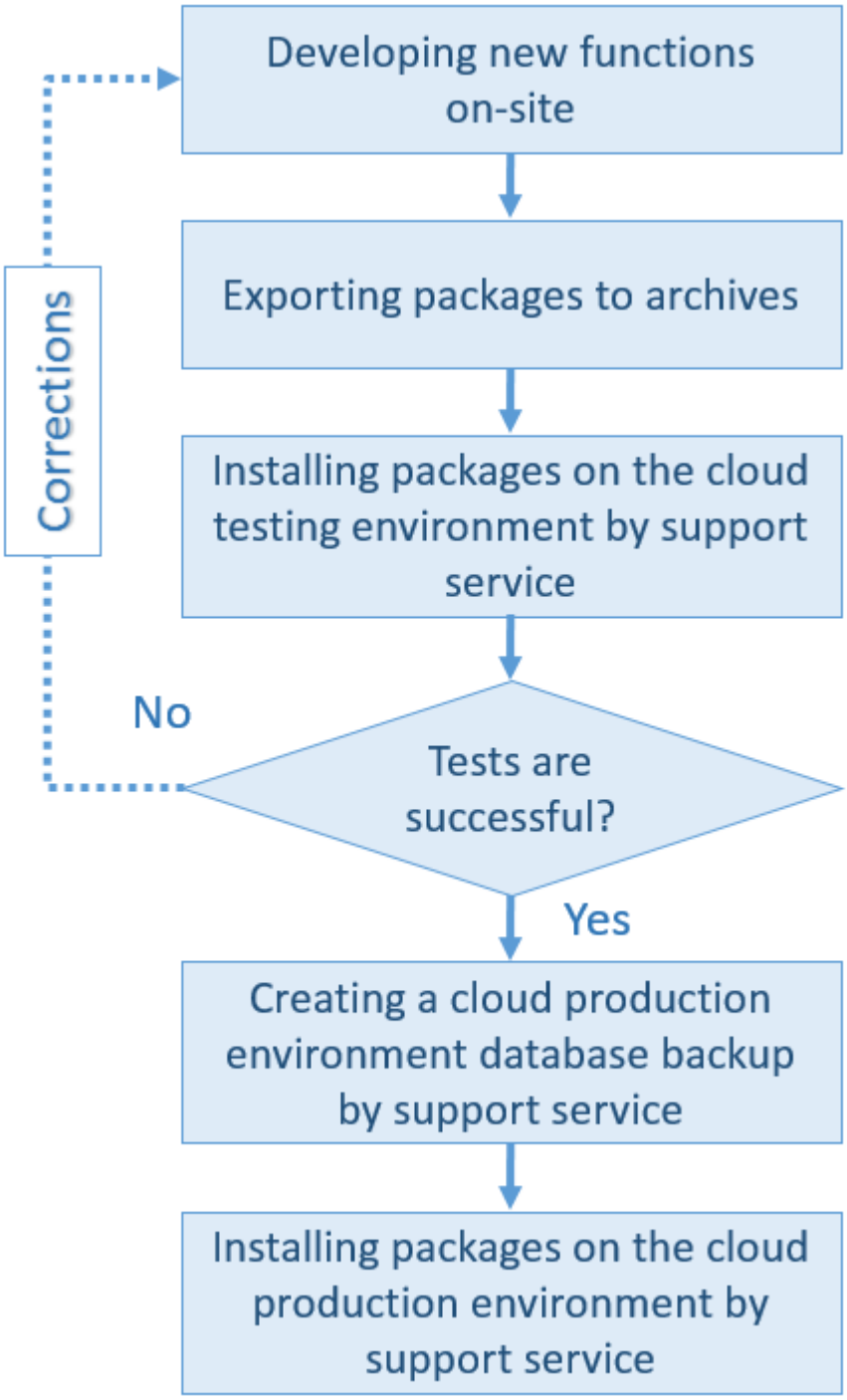
Fig. 2. Development sequence for cloud deployment



The environments are deployed both on-site and cloud

If the development environment is deployed on-site, and the testing and production environments are deployed on the cloud, some steps are performed by the developers and some are done by bpm'online support (Fig. 3).

Fig. 3. Development sequence for on-site and cloud deployment



Development rules

Introduction

During the creation of new functionality, bpm'online developers and partners have compiled a set of rules and recommendations.

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 JavaScript frameworks 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 **testing 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 expanding and replacing modules and schemas

If you want to create a replacing 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.

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

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.

To install bpm'online on DBMS, MS SQL, you need to perform the following steps:

1. Install and/or enable required Windows components on the application server.
2. Install MS SQL Server on the database server.
3. Install Redis database on the user session storage server.
4. Install SVN server on the package repository computer.
5. Install the bpm'online application on the application server.

> ATTENTION
>
> To ensure compatibility with cloud infrastructure of the bpm'online platform, the application database should be created in SQL Server 2016.

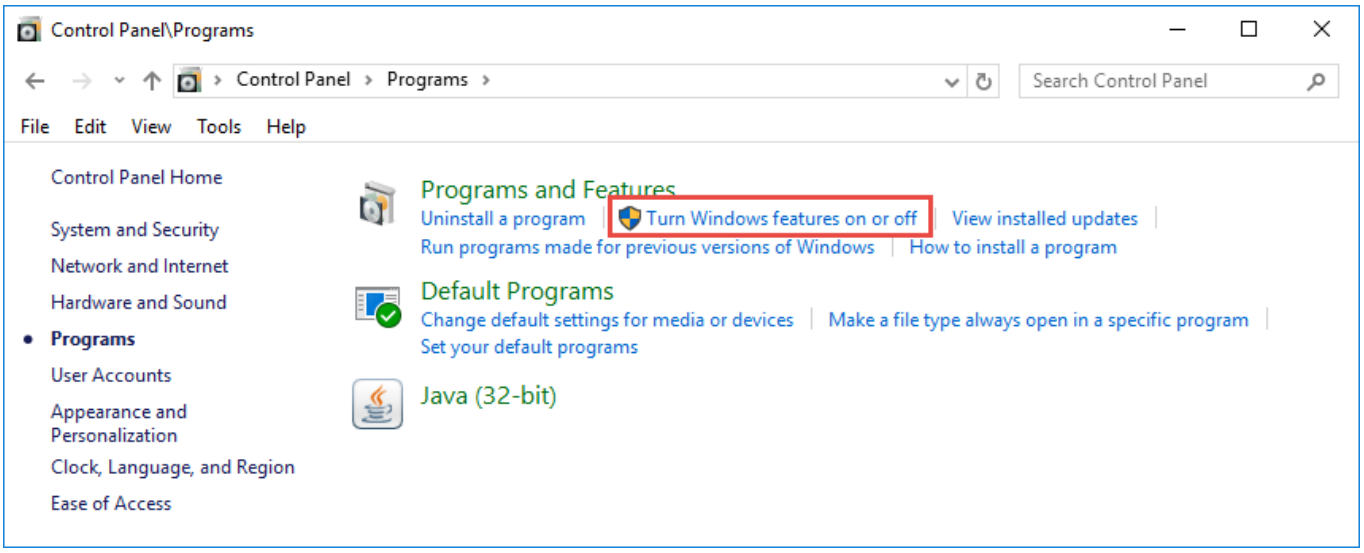
> NOTE
>
> Installation procedure of MS SQL Server or SVN is covered in the documentation of each product.

Installing and configuration of required Windows components

> ATTENTION
>
> This article lists the components that should be enabled in Windows 8 and Windows 10 operating systems.

To check the availability of the required components, select [Turn Windows features on or off] in the [Control Panel] section of the [Programs and Features] section (Fig. 1).

Fig. 1. Configuration Command of the Windows Control Panel Components



Check the [Windows Features] window for the components listed in the Table 1 and select missing components for installation (if need).

Fig. 2. [Windows Features] window

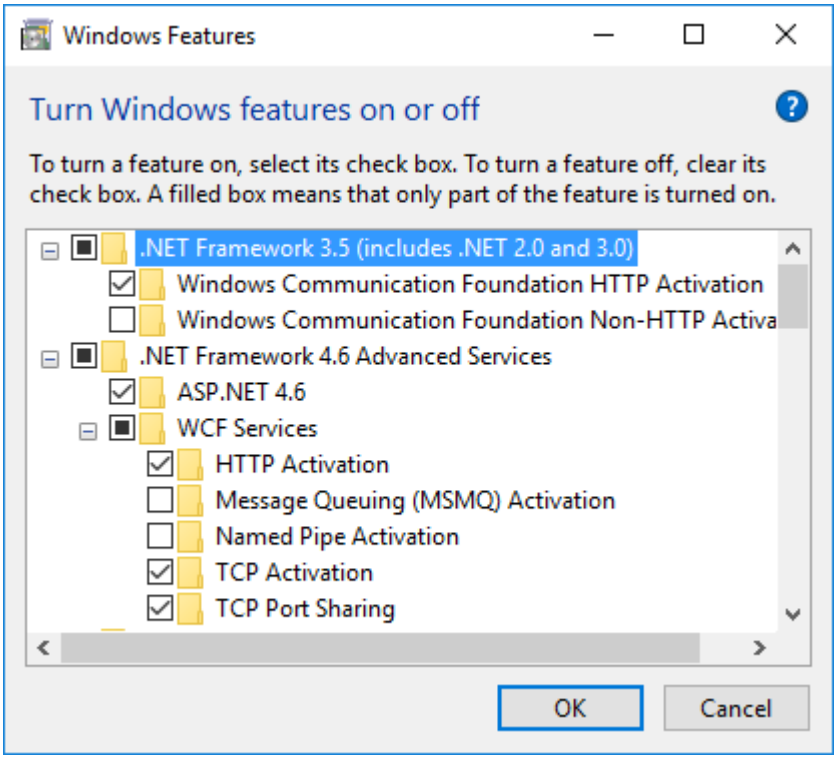


Table 1. Required winrows components

Component	Component items
.NET Framework 3.5 (includes .NET 2.0 and 3.0	Windows Communication Foundation HTTP Activation
.NET Framework 4.6 Advanced Services	ASP.NET 4.6 WCF services • HTTP Activation • TCP Activation • TCP Port Sharing
Internet Information Services	Web Management Tools World Wide Web Services

NOTE

If need the other components (for example, FTP Server) can be installed.

ATTENTION

To install and launch IIS server on the Windows 7, enter the "C:\Windows\Microsoft.NET\Framework\v4.0.30319\aspnet_regiis.exe –i" command in the command line.

bpm'online installation sequence

1. Extract the application

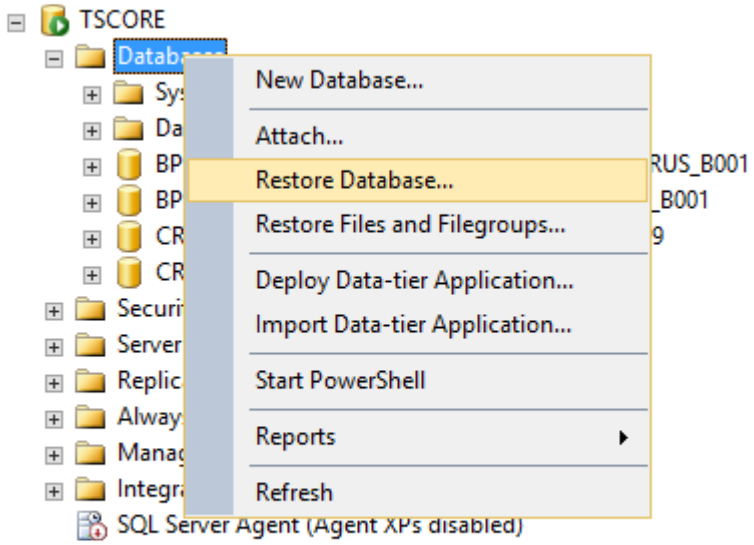
Extract zip archive with the application ti the folder on the application server (for example, c:\bpmonline).

2. Restore the database from a backup

Copy the database backup (located in the "db" folder of the extracted application, for example, C:\bpmonline\db) to the database server.

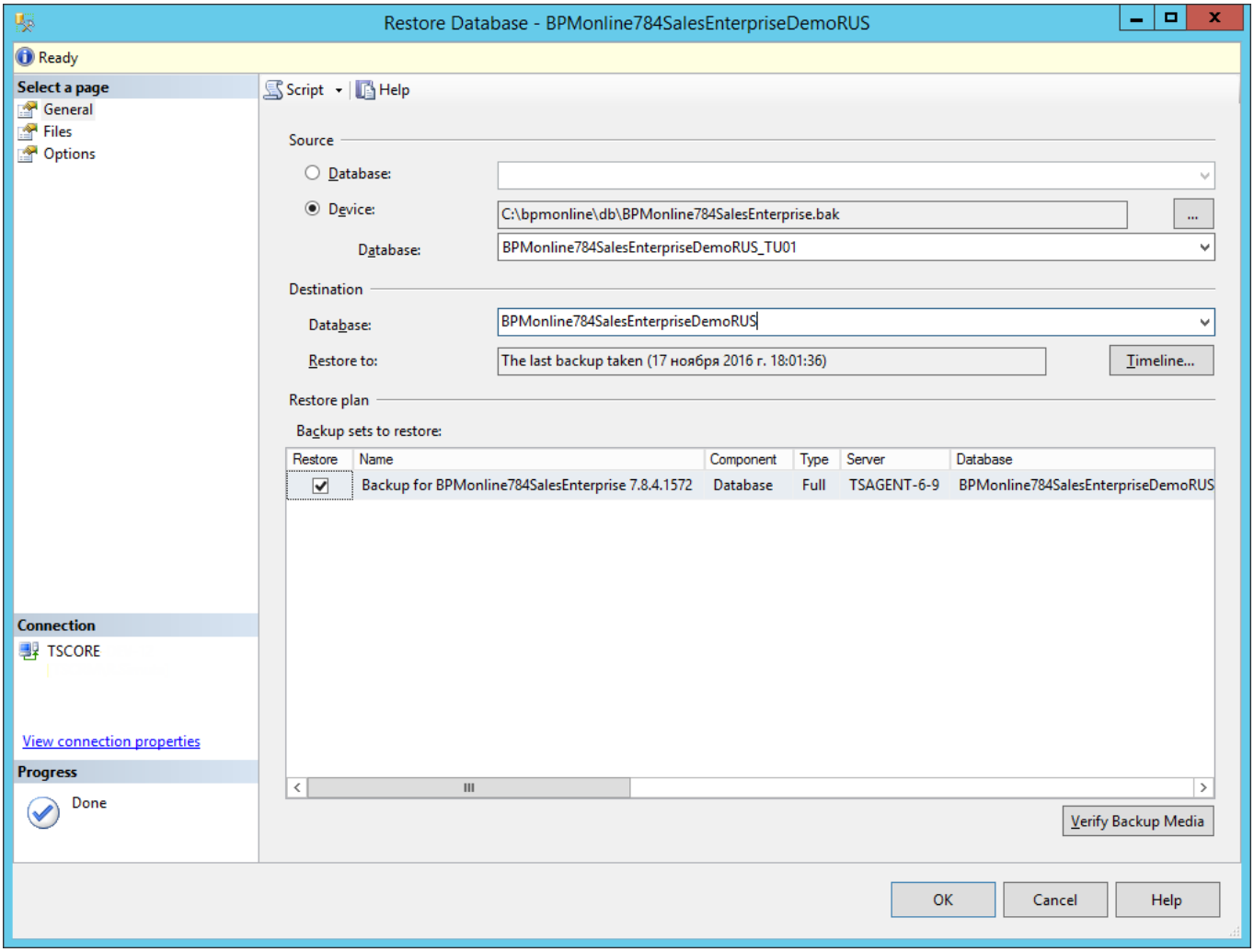
Run the Microsoft SQL Management Studio and select the [Restore Database...] command in the context menu of the "Database" section to restore the database from the backup. (Fig. 3).

Fig. 3. Restoration of the database from the backup



In the opened window, select the [Device] as a source and then spesify a path to the database backup file (Fig. 4).

Fig. 4. Database restoration setup from the backup file



Click [OK] button to launch the database restoration.

> ATTENTION
>
> If you need to get user connected to the database as a database user (not as a domain user), add corresponding user name as the owner of the restored database.

3. Set up the ConnectionStrings.config file

ConnectionStrings.config is located in the application root folder (for example, C:\bpmonline). It is standard XML file. Add there the following information:

The <db> element

- *DataSource* – database server, where the database is deployed.
- *InitialCatalog* – the name of the deployed database.

```
<add name="db" connectionString="Data Source=DBSERVER;Initial
Catalog=BPMonline78;User ID=CustomUser;Password=CustomPassword; Persist Security
Info=True; MultipleActiveResultSets=True; Pooling = true; Max Pool Size = 100; Async
= true" />
```

The <redis> element

- *host* – the computer name, where the Redis server is deployed.

ATTENTION

It is recommended to specify the network name of the machine instead of localhost. Using the setting with localhost under the high load can lead to slow performance.

- *db* – database number, that will be used to store sessions (a number of available databases is configured in the Redis configuration file).

ATTENTION

Each bpm'online application must have its own unique Redis database number.

The <reports> element

- *DataSource* – database server, where the database is deployed.
- *InitialCatalog* – the name of the deployed database.

ATTENTION

The configuration of the server, database, user and password in the <reports> section must be completely identical to the <db> section.

The <defPackagesWorkingCopyPath> element

The *connectionString* attribute should point to a particular local path. The working copies of application packages will be uploaded by this path.

The <sourceControlAuthPath> element

The *connectionString* attribute should point to a particular local path. The client identification information for authentication in the storage will be stored there.

An example of configuring the ConnectionStrings.config file:

```
<?xml version="1.0" encoding="utf-8"?>
<connectionStrings>
    <add name="db" connectionString="Data Source=DBSERVER;Initial
Catalog=BPMonline710;User ID=CustomUser;Password=CustomPassword; Persist Security
Info=True; MultipleActiveResultSets=True; Pooling = true; Max Pool Size = 100; Async
= true" />
    <add name="redis"
connectionString="host=RedisServer;db=0;port=6379;maxReadPoolSize=25;maxWritePoolSize
=25" />
    <add name="reports" connectionString="XpoProvider=MSSqlServer;Data
Source=DBSERVER;Initial Catalog=BPMonline710;User
ID=CustomUser;Password=CustomPassword" />
    <add name="defWorkingCopyPath" connectionString="C:\Projects\WorkingCopy" />
    <add name="defPackagesWorkingCopyPath"
connectionString="C:\Projects\TerrasoftPackages" />
    <add name="clientUnitContentPath"
connectionString="C:\Projects\Dev\ClientUnitSrc" />
    <add name="sourceControlAuthPath" connectionString="C:\Projects\WorkingCopy\Svn"
/>
    <add name="elasticsearchCredentials" connectionString="User=gs-es;
Password=DEQpJMfKqUVTWg9wYVgi;" />
</connectionStrings>
```

NOTE

> The ConnectionStrings.config may contain other configurations. In most cases they are no longer used by the system and will be removed later.

4. Web.config internal file setup

The internal Web.config located in the Terrasoft.WebApp folder is configured in a following way:

The <appSettings> element

- *UseSvn* – by default does nit appear in the Web.config. By default it has the *true* value, i.e. there is a functionality for working with SVN storages in the [Configuration] section. If this functionality needs to be disabled, then add the *UseSvn=false* element to the Web.config.
- *SeparatedJsFiles* – is needed at the debugging, to display the JavaScript page code in a readable view. Add this parameter if it is disabled. Optional.

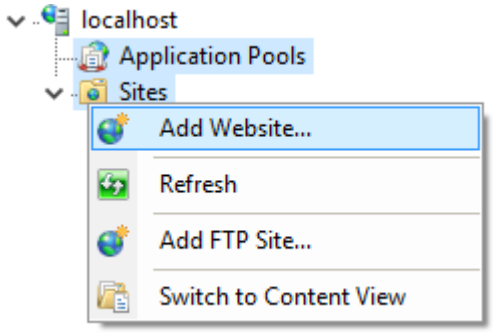
An example of internal Web.config configuration (<appSettings>) section:

```
<appSettings>
  ...
  <add key="UseSvn" value="true" />
  ...
  <add key="IncludeDebugInformation" value="true" />
  ...
  <add key="CompilerSourcesTempFolderPath" value="C:\Projects\Dev\src" />
  ...
</appSettings>
```

5. Configure the application site on IIS.

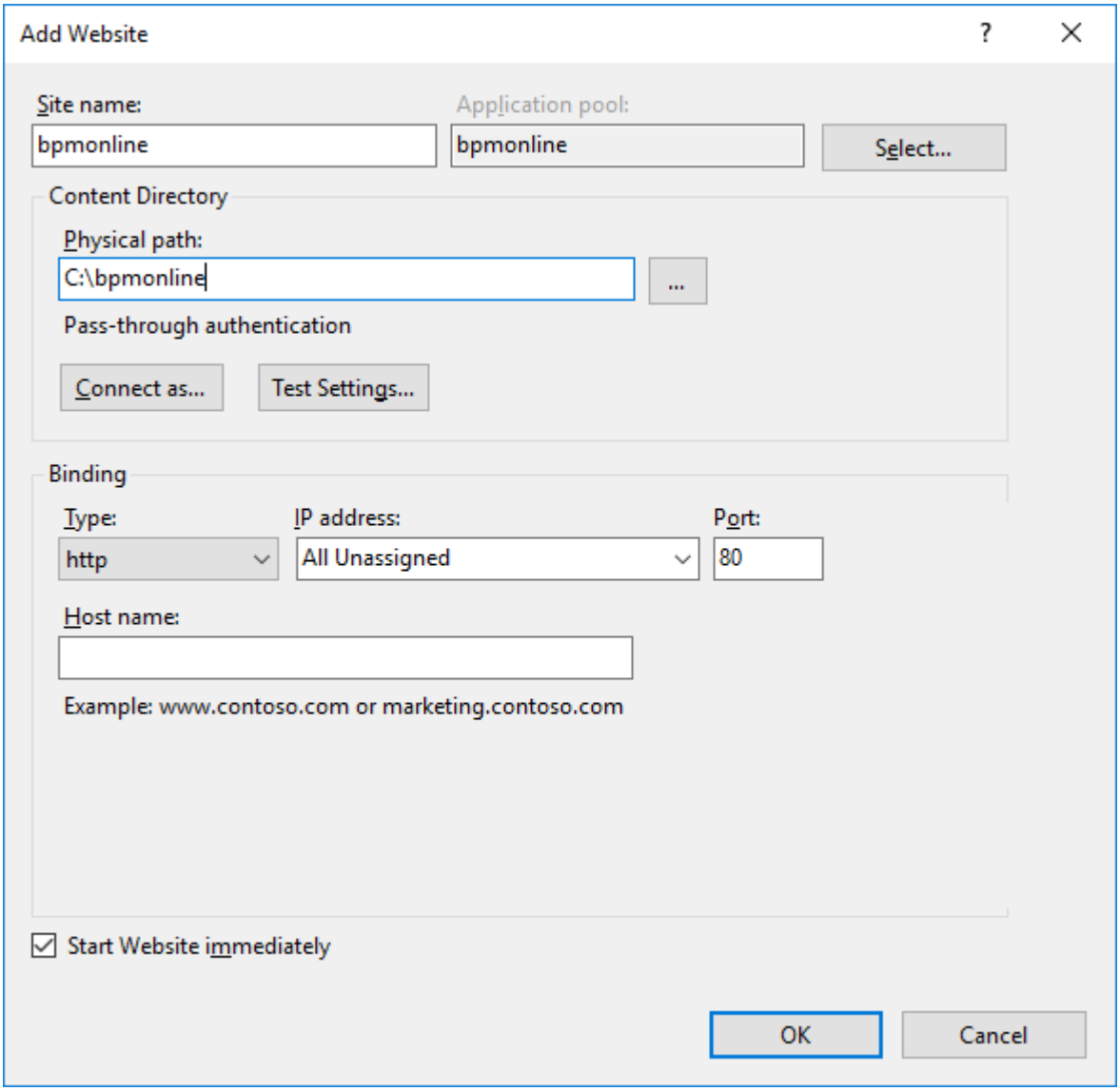
Launch the Internet Information Services and add a new website (Fig. 5).

Fig. 5. Adding the website in IIS



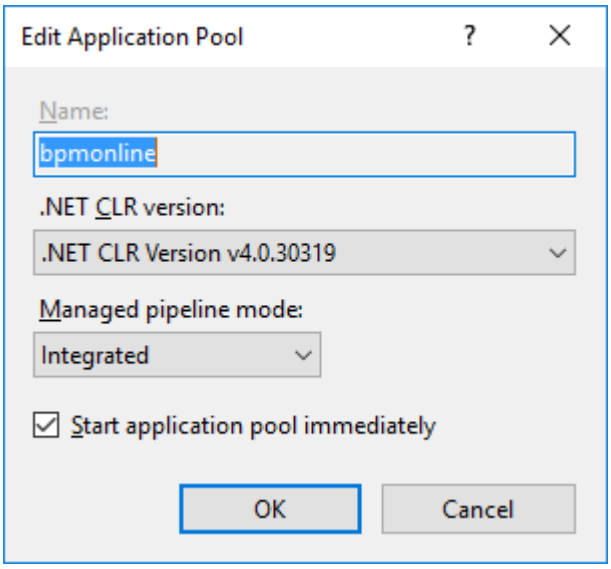
Specify an arbitrary name of the website in the website settings and specify the root folder of the application as the content directory (Fig. 6). You can also specify the external address of the website in this window.

Fig. 6. Settings of the created application



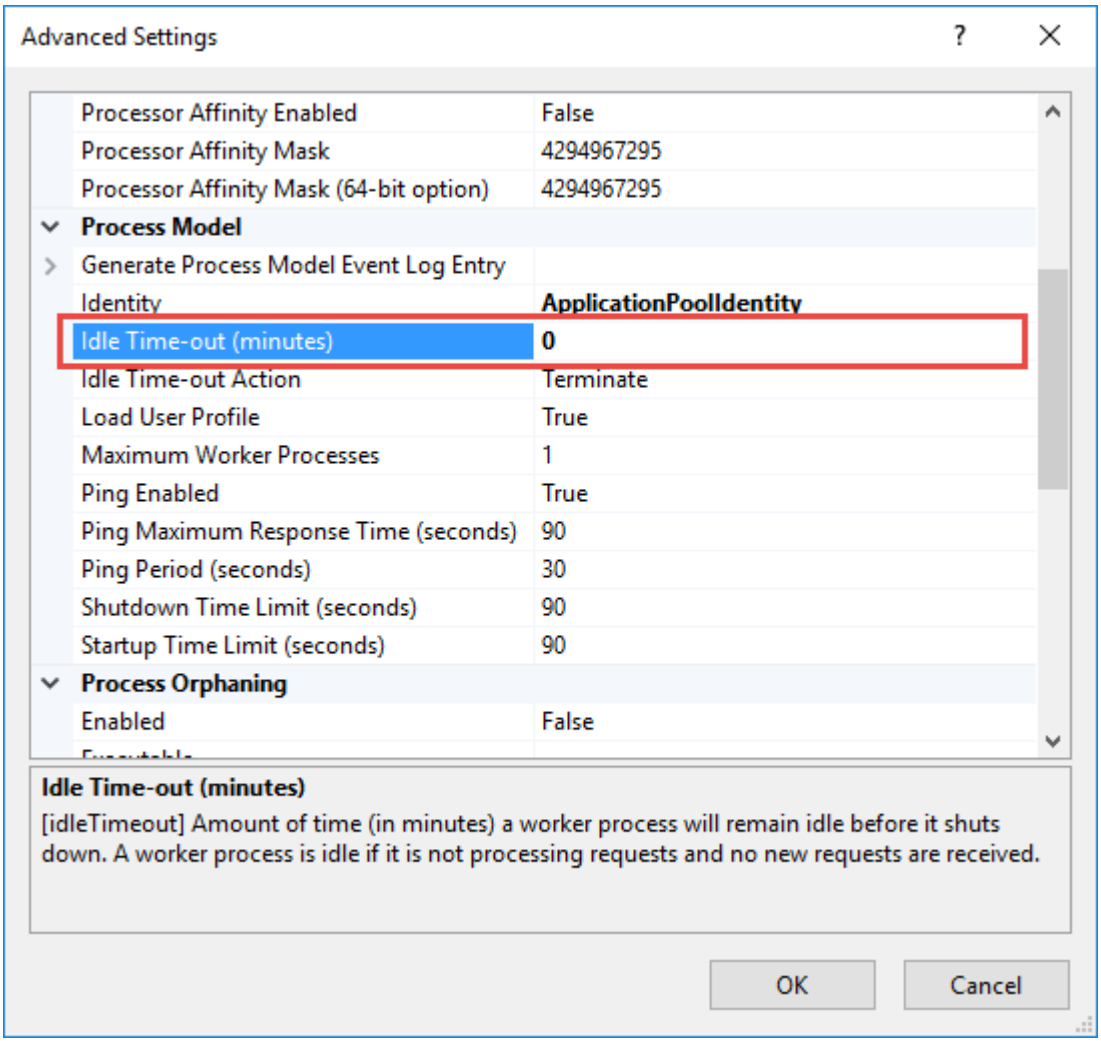
Then, in the list of application pools, check the version of the .NET environment for the created pool – version v4.0 and the [Integrated] managed pipeline mode must be specified. If another version or mode is specified, change them by selecting the [Basic settings...] on the [Actions] panel.

Fig. 7. Setting the required version of the .NET environment



Select the [Advanced settings...] in the context menu of the selected pool. Set the "0" value in the [Idle Time-out (minutes)] field in the opened window(Fig. 8).

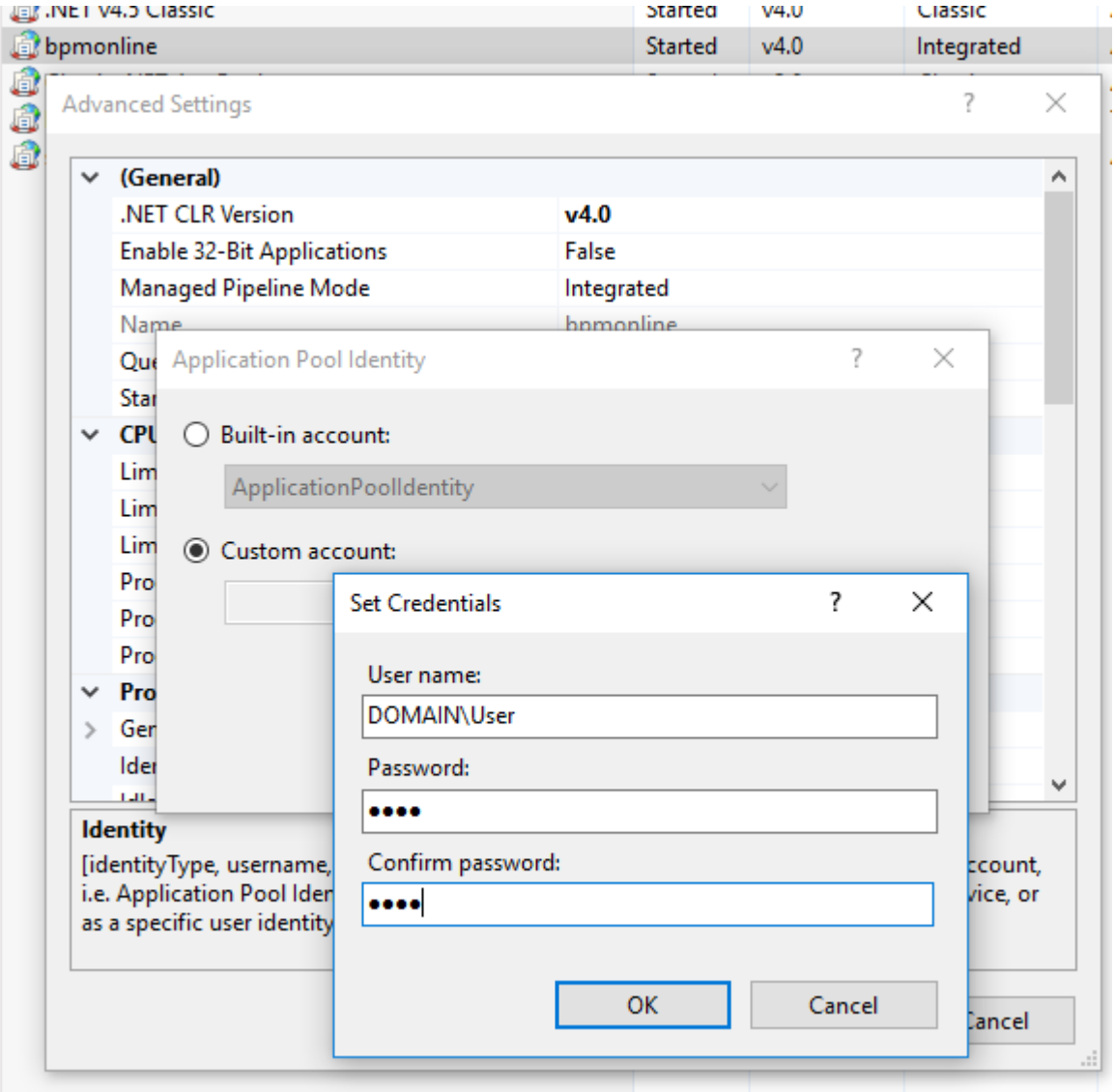
Fig. 8. Setting the Idle Time-out



Specify the account, that will be used by application to access the application folder. To do this, click the [...] button in the [Identity] field to edit this property. In the opened window, select the [Custom account] option and click the [Install] button.

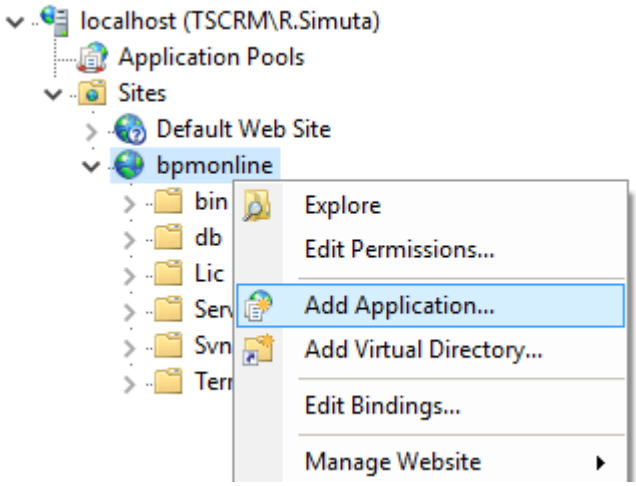
Then, enter the user name and password of the domain user account that is used by the application to access the file system. For example, it can be the domain user that has started the bpm'online installation (Fig. 9).

Fig. 9. Identity configuration for the application pool.



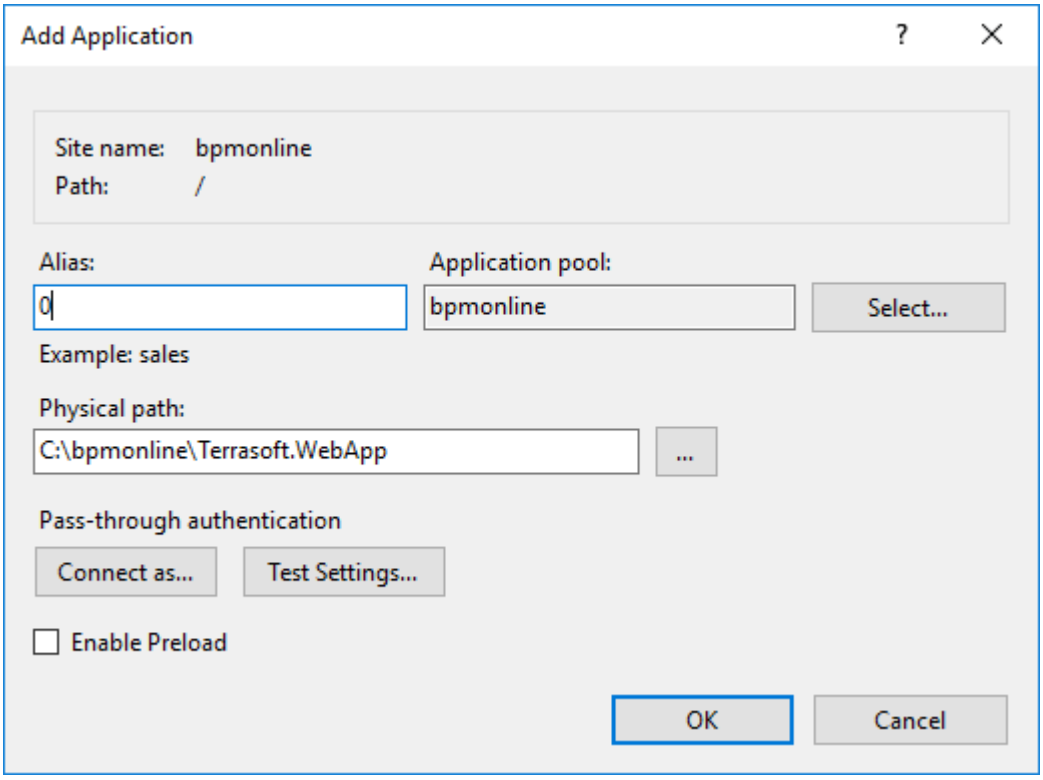
Add the application to the website (Fig. 10).

Fig. 10. Adding application to the website



In the application settings, set "0" value in the [Alias] field, and the path to the "*Terrasoft.WebApp*" folder in the [Physical pass] field (Fig. 11).

Fig. 11. Settings of the created application

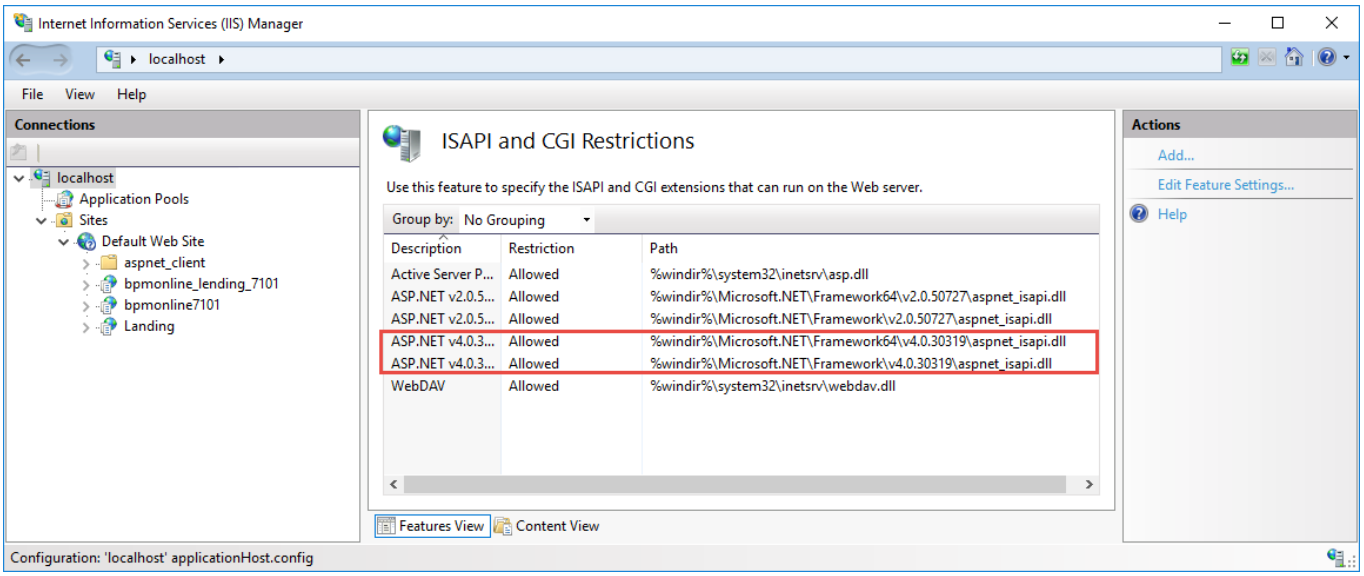


> ATTENTION
>
> The "0" application corresponds to the [Default] workspace. Before creating a new workspace, you must add a new application to the system's website, indicating the next sequential number (1, 2, ...) as the alias. Other application settings must be the same as in the "0" application.

Make sure that:

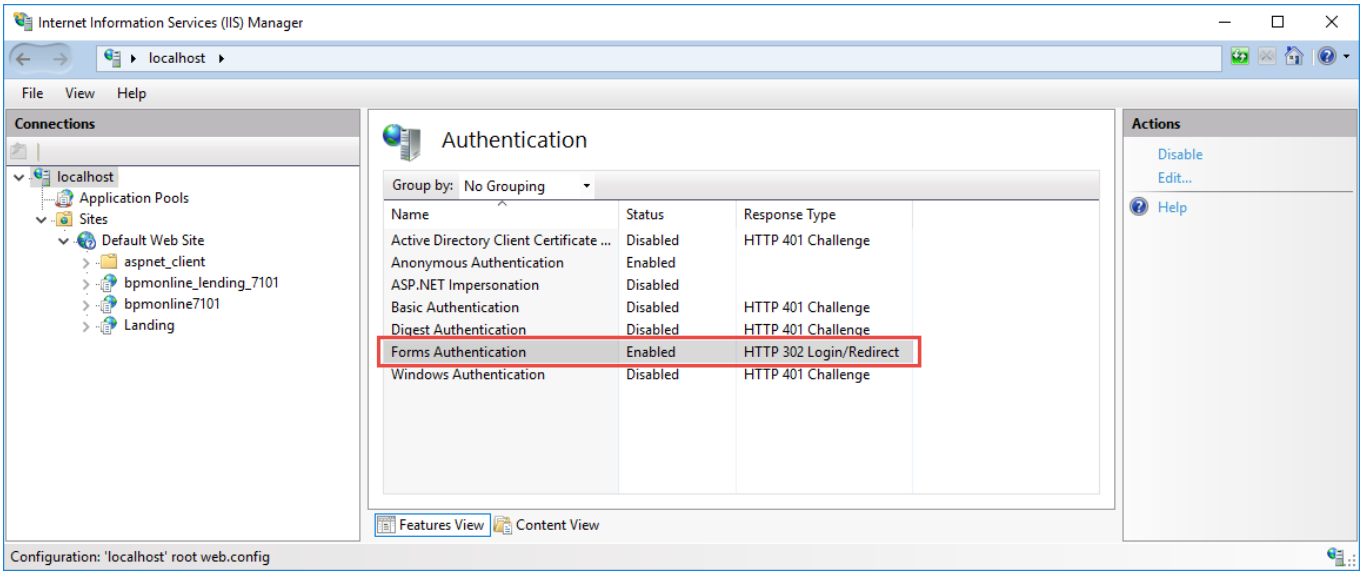
a. The ISAPI and CGI limitations are not set in the used ASP.NET version (Fig. 12).

Fig. 12. ISAPI and CGI limitations



d. Forms authentication is enabled (Fig. 13).

Fig. 13. Authentication settings



Used ports

Ports by default:

Application name	Used port
bpm'online	80 (http), 443 (https) or another specified for application in the IIS. For more information please see the article.
SVN	3690
Redis	6379
MSSQL Server	1433 or another if more than one instance is installed. For more information please see the article.

Deploying the bpm'online "cloud" application

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 data center 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 user and setup workspaces

The next step before starting development is the creation of a user with administrator rights and adding a new workspace.

Creating a new user

The system already has a user with system administrator rights called "superuser". You can carry out development in the bpm'online system using this user account. We recommend creating a separate account for each company employee who will perform development.

For more information about creating of new user and adding necessary rights, see article "How to add new user in bpm'online". Assign a role [System administrators] to the new user so that they would be able to perform development in the system.

Creating a new workspace

A workspace in the bpm'online system is a set of packages. Each of them, in its turn, is a set of definite functionalities. You can form the final functionality of the system by managing the combination of packages in a certain workspace. All workspaces are independent.

> ATTENTION
>
> It is not recommended to perform development in several configurations. Please, use *Default* configuration only. Creation of configurations and development in the production environment is strictly forbidden.

Create repository in SVN server

The next step, necessary for starting development on the platform, is creating the repository on the server of the version control system.

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.

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.

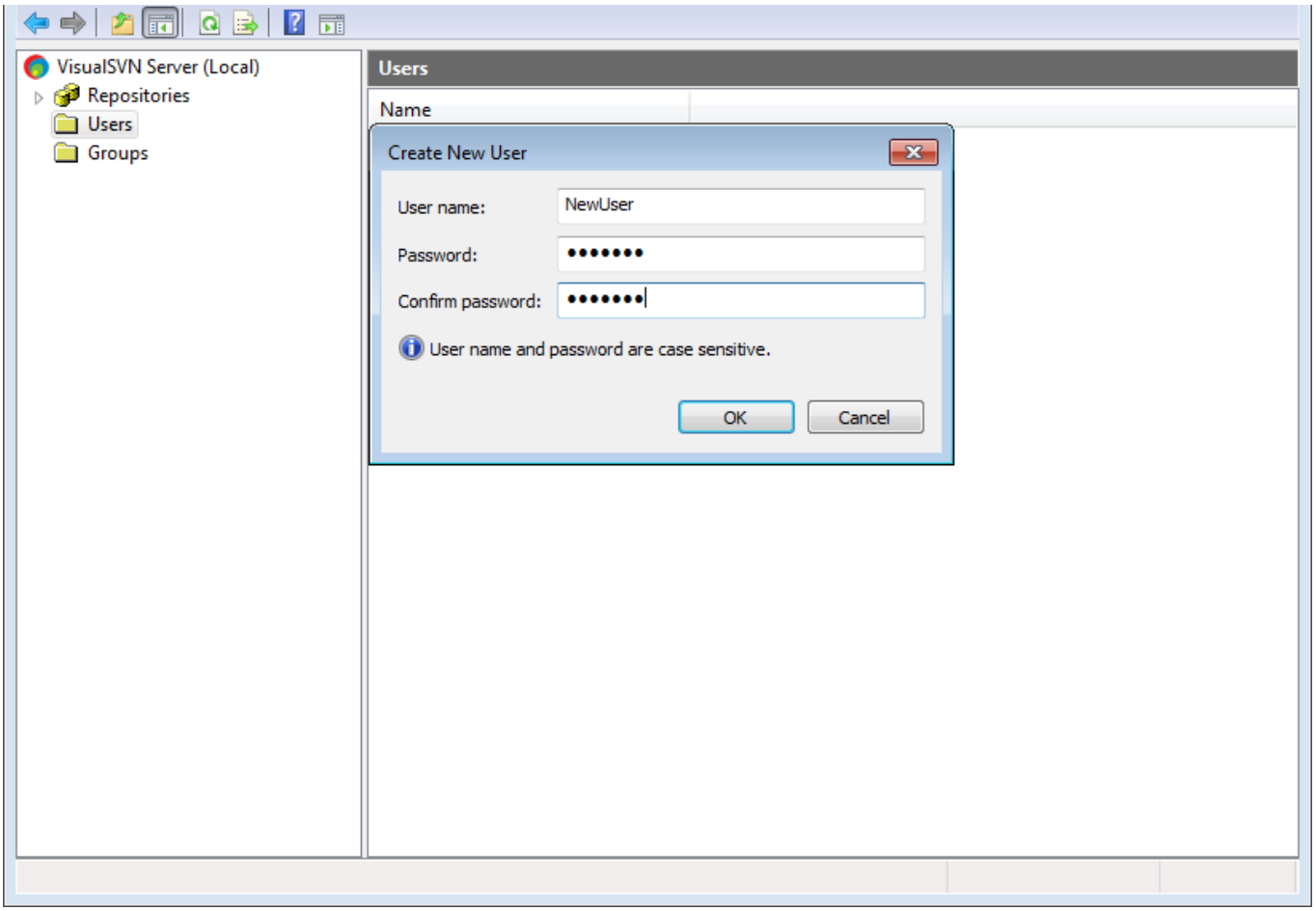
SVN server can function independently or through Apache web-server (it is installed automatically be means of the VisualSVN and CollabNet utilities). In the first case, repositories are accessed through svn:// protocol. In the second case repositories are accessed through the http(s):// protocol.

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.

Principles of operation with repositories of version control system

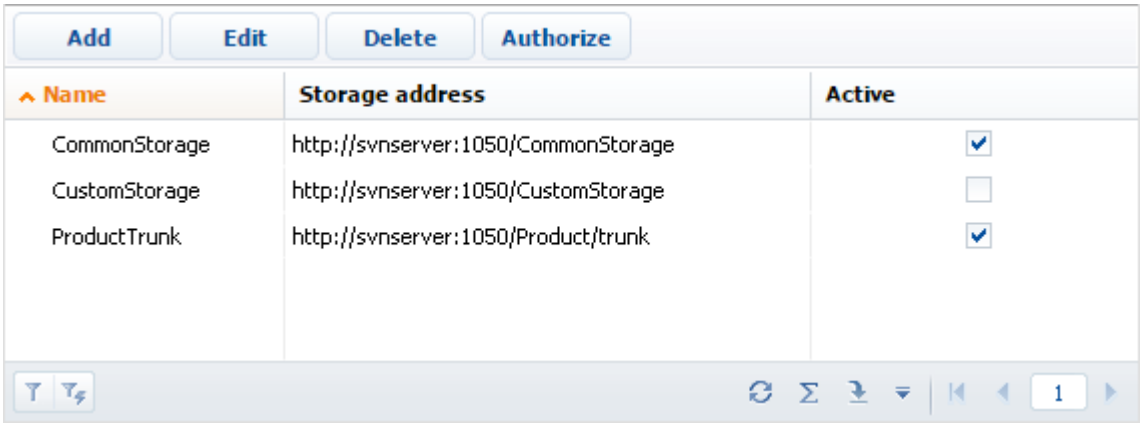
- Repository list is common for all workspaces of the system.
- You can add newly created packages to any repository in the list.
- You can commit an already installed package only to the repository that was specified when the package was created.
- You can install any number of packages from the list of available repositories in the workspace

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

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.

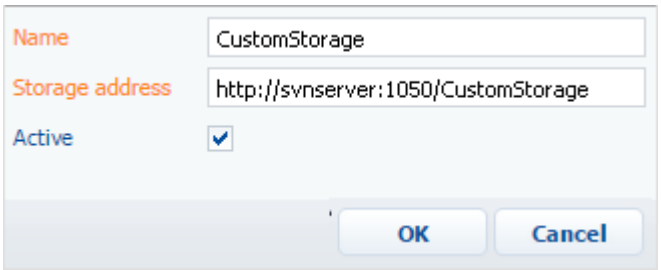
Fig. 2. - Window with repository list of version control system



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.

Previous step — **Create user and setup workspaces**

Next step — **Creating and installing a package for development**

Working with packages

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 packages from an application**";
- "**Exporting packages from the application interface**";
- "**Creating a package in the file system development mode**".

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*), packages that extend the functionality of the system (for example, 1C integration packages, telephony, etc.), and packages developed by third-party developers. These packages are installed from zip archives by using the WorkspaceConsole utility.
- Custom packages are created by the users of the system. These packages can be attached to the SVN repository.

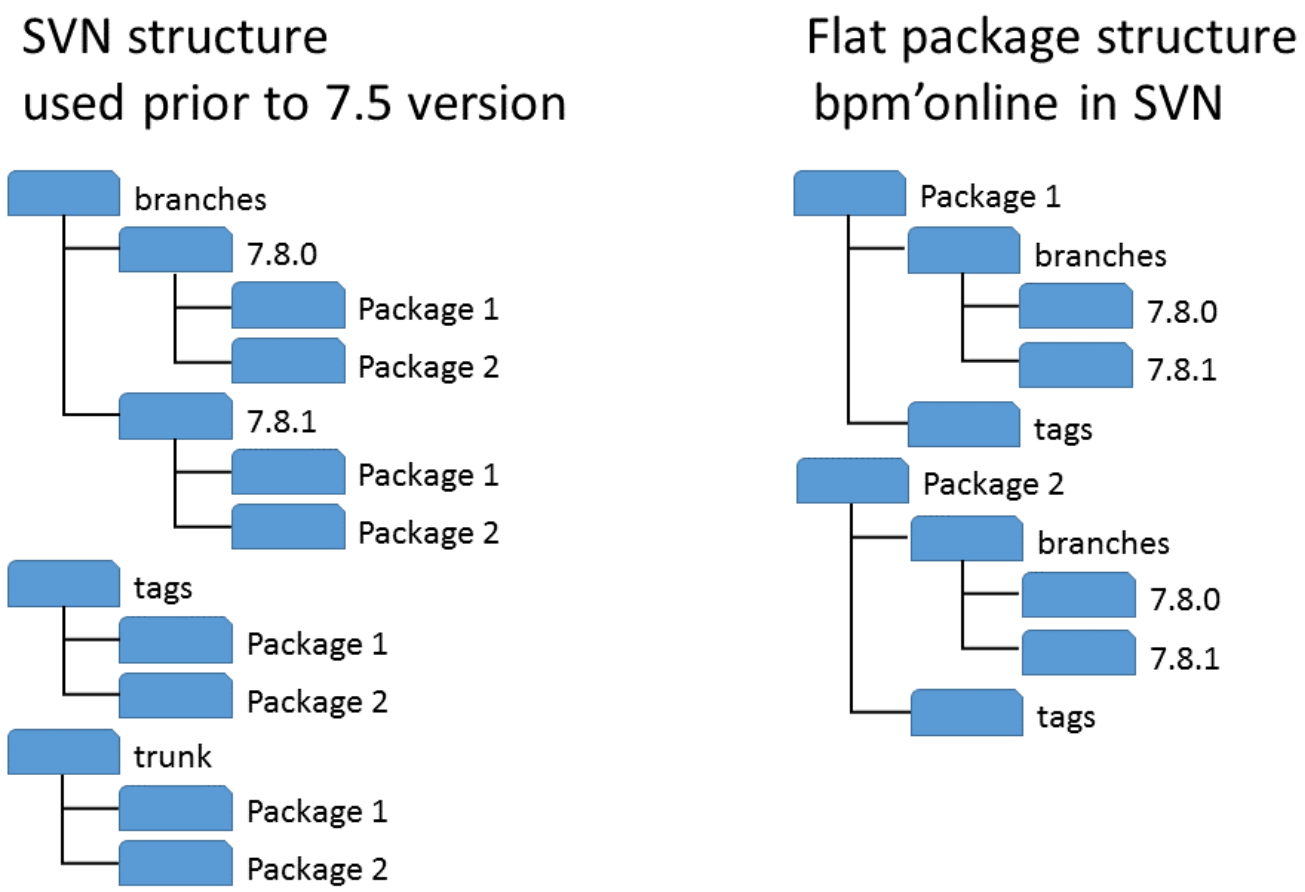
Configuration elements of the pre-installed packages cannot be modified. You can develop additional functionality or modify the existing functionality only via custom packages

Flat package structure in the repository

Bpm'online uses flat package structure. The “branches” and “tags” directories that store different versions of the

project are located inside the package root directory (Fig. 1). Thus, each package is a separate version repository of a development project.

Fig. 1. Previous structure of the version repository and the flat bpm'online package structure



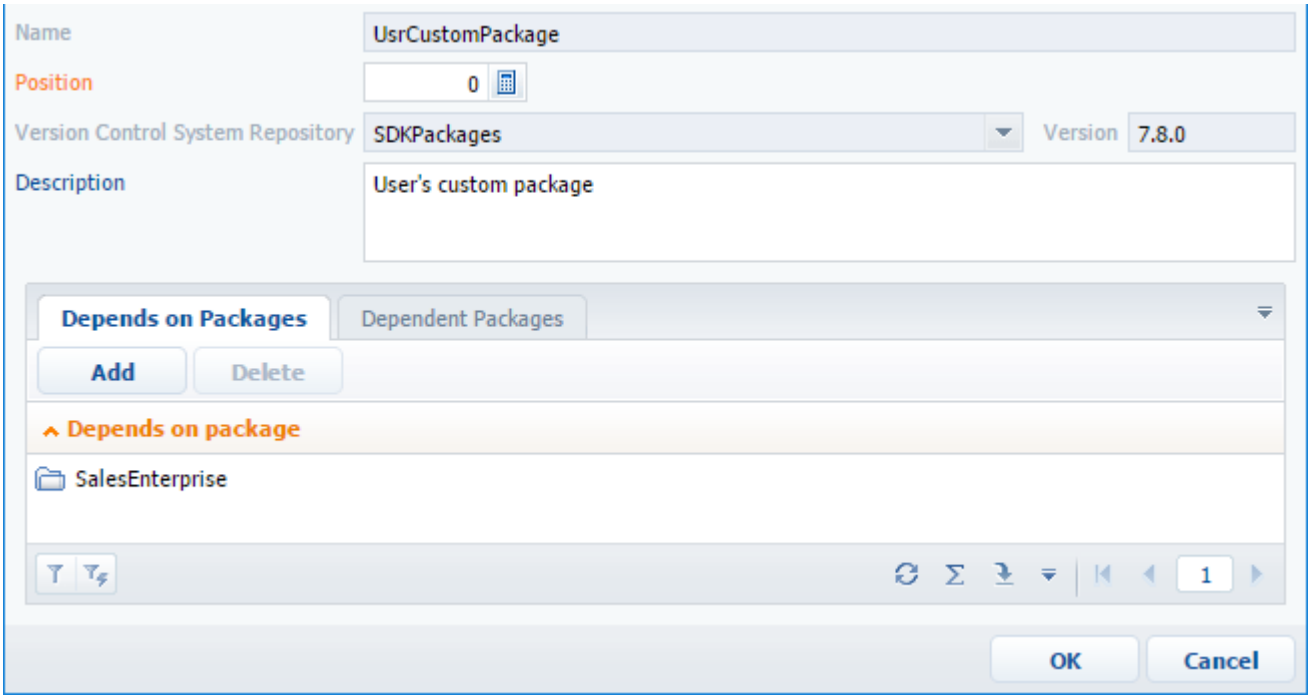
In contrast to the previously used storage structure, the *trunk* directory is not used in bpm'online. The application only works with a specific package version that was specified during its creation or installation from the repository. Different package versions are stored in the *branches* directory.

Package version

One of the main 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.

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(1477652941000)\/",
    "Maintainer": "Customer",
    "Description": "Пакет пользователя",
    "DependsOn": [
      {
        "UId": "e14dcfb1-e53c-4439-a876-af7f97083ed9",
        "PackageVersion": "7.8.0",
        "Name": "SalesEnterprise"
      }
    ]
  }
}
```

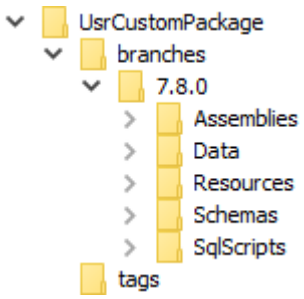
All elements of the package are of the same version as the package itself.

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*.

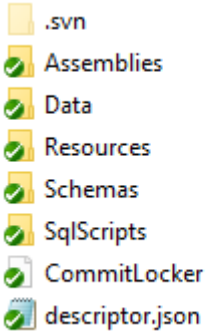
The *tags* directory stores tags. The tags in the version control system represent a "snapshot" of the project at a certain point in time, a static copy of the files required for saving some critical stage of development.

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.

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

The *CommitLocker* file is a utility file that is used in the package locking mechanism.

Package dependencies. Basic application packages

Introduction

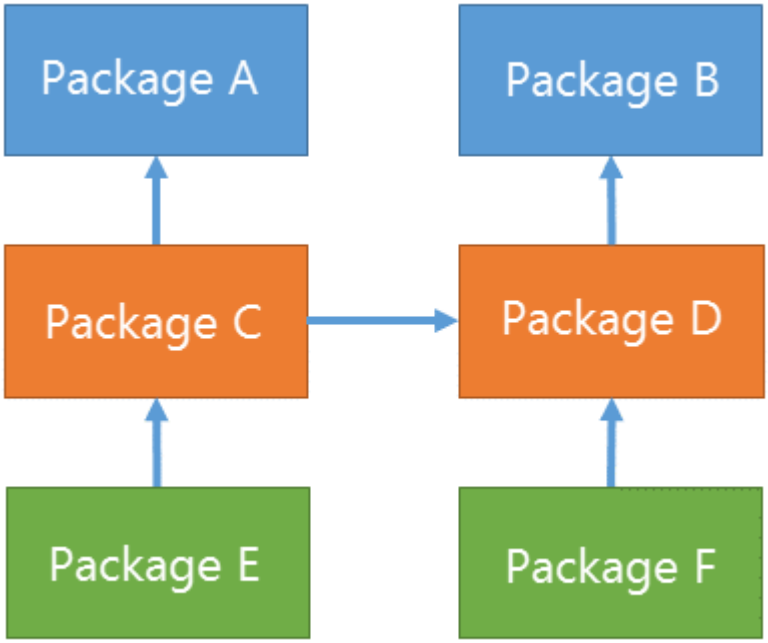
Bpm'online application development follows the basic principles of software design, in particular the "don't repeat yourself" (DRY) principle. In the bpm'online architecture, the concept of packages was built around this principle and is implemented using dependencies between packages. Each package contains 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

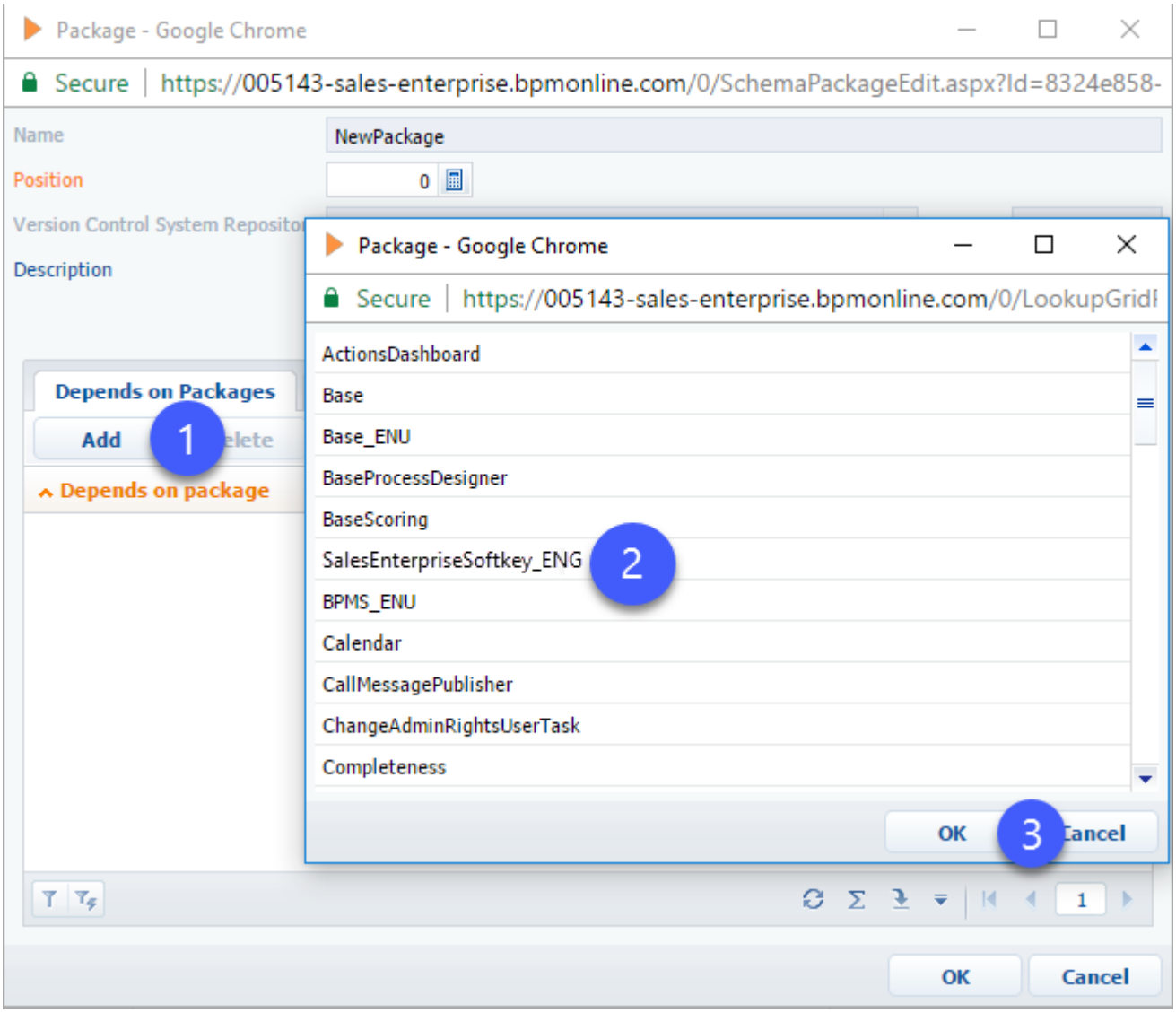


Package dependencies form a hierarchical chain. This means that if you add a package to the dependency of another package, the dependent package will contain all functionality of the added package as well as functionality of all packages that the added package depends on. The closest analogy of the package 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

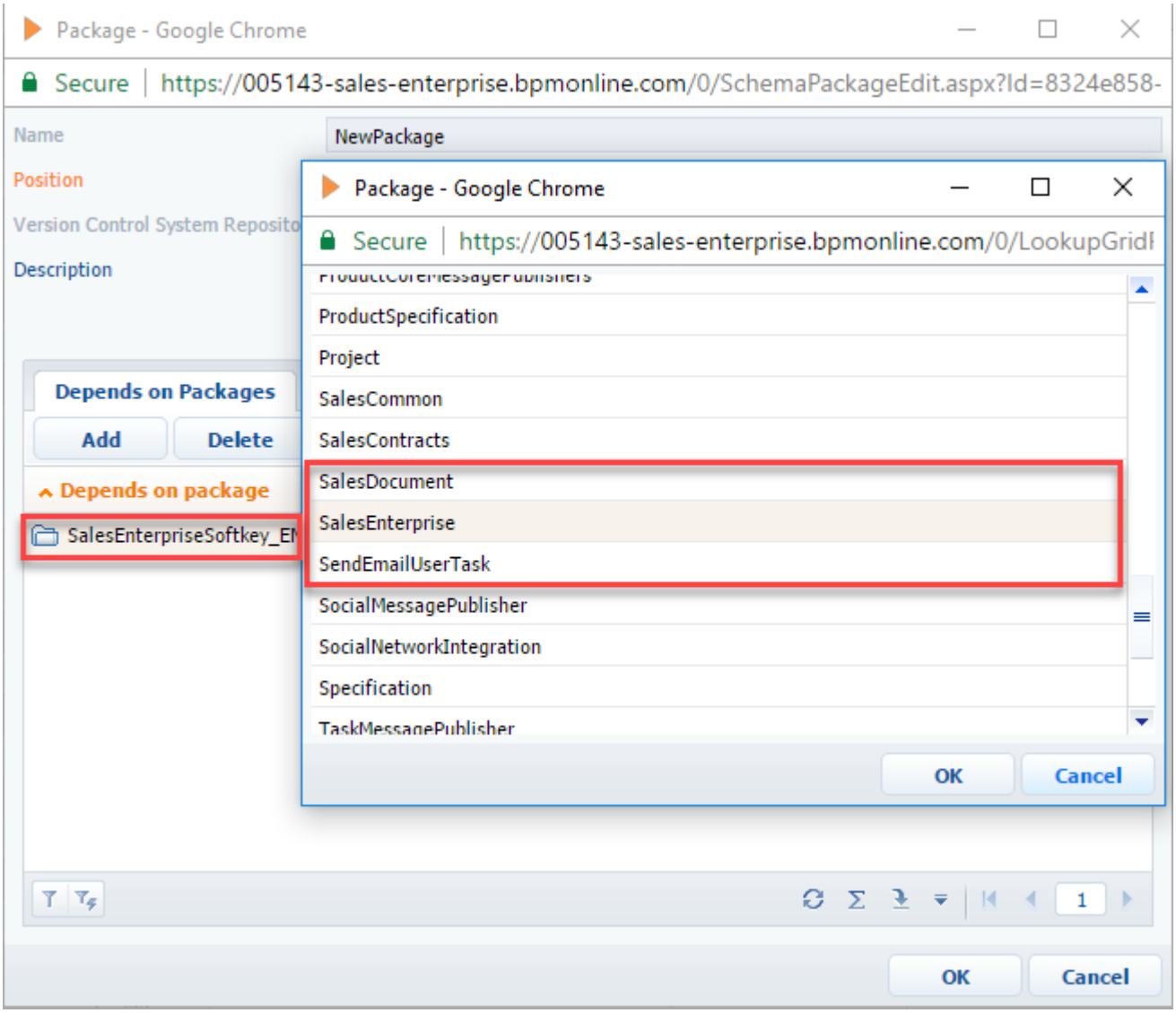
Dependencies can only be added to a custom package, and only after it has been created. To add dependencies, click the [Add] (1) button on the [Depends on packages] (Fig. 2) detail. In the opened dialog of the package lookup, select the required package (2) and click the [OK] button (3).

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



ATTENTION

After creating a new package, add it to the dependencies of the pre-installed "Custom" package.

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 a descriptor.json: Example of *descriptor.json*:

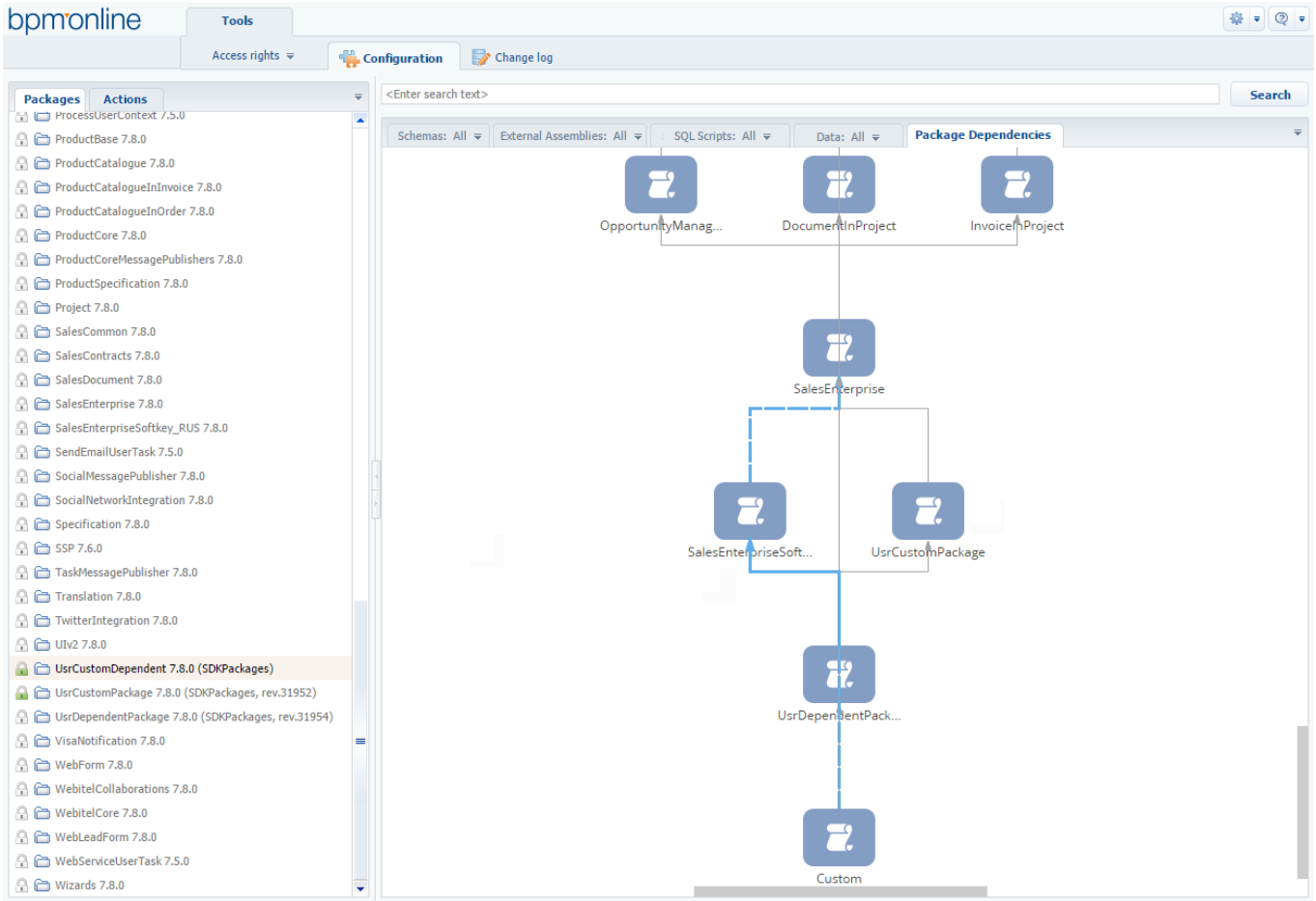
```
{
  "Descriptor": {
    "UId": "974305e3-b01c-4020-b511-6fe9ac644d95",
    "PackageVersion": "7.8.0",
    "Name": "UsrDependentPackage",
    "ModifiedOnUtc": "\/Date(1478766323000)\/",
    "Maintainer": "Customer",
    "Description": "Пример пакета, зависящего от других пакетов",
    "DependsOn": [
      {
        "UId": "8bc92579-92ee-4ff2-8d44-1ca61542aa1b",
        "PackageVersion": "7.8.0",
```

```
        "Name": "UsrCustomPackage"
      },
      {
        "UId": "0ece3a34-413f-415b-919b-7b29bb31701f",
        "PackageVersion": "7.8.0",
        "Name": "SalesEnterpriseSoftkey_RUS"
      }
    ]
  }
}
```


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. 4).

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 English version of the SalesEnterpise product, the [SalesEnterpiseSoftkey_ENG] depends only on the [SalesEnterpise] package and all its dependencies (Fig. 4). The [Custom] and [UsrDependentPackage] packages also depend on the [SalesEnterprise] package.

In the package hierarchy, the [Custom] package must be the root package with direct dependencies from all packages of the system. For more information about the "**Package [Custom]**" package, please see the "[Custom] package" article.

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

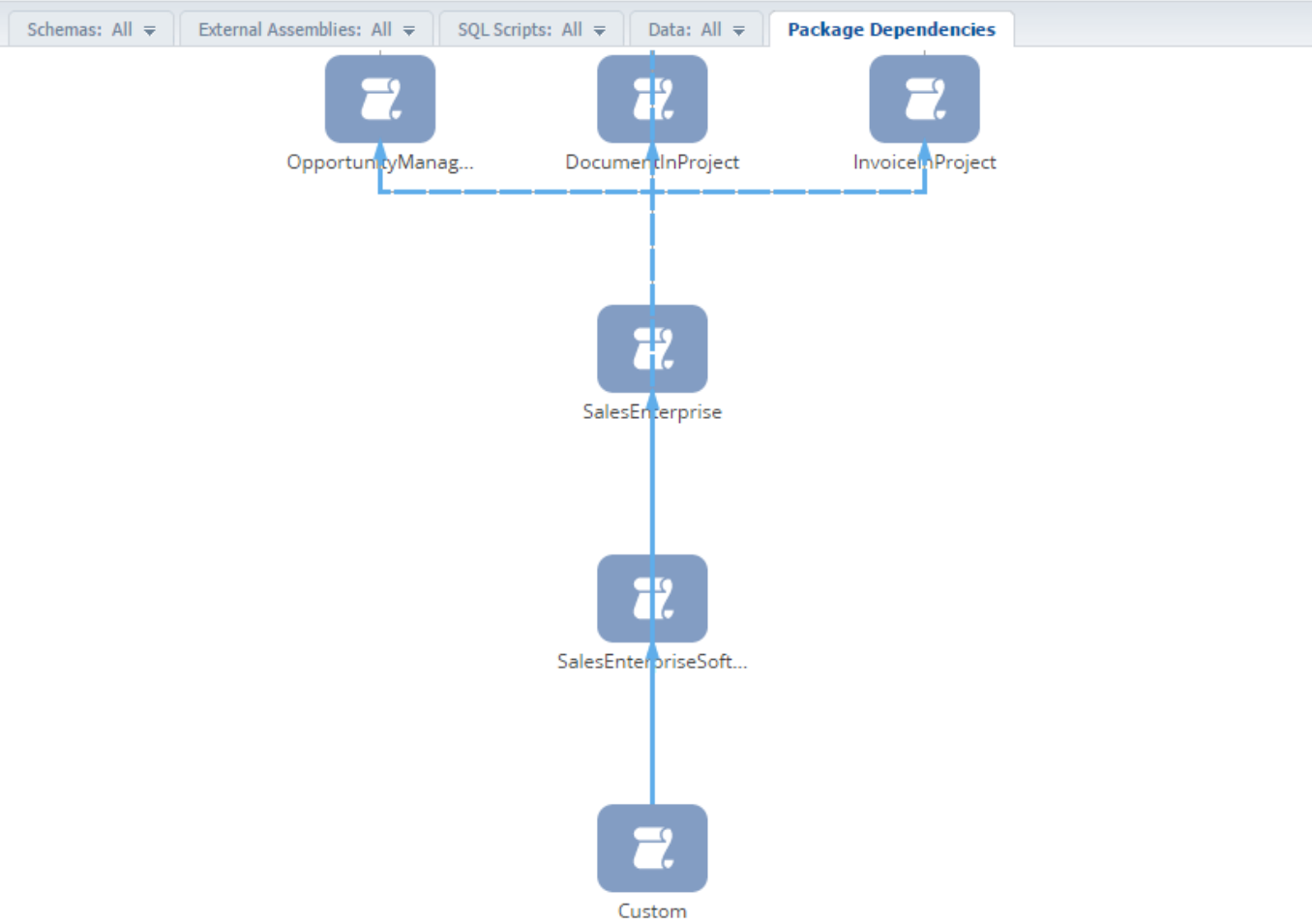
As a pre-installed package, "Custom" cannot be added to the SVN subversion control repository. The schemas can be transferred from the "Custom" package only by using the export/import function.

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 installed from the SVN or via the WorkspaceConsole utility, a dependency from this package is automatically added to the "Custom" package. When creating a new package, add dependencies for the "Custom" package manually. 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 root package 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

In bpm'online all the configuration elements are grouped in packages.

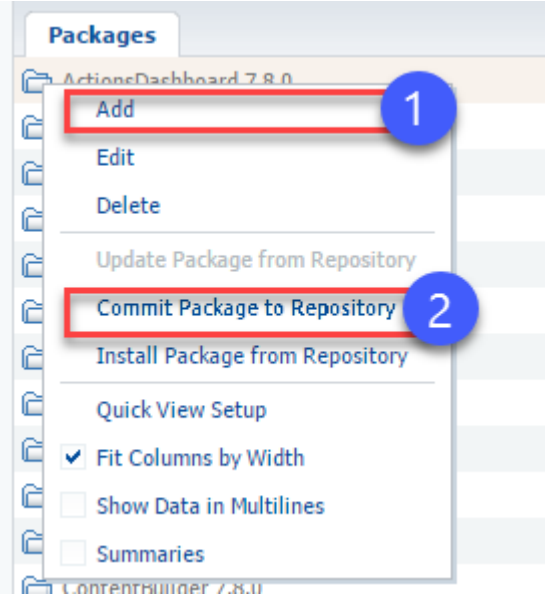
A bpm'online package is a collection of configuration elements (schemas, data, scripts, additional libraries) that implements particular block of functionality. In the file system, packages are directories with various subdirectories and files. Basic information about the packages are described in the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

"**Package dependencies. Basic application packages**"

How to create a custom package

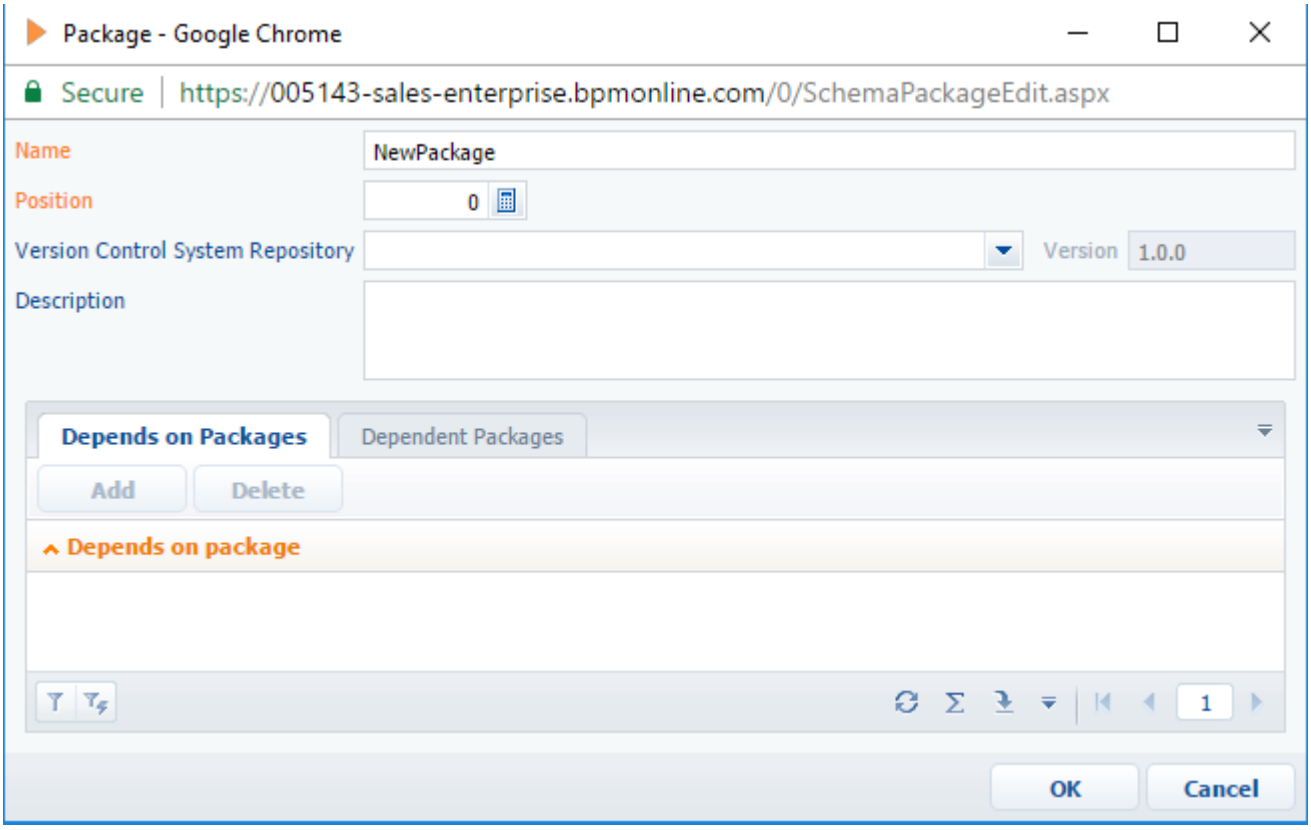
To create a new custom package, go the [Packages] tab menu of the [Configuration] section and select the [Add] action (Fig. 1. 1).

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:

- [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.
- [Name] - package name. Required field. Name cannot match the names of already existing packages.
- [Revision control system storage] — the revision control system storage name to which package modifications will be committed. 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.
- [Position] - package position in hierarchy Required field. The default value is set to 0.
- [Description] - package description, for example, extended information about package functions. This is a non-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.

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.

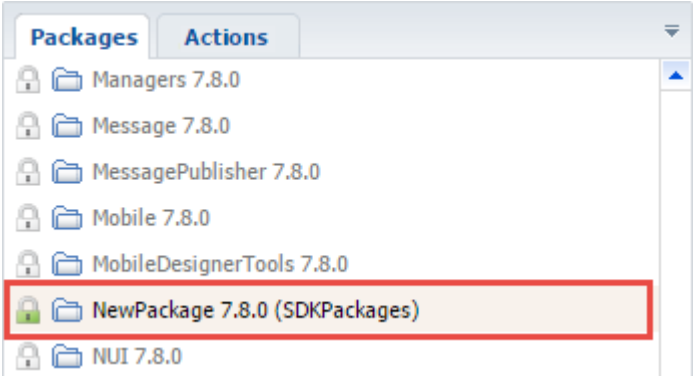
The contents of the key package mini-page fields pack will be saved in its metadata:

```
{
  "Descriptor": {
    "UId": "8bc92579-92ee-4ff2-8d44-1ca61542aa1b",
    "PackageVersion": "7.8.0",
    "Name": "UsrCustomPackage",
    "ModifiedOnUtc": "\/Date(1477652941000)\/",
    "Maintainer": "Customer",
    "Description": "Пакет пользователя",
    "DependsOn": [
      {
        "UId": "e14dcfb1-e53c-4439-a876-af7f97083ed9",
        "PackageVersion": "7.8.0",
        "Name": "SalesEnterprise"
      }
    ]
  }
}
```

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

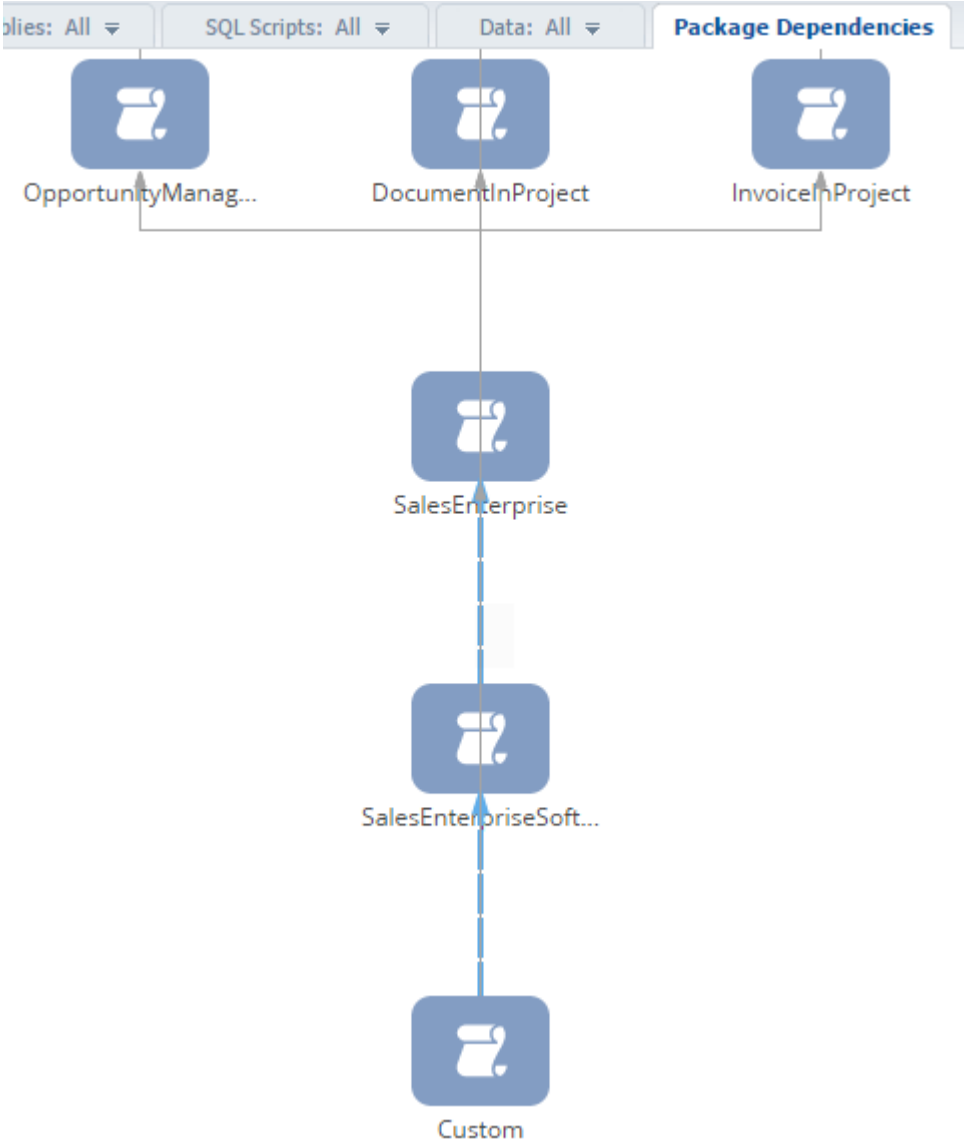


For the created package to have all the functionality that is inherent in the system, you need to specify the dependencies. To do this, indicate the last package in a hierarchy of pre-installed packages. To determine which of the packages in the hierarchy of packages is the latest, you need to go to the [Package dependencies] tab of the [Configuration] section. Next, you must find the first package located above the [Custom] package. For example, fig. 4 shows that the [SalesEnterpriseSoftkey_RUS] package is the 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.

The final stage in creating a new package is **committing it to the revision control system storage** (Fig. 1, 2).

Committing a package to repository

Introduction

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

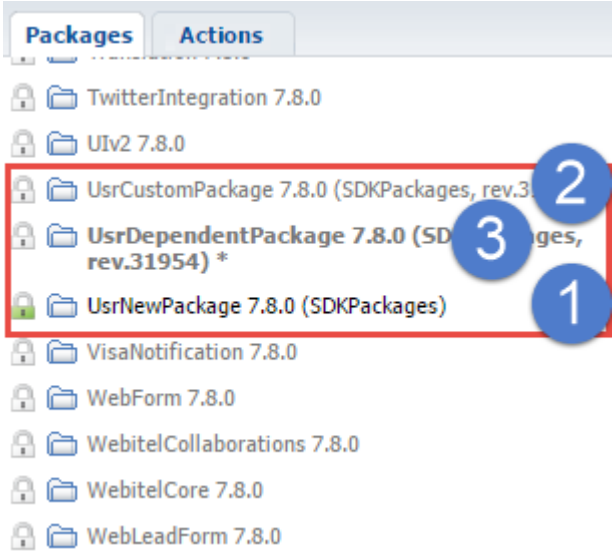
> ⚠ ATTENTION
>
> Packages are commited manually. Modifications of other configuration packages are not committed.

Package committing is required when:

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

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

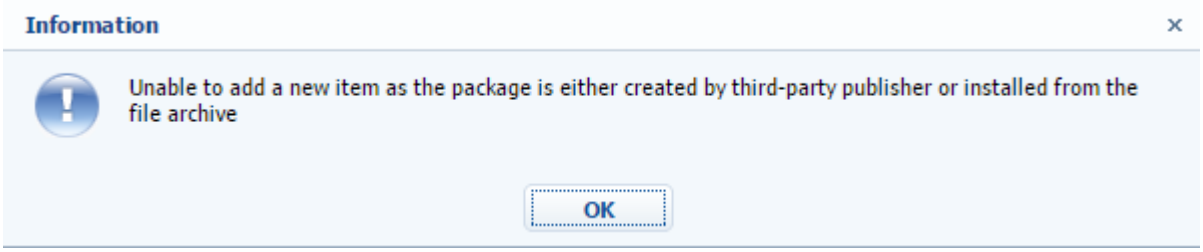


The system displays the names of custom packages that have already been committed, the name of the storage and the latest SVN revision number. The basic package is displayed in the same way as the custom package (Fig. 1.2). If a custom package has been modified, its name will be displayed in bold (Fig.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

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

If a package has not been committed to SVN, then when adding a new element to it, a notification will pop up (Fig. 2).

Fig. 2. Package not committed to SVN notification

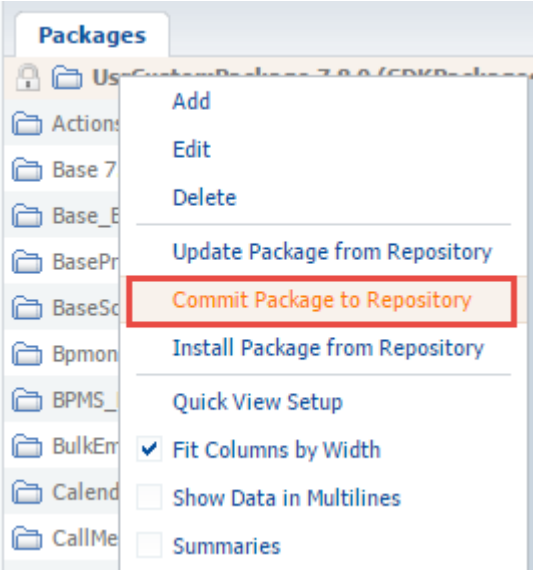


Committing a package to storage

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

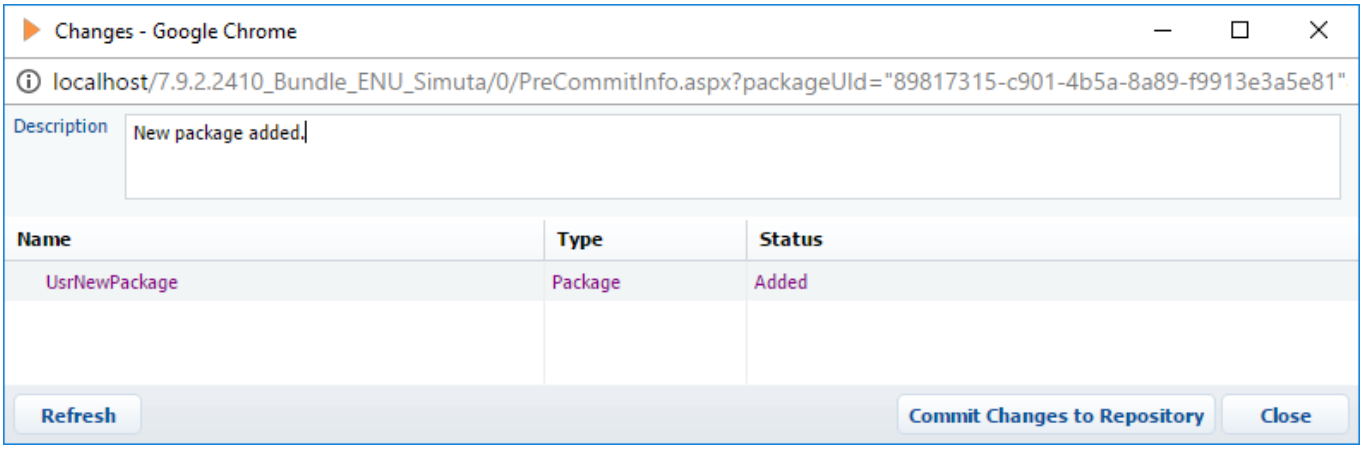
> ATTENTION
>
> When the **file system development mode is enabled**, the SVN integration mechanism is turned off. Therefore, the [Commit package to repository] action is unavailable.

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



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

Fig. 4. The [Modifications] 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 modifications to storage] button, the package will be committed and the mosdifications 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 work environments.

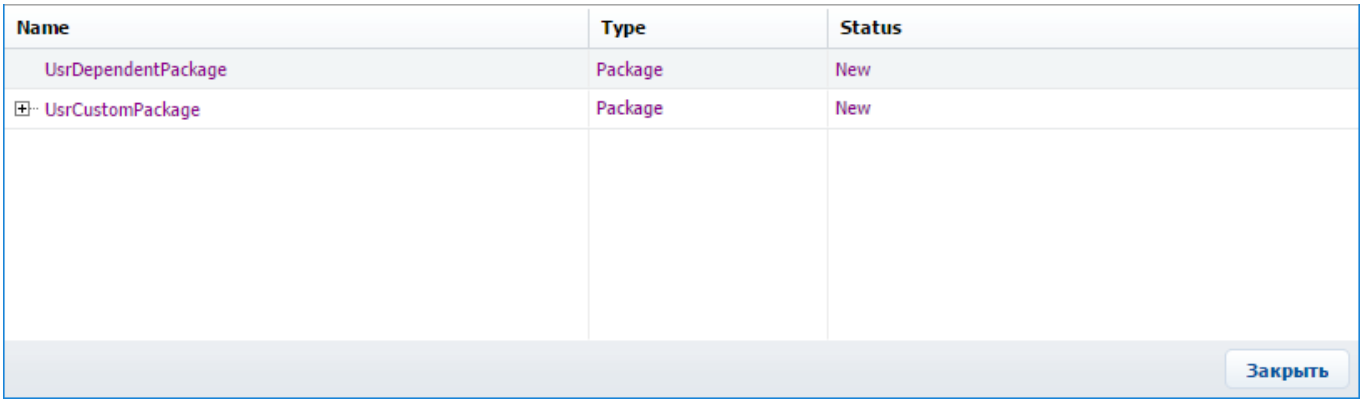
Package installation sequence

The sequence of package installation is described in detail in the "**Transferring changes using SVN**".

Changes in package hierarchy

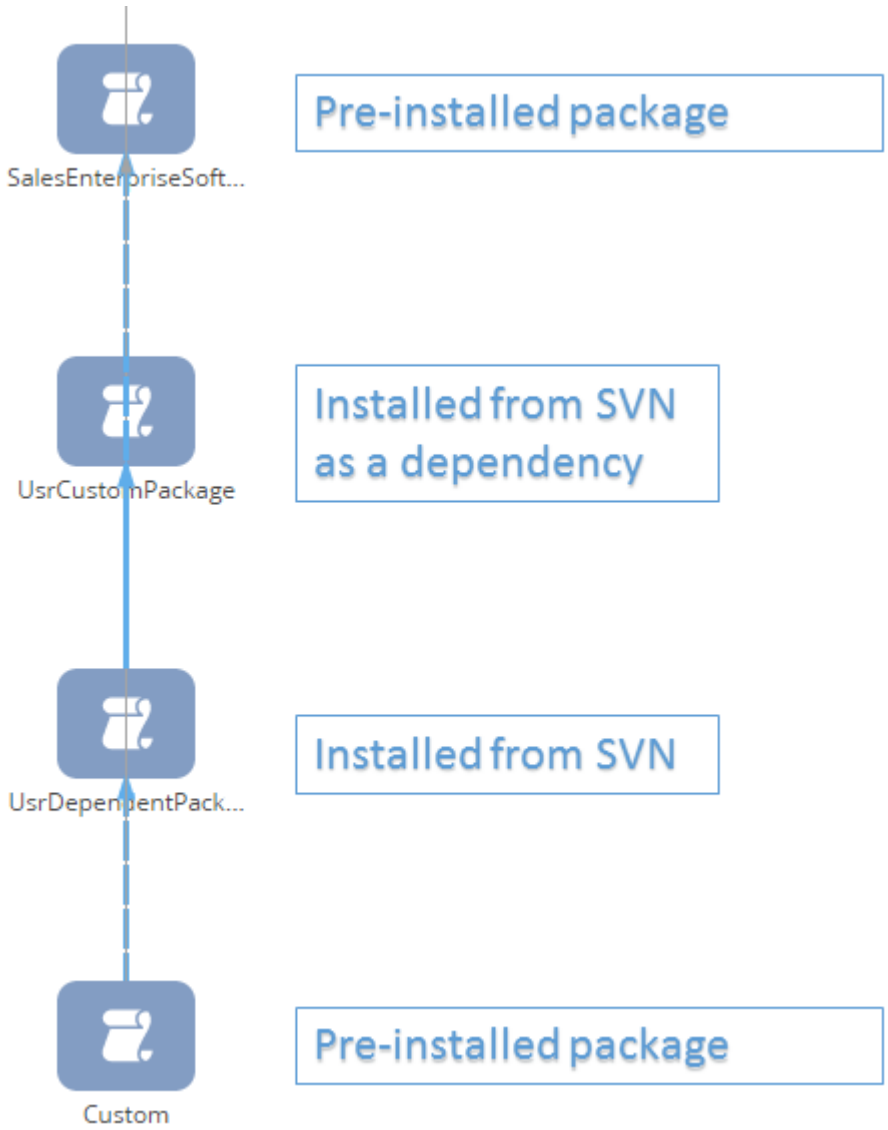
When you install a custom package, the system checks its dependencies and optionally installs or upgrades all the packages the current package depends on. For example, when you install the [UsrDependentPackage] package from the repository, the [UsrCustomPackage] package dependency will also be installed (Fig. 1).

Fig. 1. The [Changes] window after a package has been installed from SVN



This modifies the package hierarchy in the application (Fig. 2).

Fig. 2. 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": "974305e3-b01c-4020-b511-6fe9ac644d95",
    "PackageVersion": "7.8.0",
    "Name": "UsrDependentPackage",
    "ModifiedOnUtc": "\/Date(1478766323000)\/",
    "Maintainer": "Customer",
    "Description": "Example of package which depends on other packages",
    "DependsOn": [
      {
        "UId": "8bc92579-92ee-4ff2-8d44-1ca61542aa1b",
        "PackageVersion": "7.8.0",
        "Name": "UsrCustomPackage"
      },
      {
        "UId": "0ece3a34-413f-415b-919b-7b29bb31701f",
        "PackageVersion": "7.8.0",
        "Name": "SalesEnterpriseSoftkey_RUS"
```

```
        }
      ]
    }
}
```

2) Then the system checks whether the package dependencies are installed in the workspace. 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

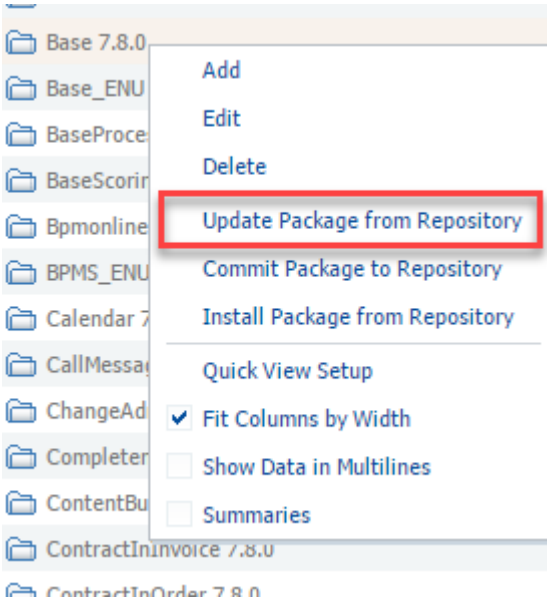
Introduction

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.

Installing packages from an application

Introduction

Since bpm'online version 7.10, packages can be uploaded and installed directly in the application interface. This enables you to install packages without using the SVN repository or the WorkspaceConsole utility.

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

Installing packages

To install packages from the application interface:

1. Go to the [System Designer] section and in the [Admin area] group, click the [Package installation] link (Fig. 1). A page for installing packages will open in a separate window (Fig. 2).

Fig. 1. System designer

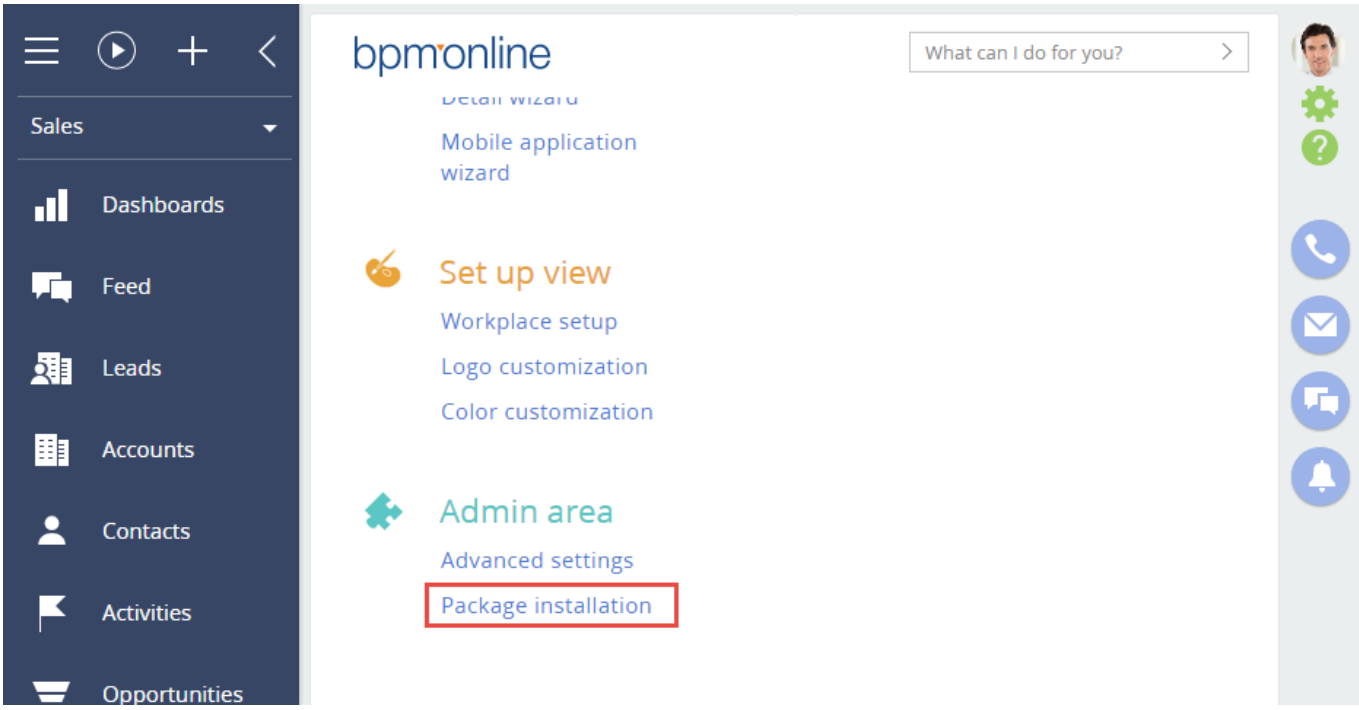
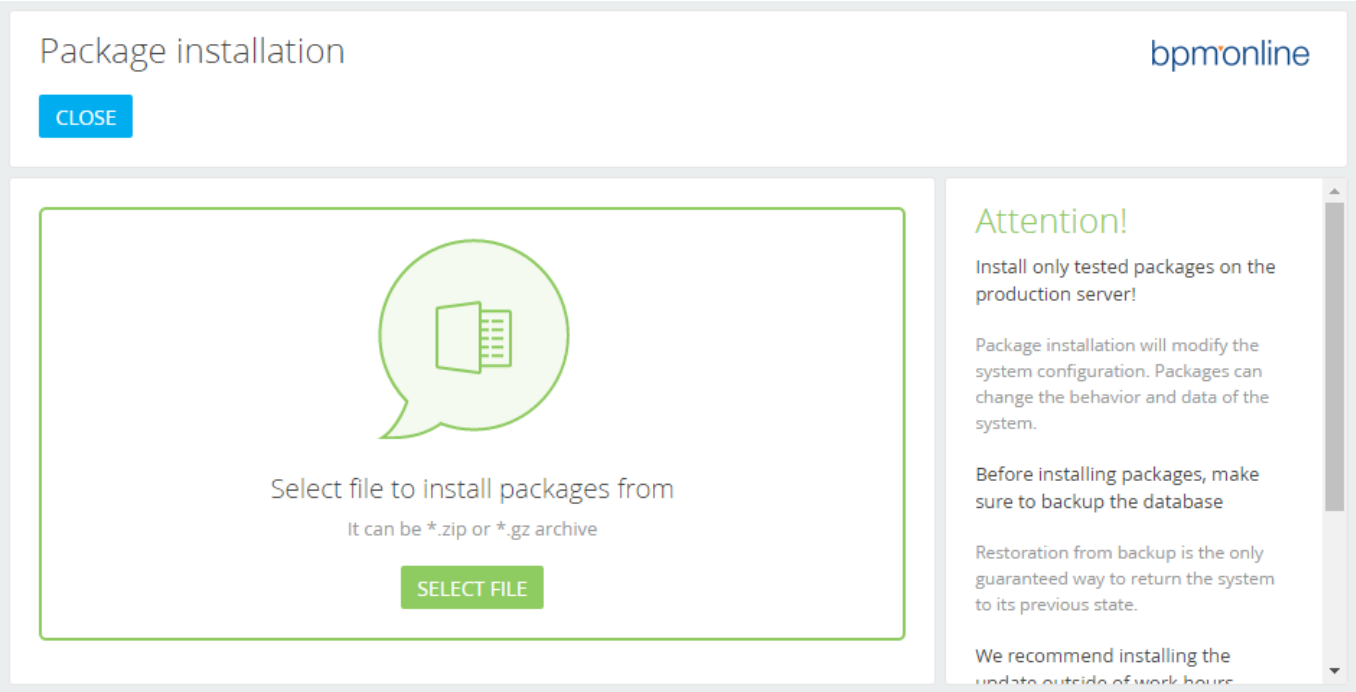


Fig. 2. Package installation page



2. Click the [Select file] button, select the package file (*.gz), or select several packages in a *.zip-archive (Fig. 3). The system will be backed up (Fig. 4) and the package(s) will be installed (Fig. 5).

ATTENTION

The interface uses package archives (*.gz), which can be downloaded **from the database** or from **the SVN repository** only using the WorkspaceConsole utility (see: "**Working with WorkspaceConsole**").

To install several *.gz-packages at once, add them to a *.zip archive, and select it on the package installation page. The application will automatically unpack the *.zip archive and install the packages.

Fig. 3. Selecting a package for import

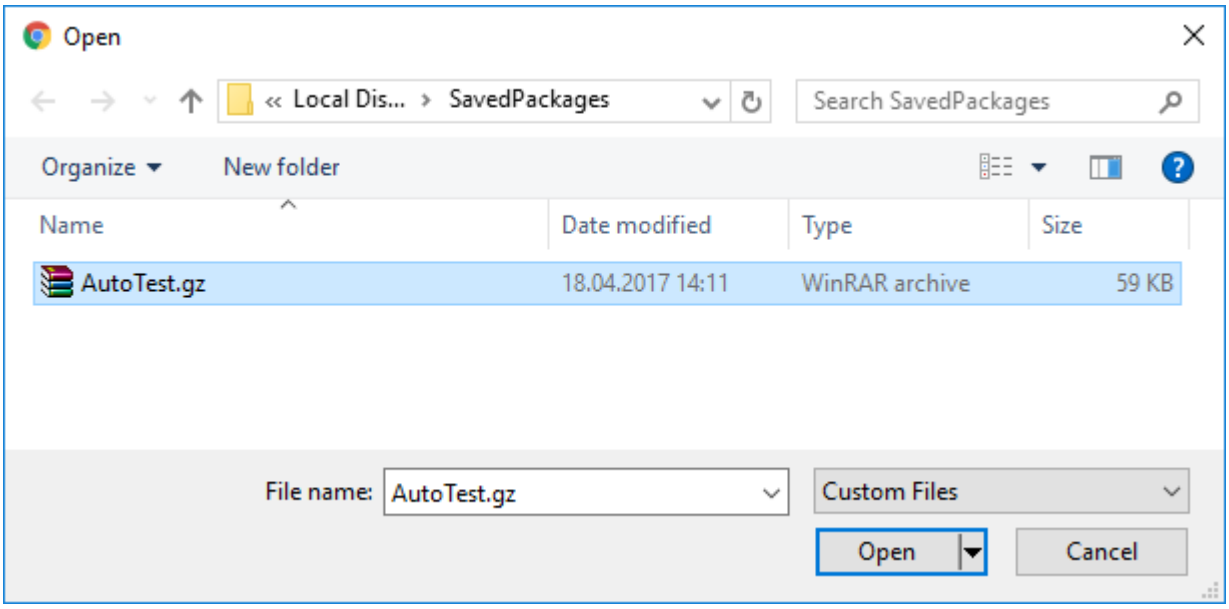


Fig. 4. System backup

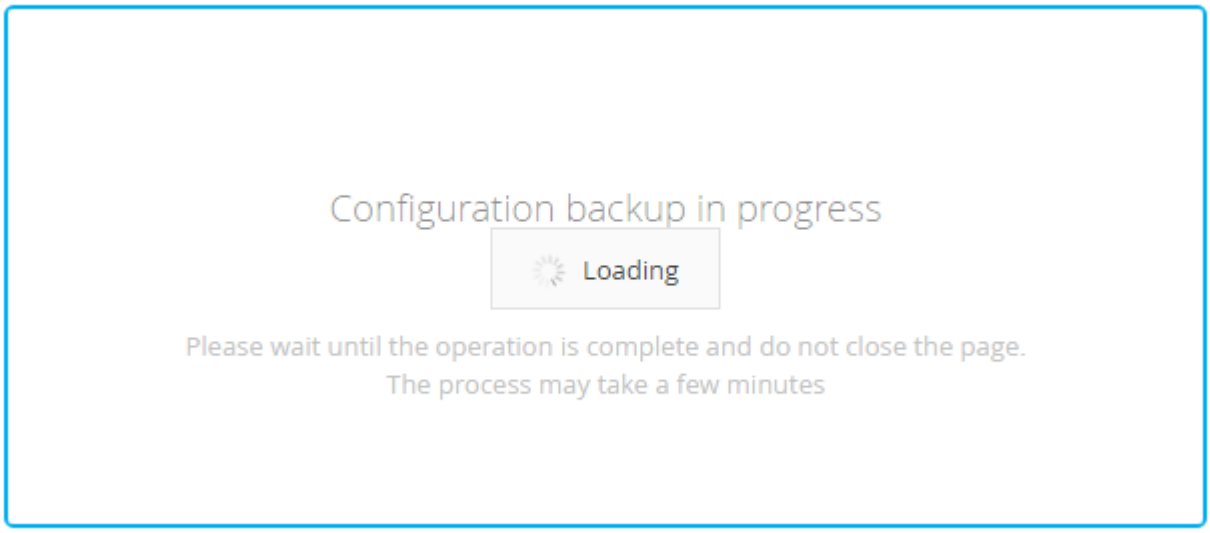
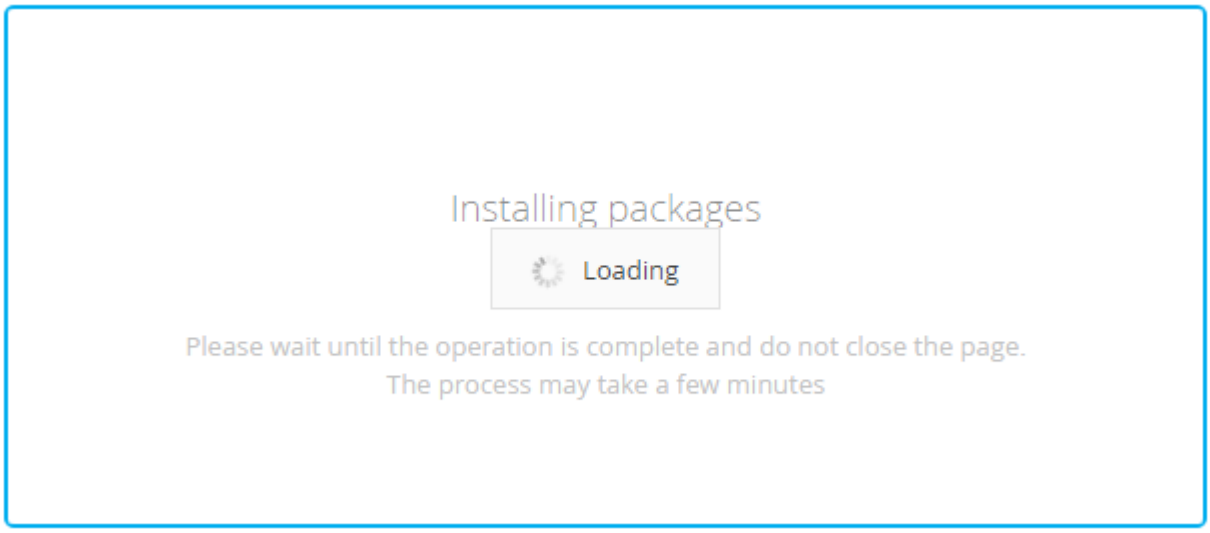


Fig. 5. Installing a package



A corresponding message will be displayed (Fig. 6).

Fig. 6. Successful package installation message

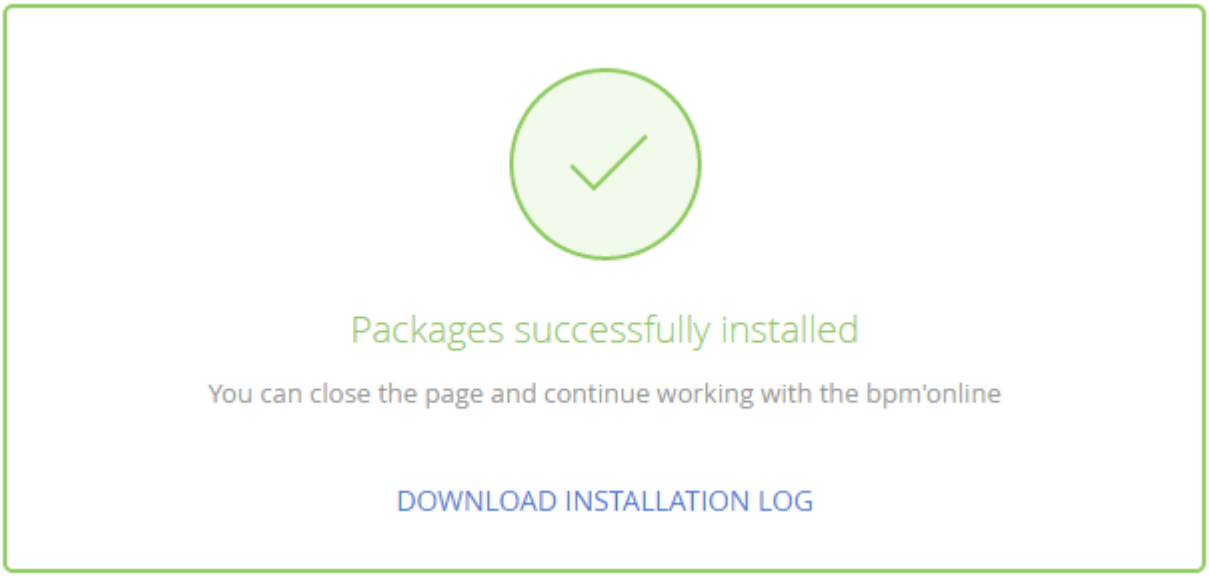
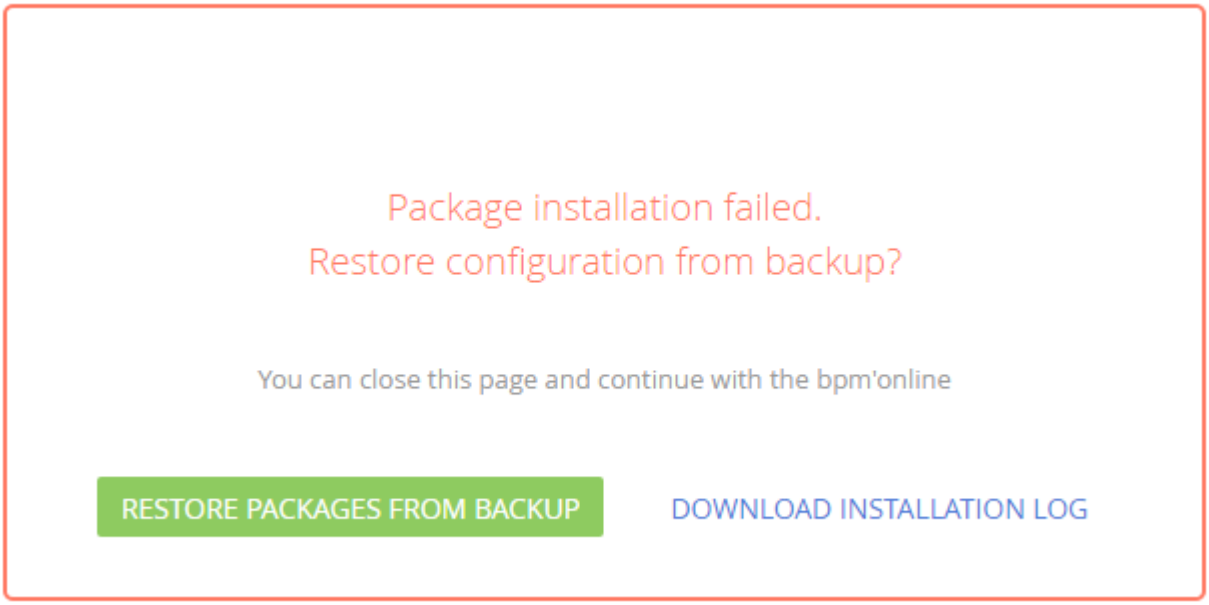


Fig. 7. Unsuccessful package installation message



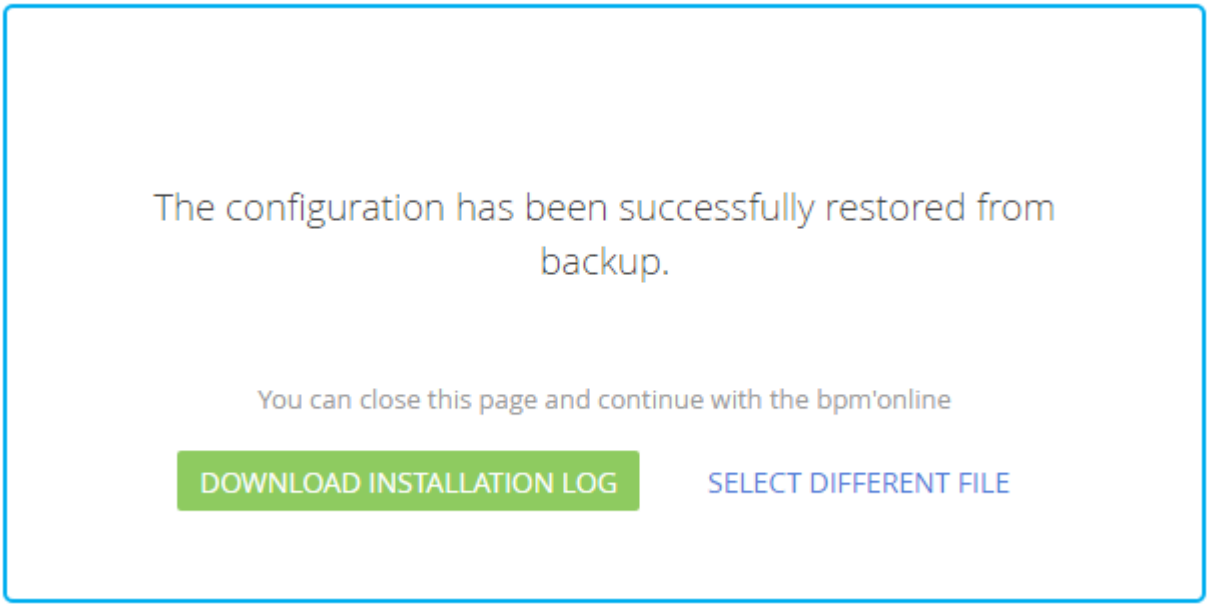
After the installation is complete, you can download the log file by clicking the [Download Installation Log] button.

> ATTENTION
>
> You cannot create 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.

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. 7). After the backup is restored, a corresponding message will be displayed (Fig. 8). You can select another package for import.

Fig. 8. Restoring a backup copy message



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

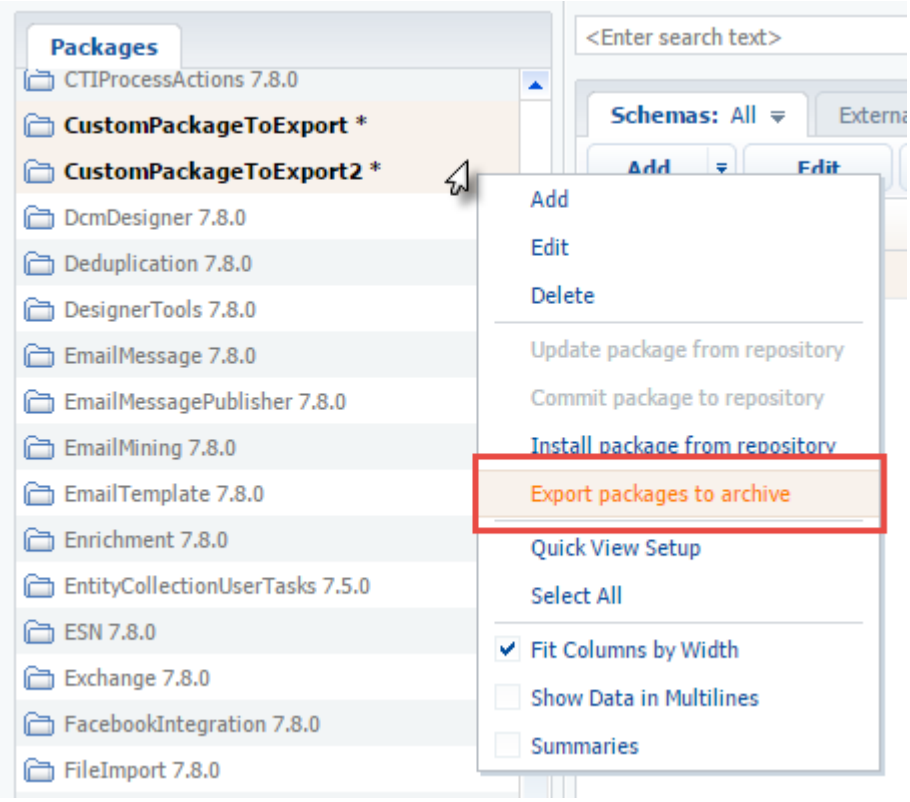
To install packages from the application interface:

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

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

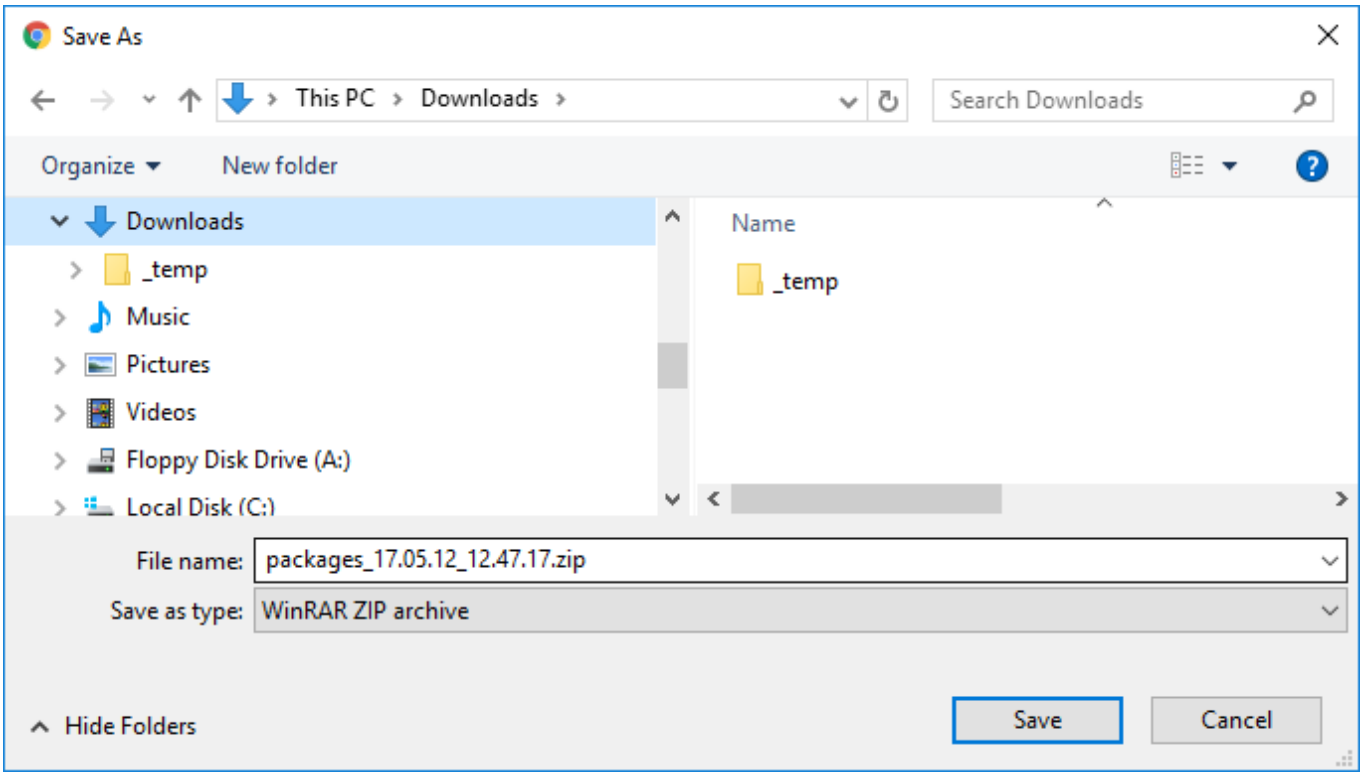
3. Trigger the [Export packages to archive] action (Fig. 1).

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



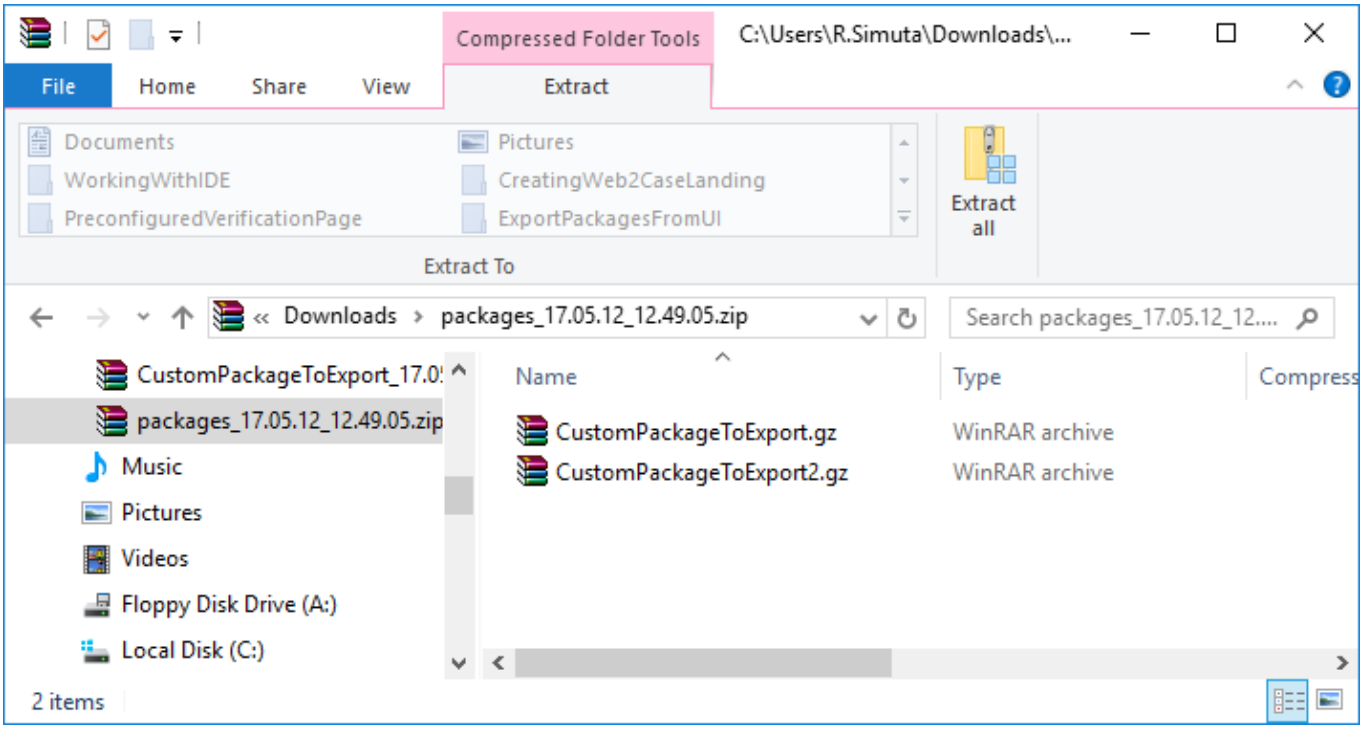
Depending on the browser settings, the zip-archive with packages will either be saved to the default downloads folder, or the browser will display a dialog box for selecting a folder for the archive (Fig. 2).

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 packages from an application**").

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

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!

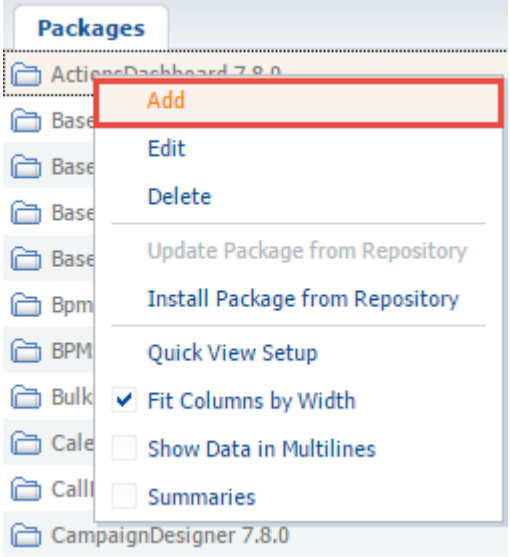
Ensure that **application is configured to access the SVN repository** before binding the package to the SVN repository when development mode in the file system is enabled.

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).

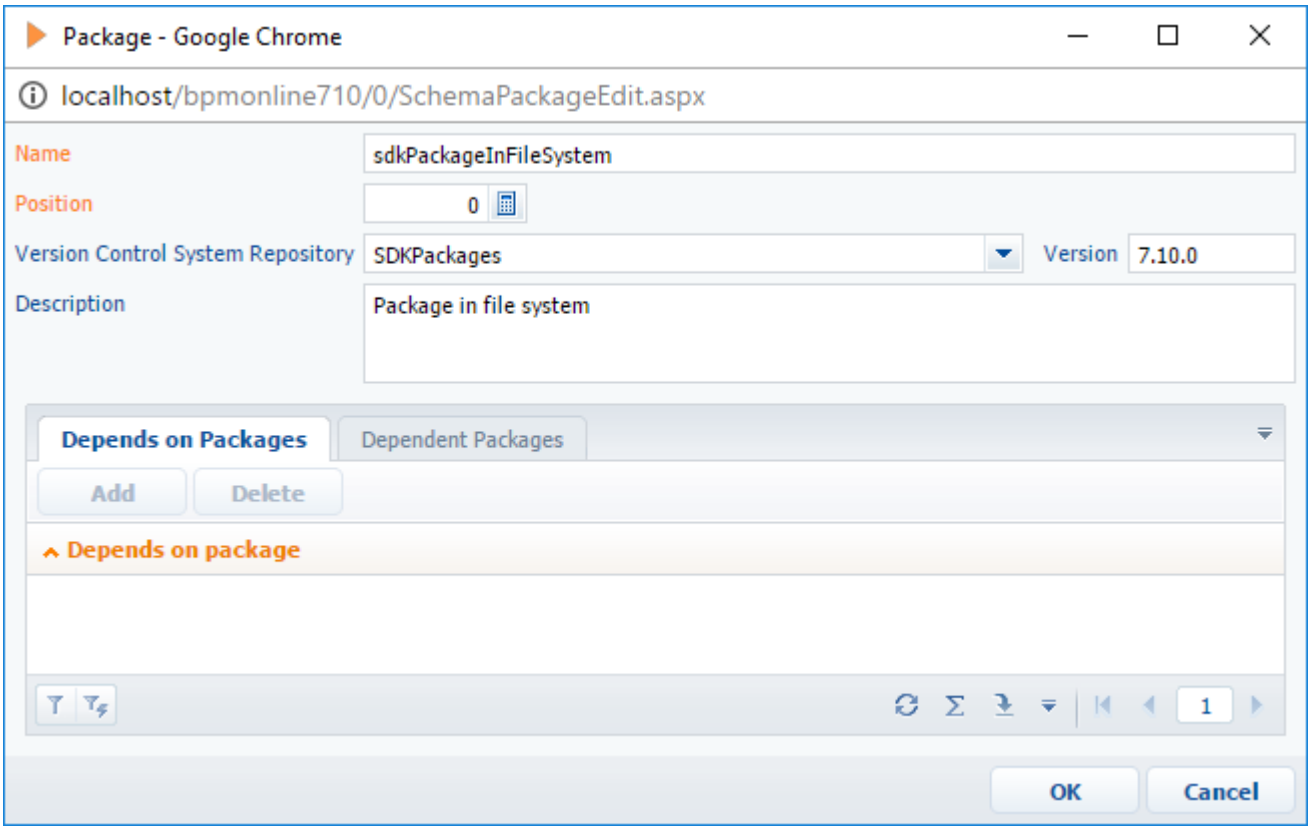
Fig. 1 Adding a package in the [Configuration] section



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.

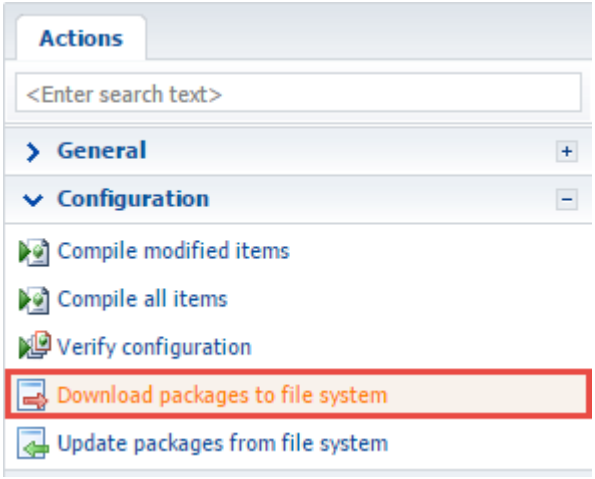
Fig. 2 Package summary



2. Download created package to file system

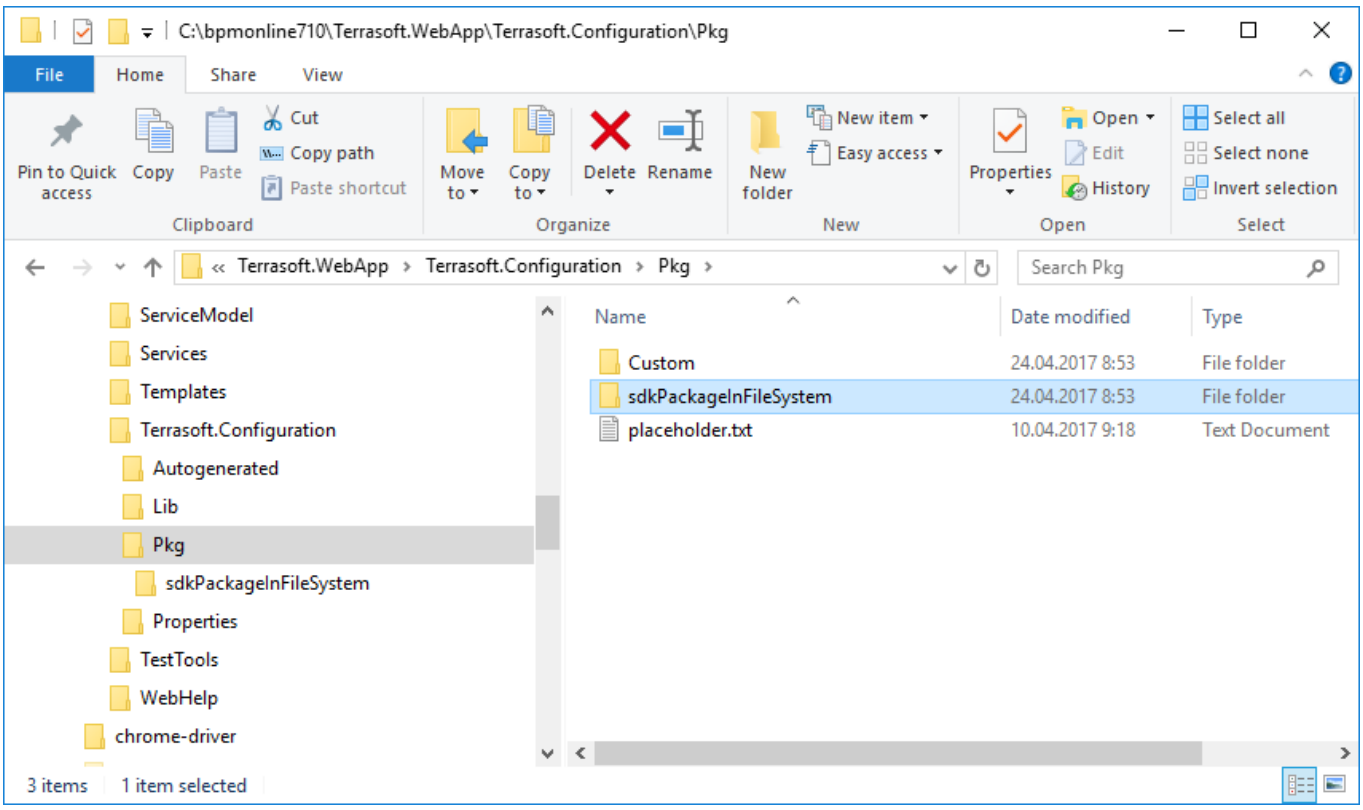
Click [Download packages to file system] (Fig. 3).

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.

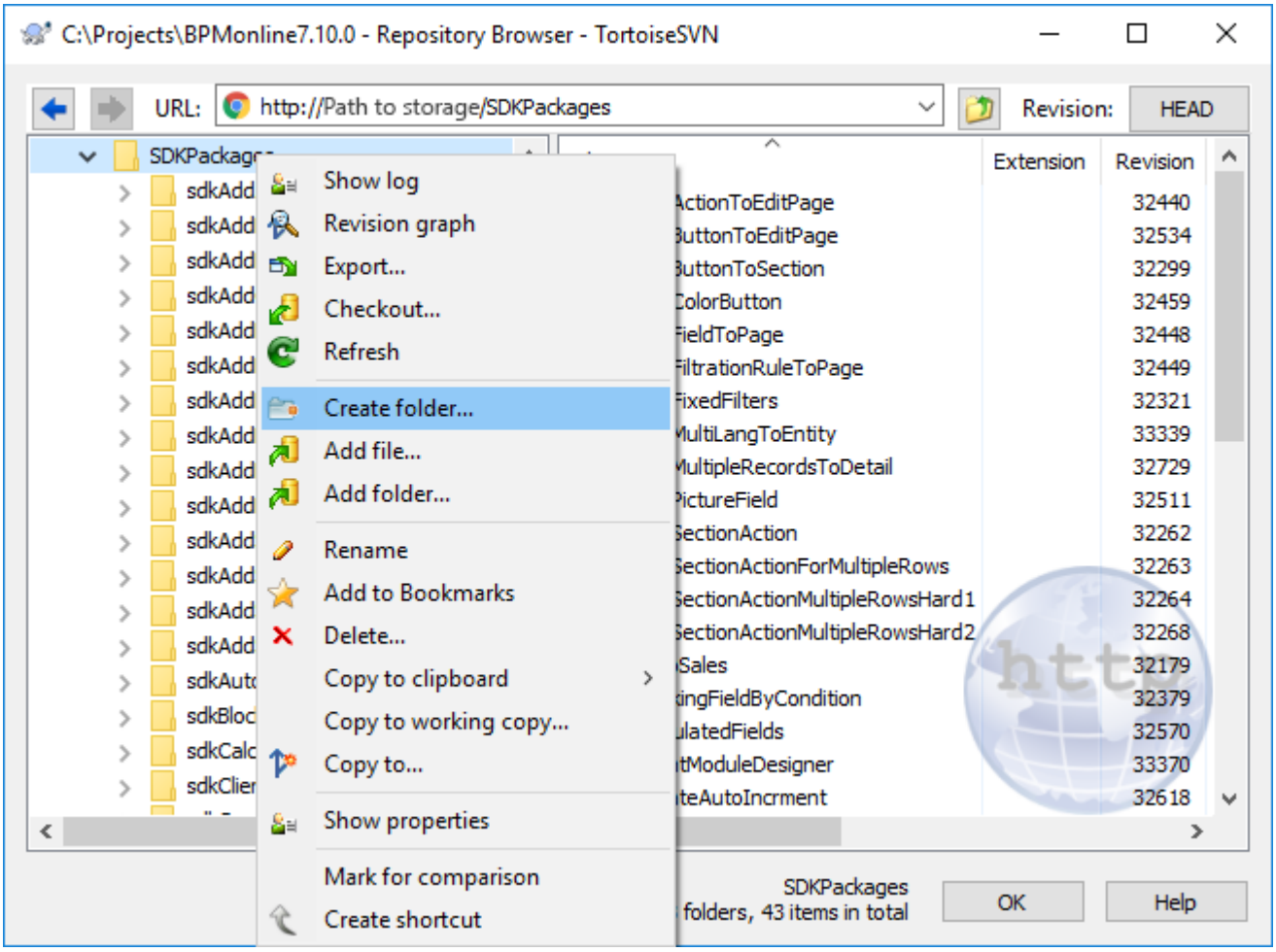
3. Create necessary folders for the packages in the SVN repository

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.

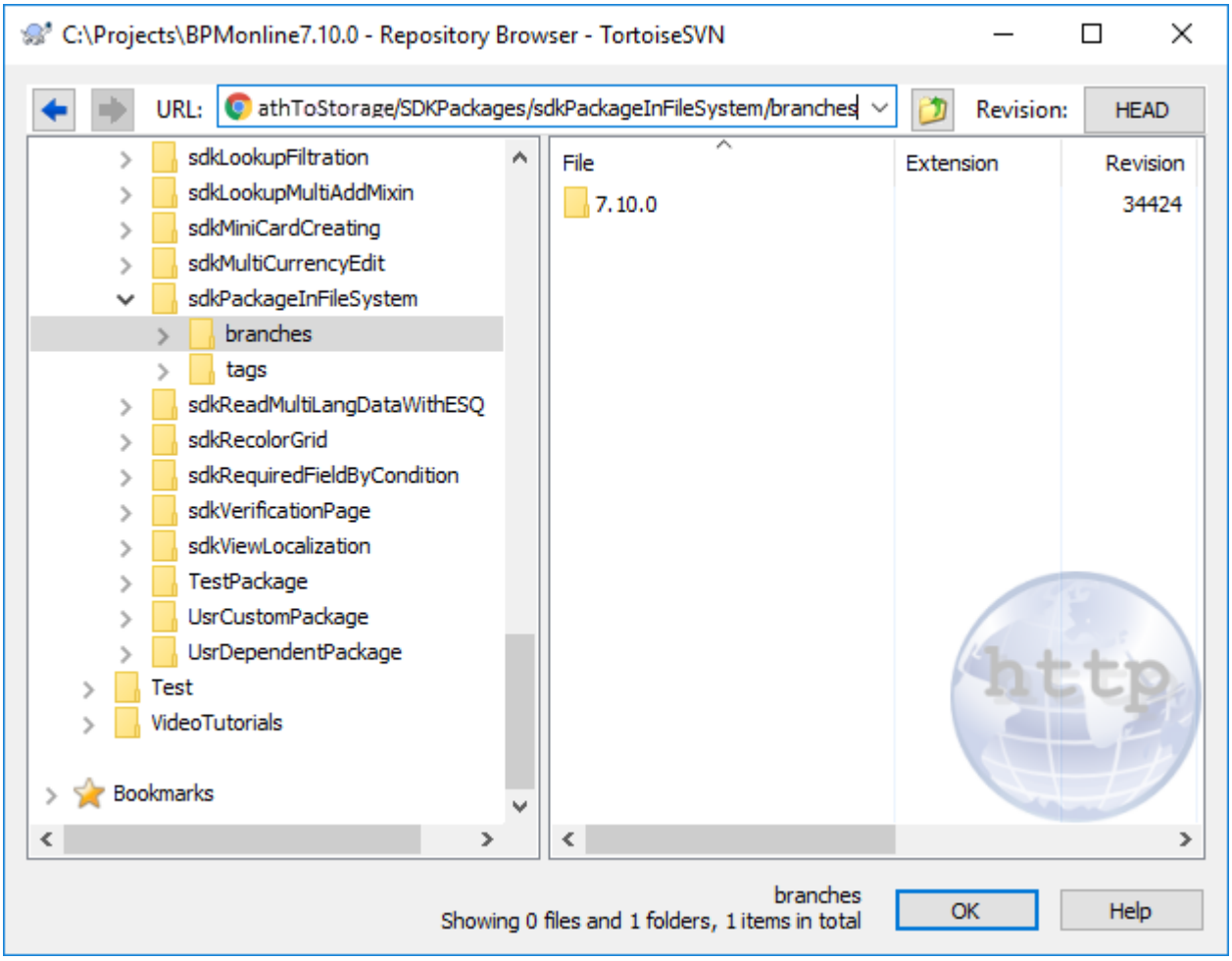
> Attention!
>
> This is a brief example of working with SVN via TortoiseSvn. More information about working with SVN repository via TortoiseSvn can be found in the documentation.

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

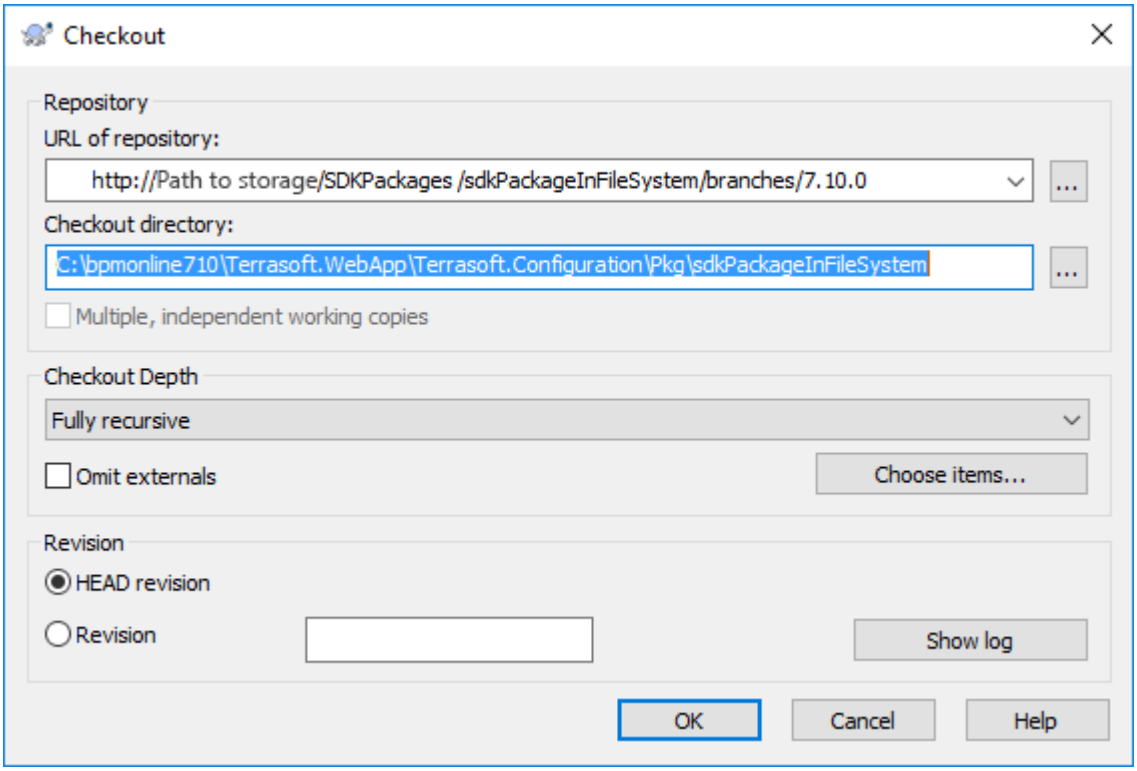
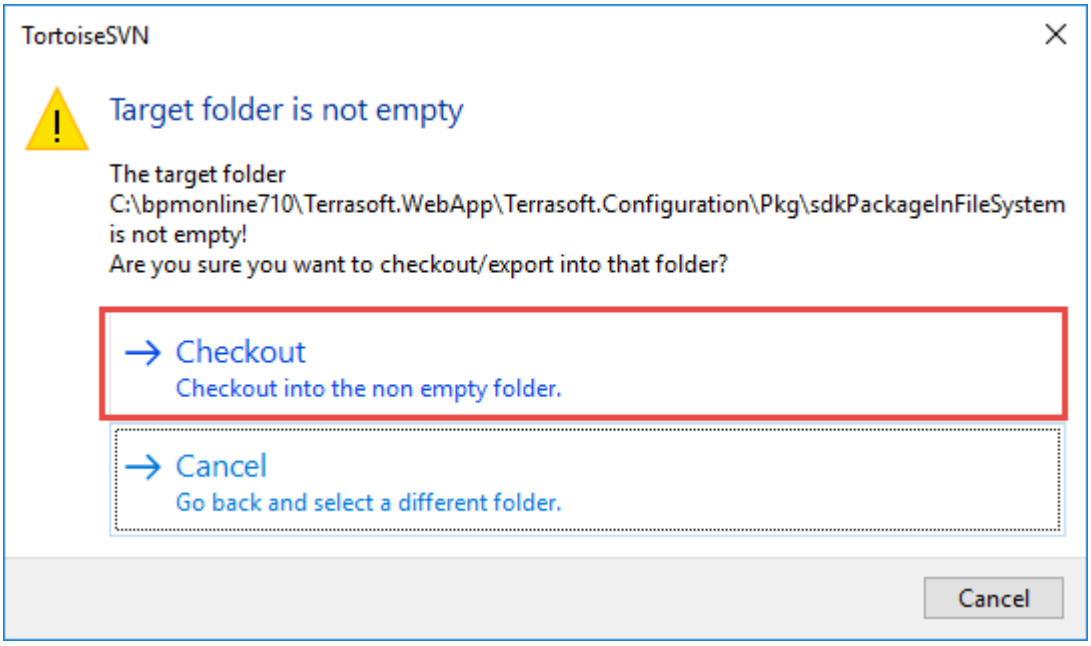
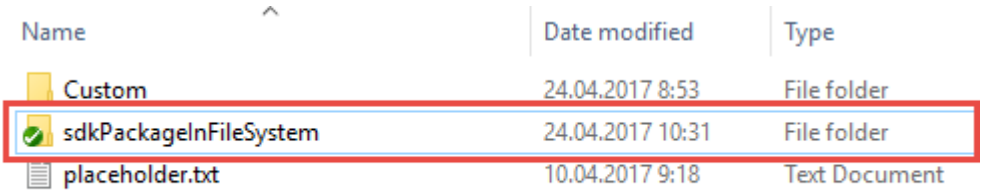


Fig. 8 Confirmation of the SVN checkout operation to the existing folder.



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

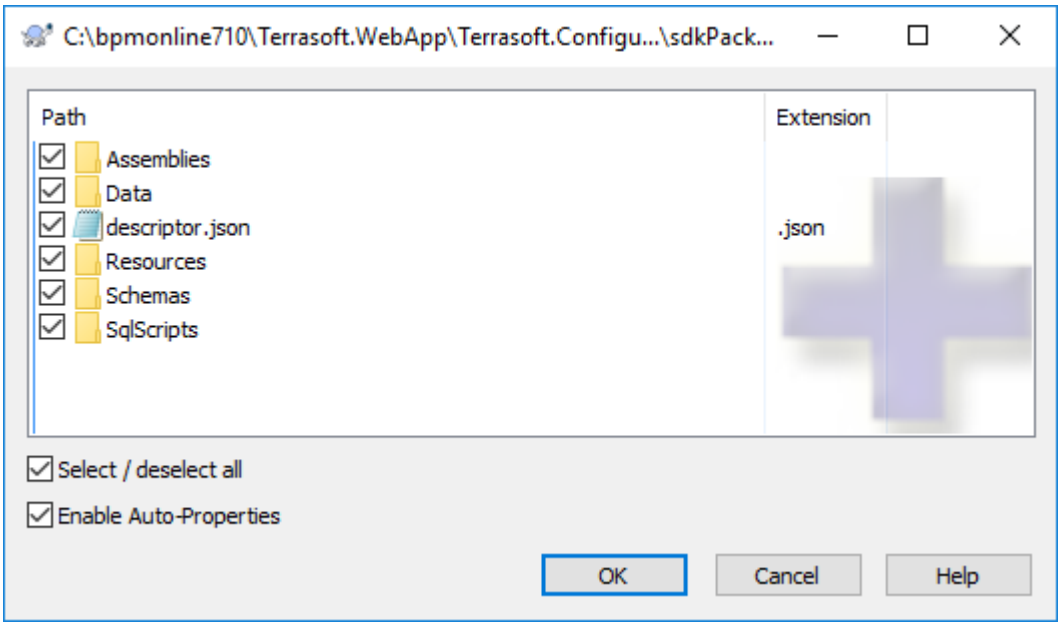
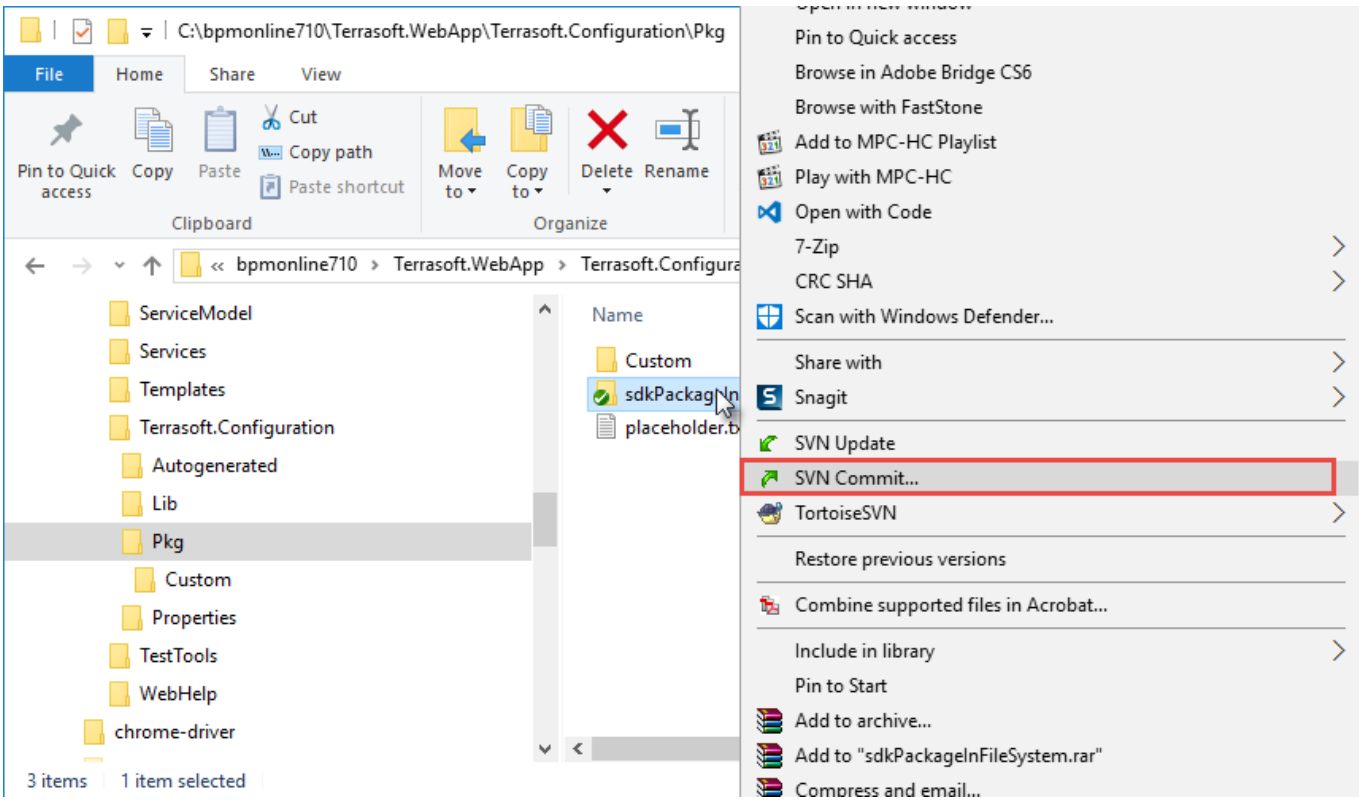


Fig. 11 Committing changes to the repository



Transferring changes between the working environments

It often becomes necessary to transfer individual changes between the **working environments** 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:

- 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.

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:

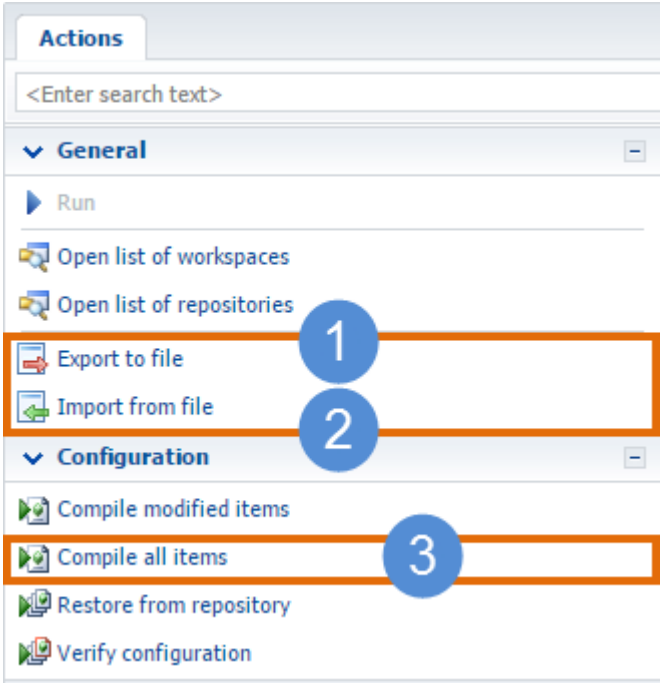
- The mechanism enables you to export and import only schemas. You cannot export or import packages. In addition, it is not possible to transfer data connected to the packages.
- Administrator access rights in the application are required.
- It is not possible to load several schemas at the same time.
- The schema export/import mechanism interfered with the package versioning, which can make it difficult to identify the source of errors.

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



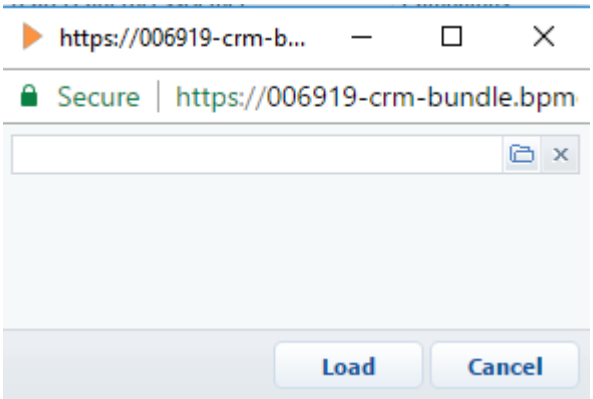
A file with the schema name and *.md extension saved on your hard drive as a result.

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

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.

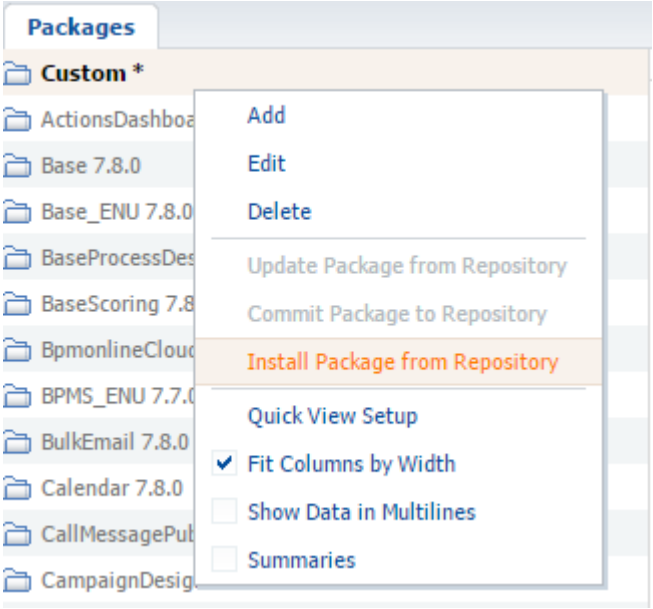
5. Install the package from the repository

> 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 "**General information about the [Configuration] section**" article.

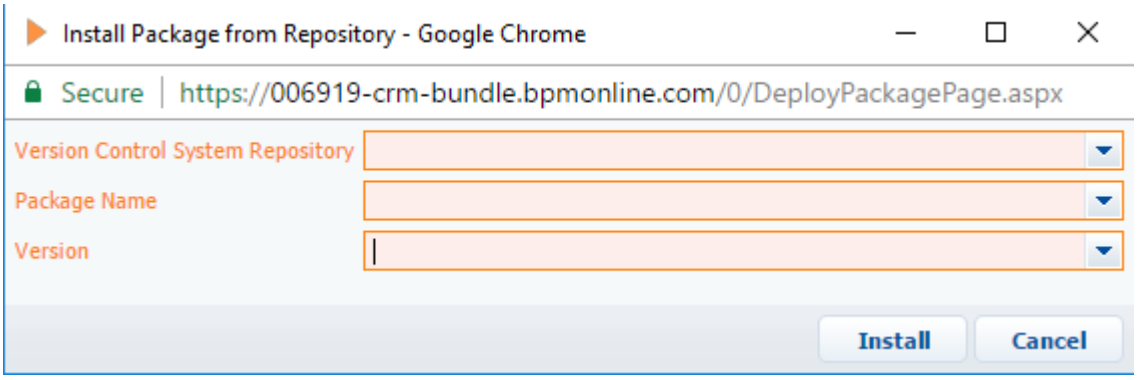
To install the package from the repository, go to the [Configuration] section, right-click the [Packages] tab and select the [Install Package from Repository] option (Fig. 1).

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 (see step 2), 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.

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.

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

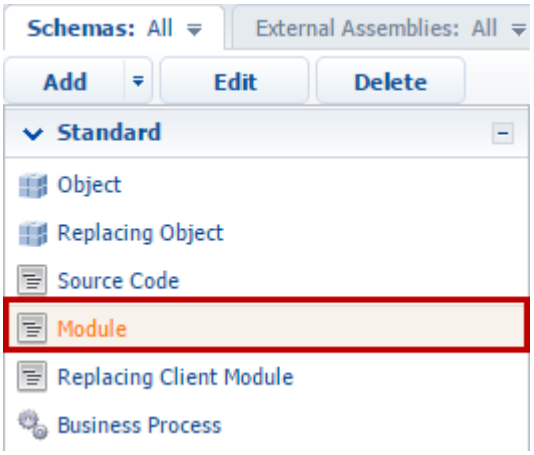
Non-visual modules represent system functionality that is not associated with data binding or data display in the UI. Examples of non-visual modules in the system are business rule modules (BuisnessRuleModule) and utility modules that implement utility functions.

To create a non-visual module schema:

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

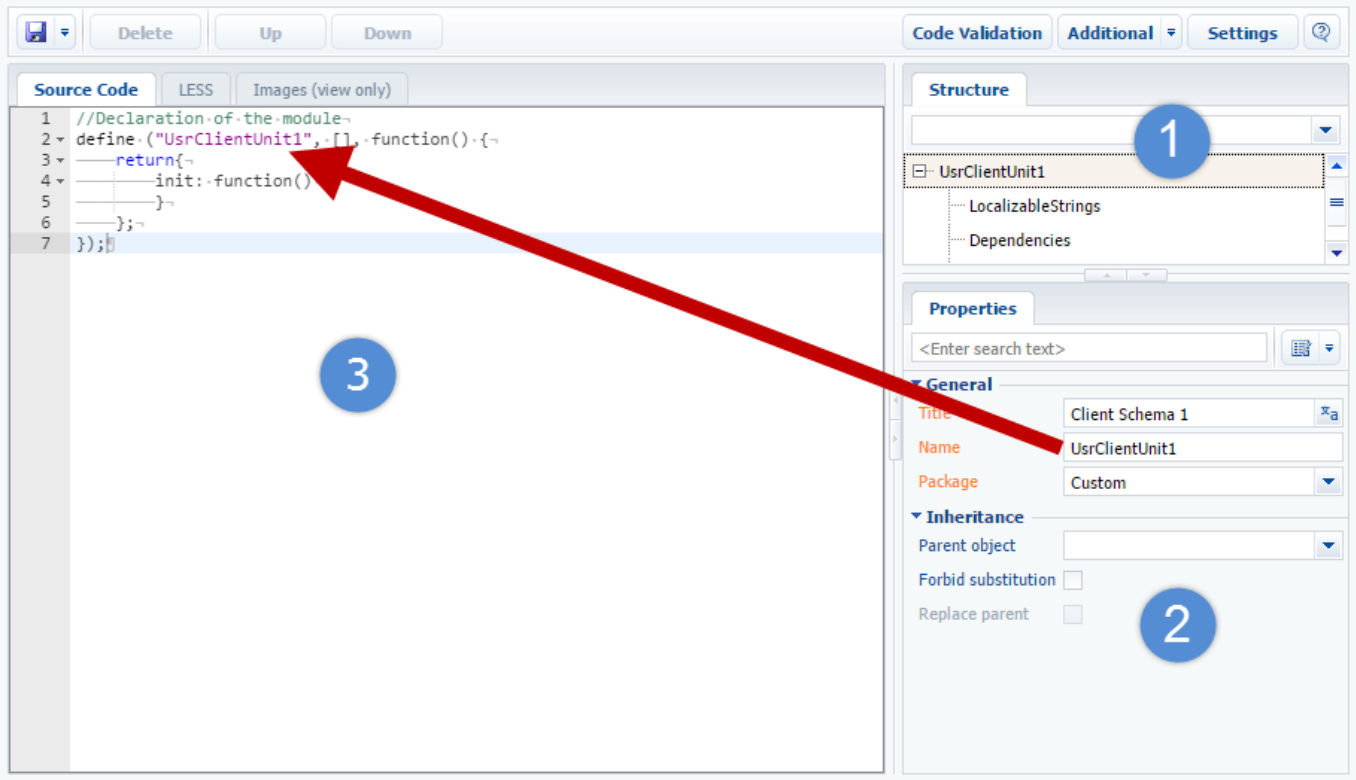
2. On the [Schemas] tab, select [Add] > [Module] (Fig. 1).

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:

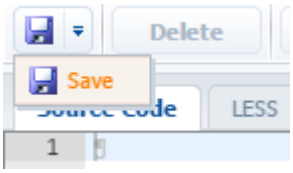
- [Name] – schema name. May contain only Latin characters and numbers. Contains the prefix specified in the [Object name prefix] system setting (SchemaNamePrefix).

- [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.

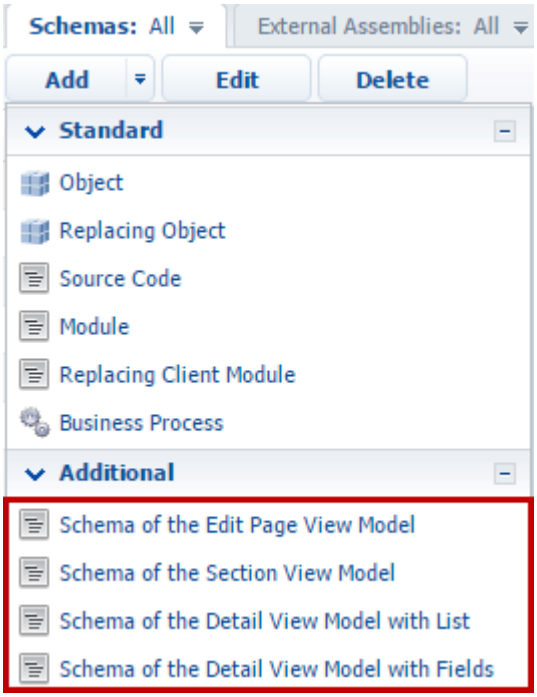
Custom view model schema is a visual module schema. It is a configuration object for generating views and view models by the *ViewGenerator* and *ViewModelGenerator* generators. For more information about the schema structure, please see the "**Client view model schemas**" article.

To create a schema of the visual module:

1. Go to [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

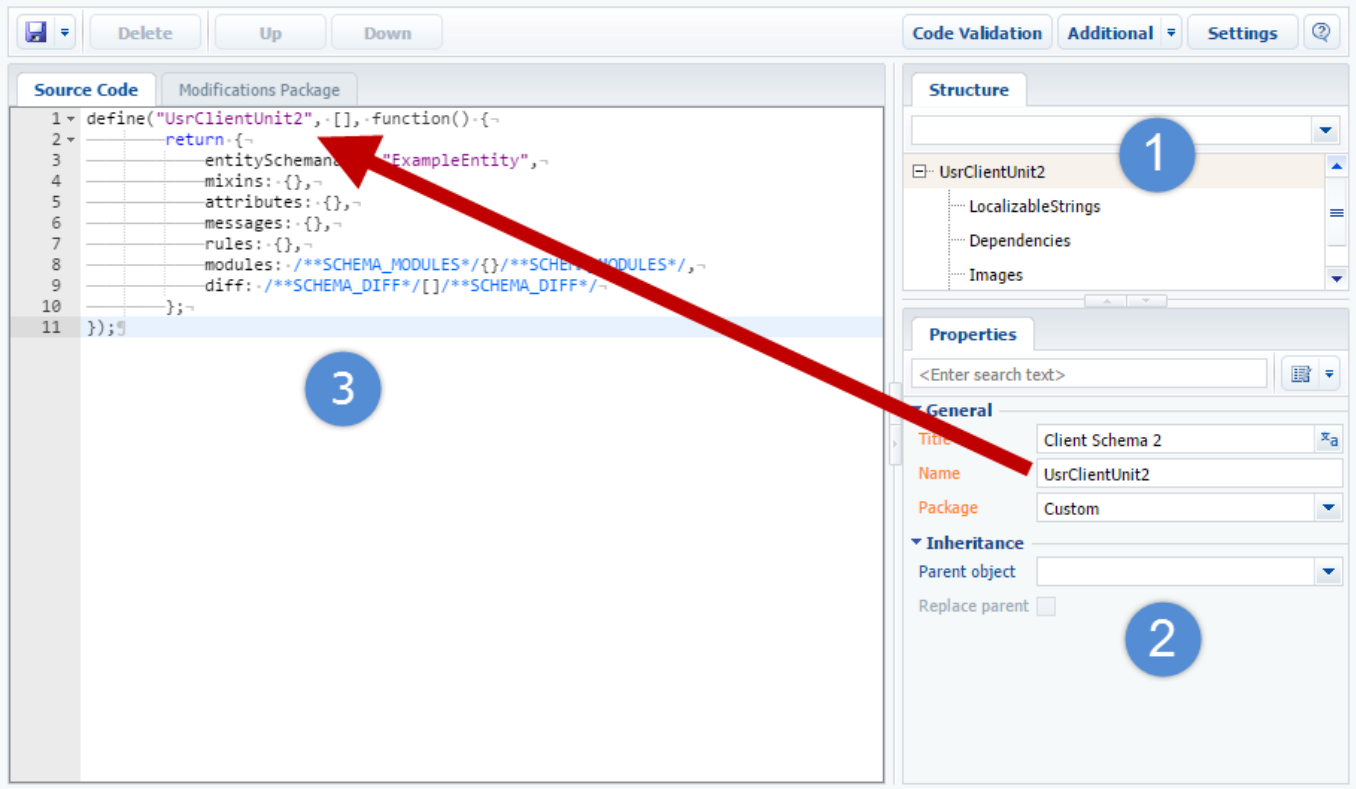


You can add the following types of visual module schemas:

- [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

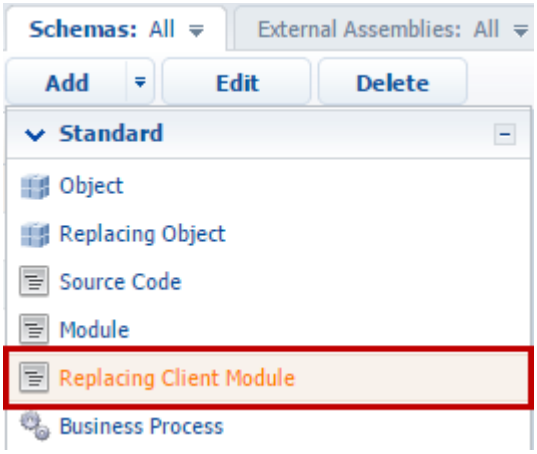
Replacing schemas are used to extend the functions of the already existing schemas. The existing schemas also may be replacing and belong to different packages.

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

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

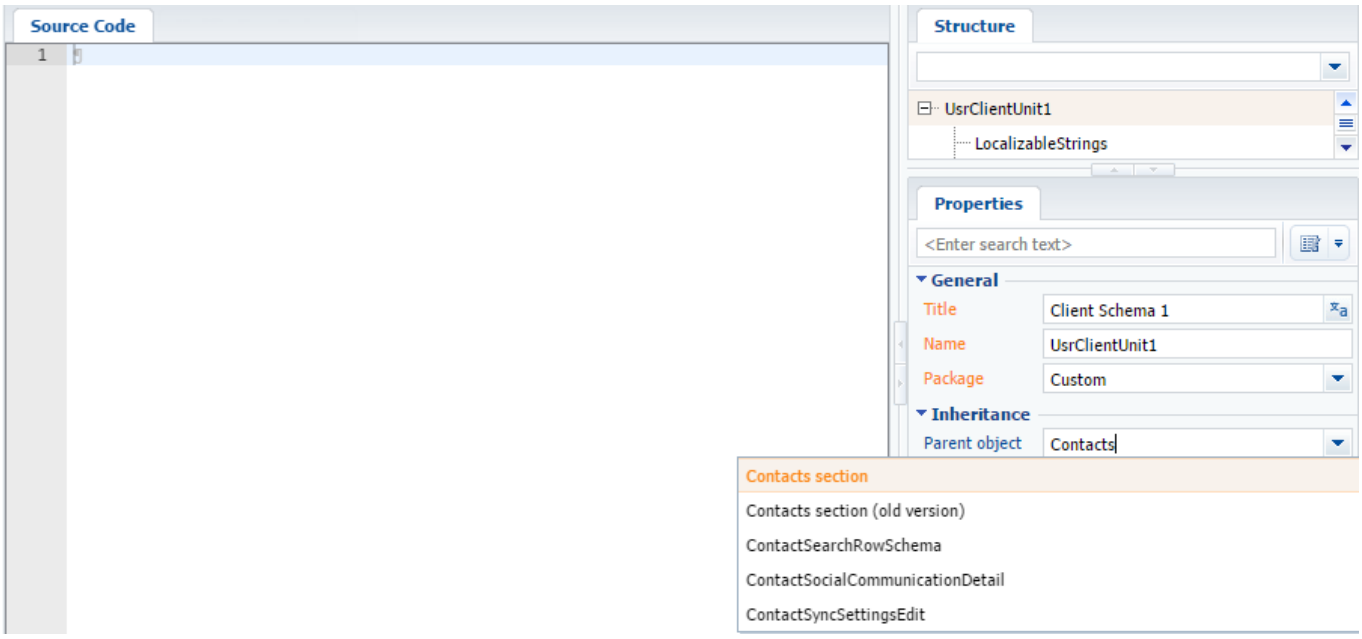
2. On the [Schemas] tab, select [Add] > [Replacing client module] (Fig. 6).

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

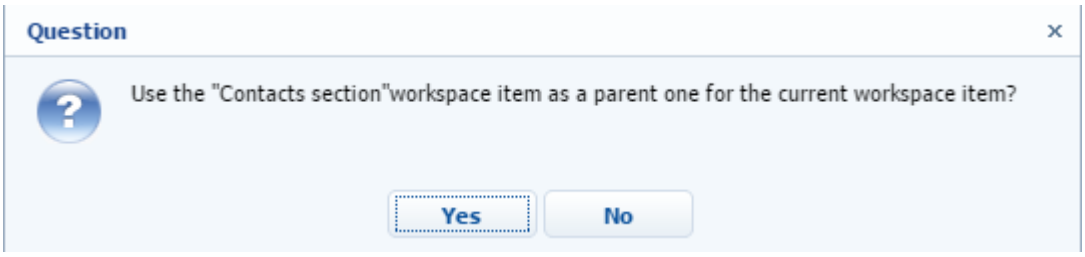
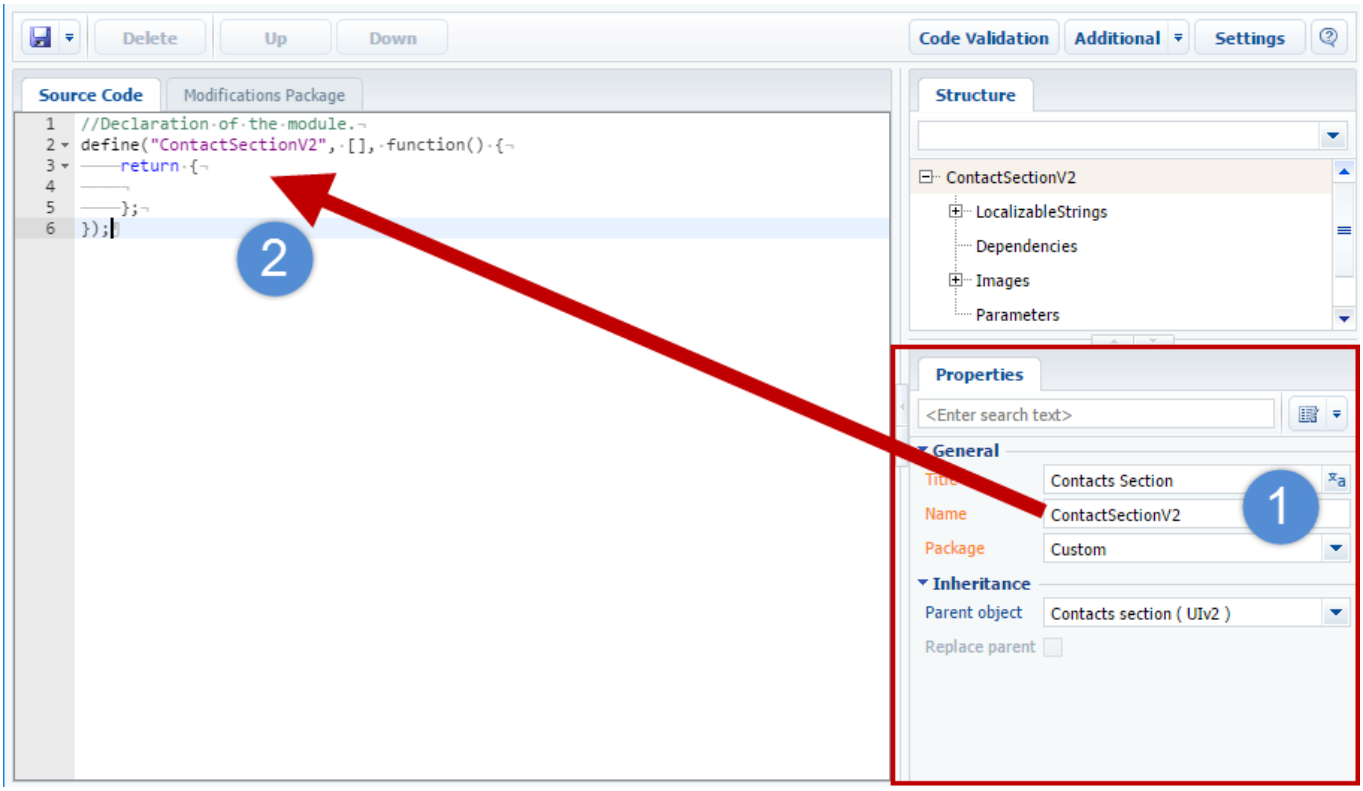


Fig. 9. A replacement client schema in the client schema designer



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

Access to the data in bpm'online is done through the ORM system, which ensures work of BPM and DCM systems and the access rights system.

The ORM objects in bpm'online are based on objects ("Entity"). 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.

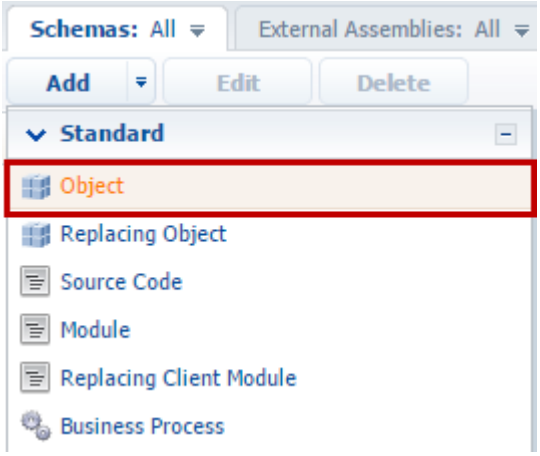
Custom entity schemas can be created as part of configuration and placed in custom packages. Base schemas can be replaced by custom schemas.

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



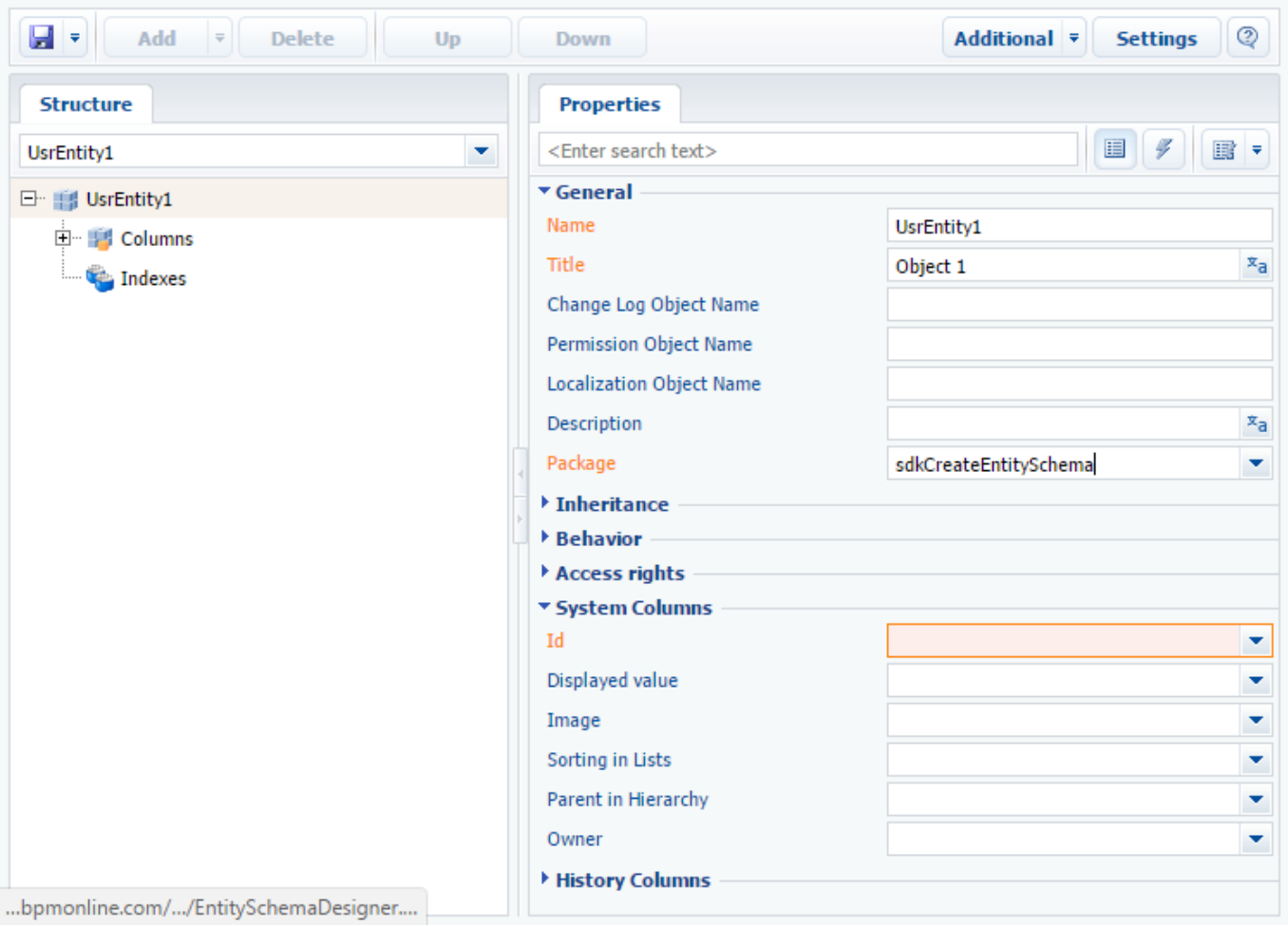
You need to assign values to the following required properties for the created entity schema (Fig. 2):

- [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

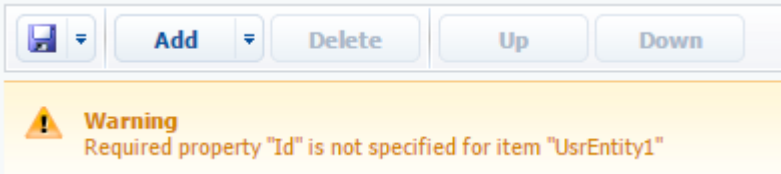


NOTE

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.

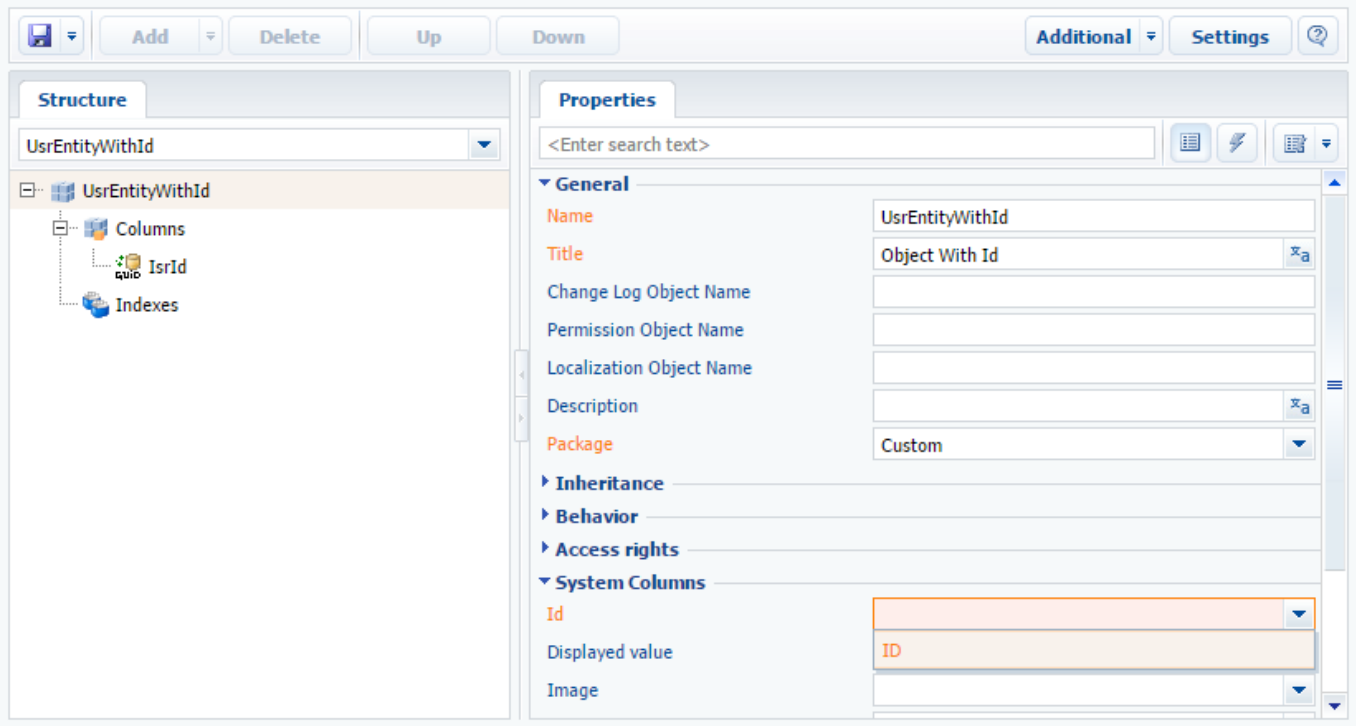
The entity is the representation of the table record in the database and it must have the id column used as a table primary key. If you try to save an object schema without an Id, a warning will pop up (Fig. 3).

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

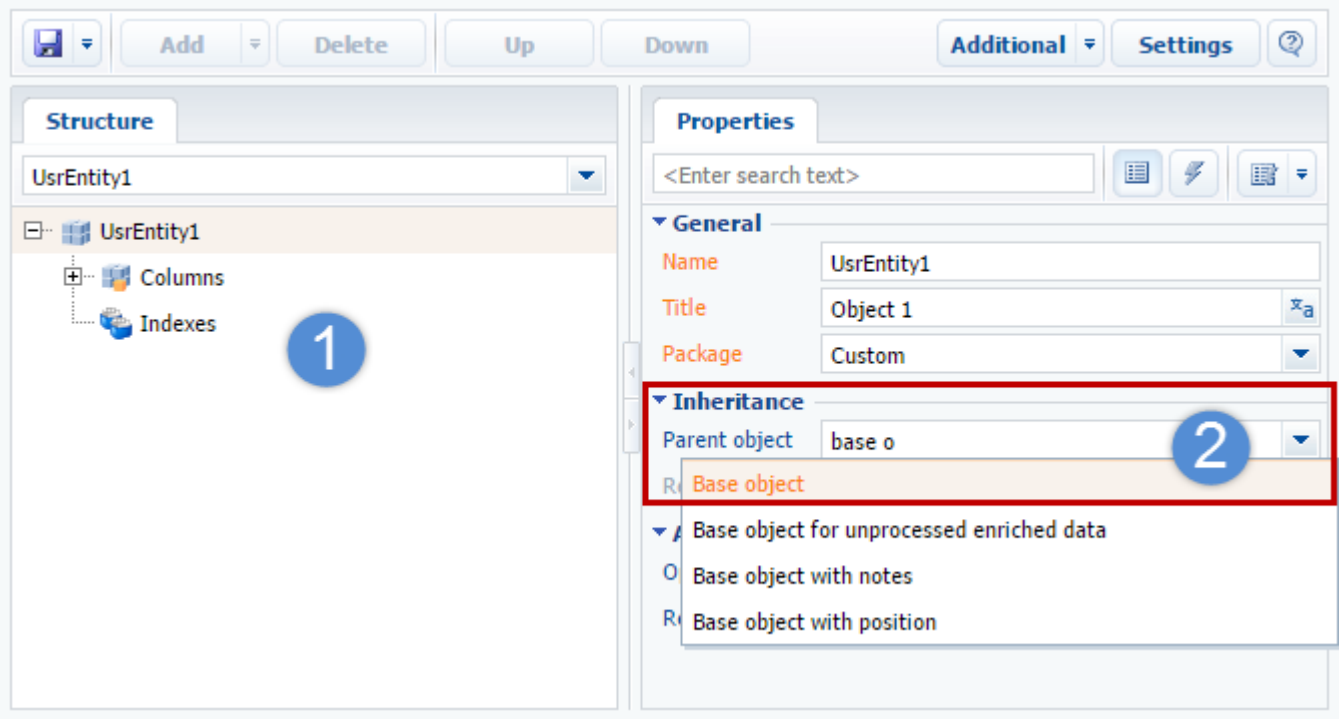
The procedure for adding the column is covered in the “Adding the custom column to the entity” section.

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

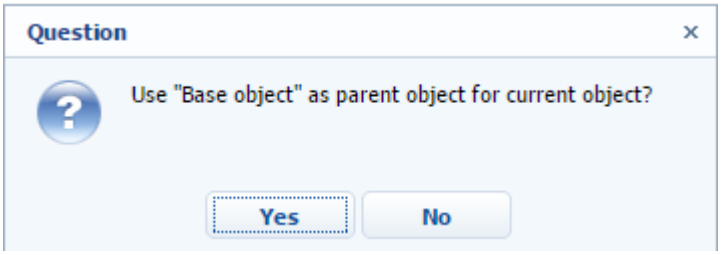
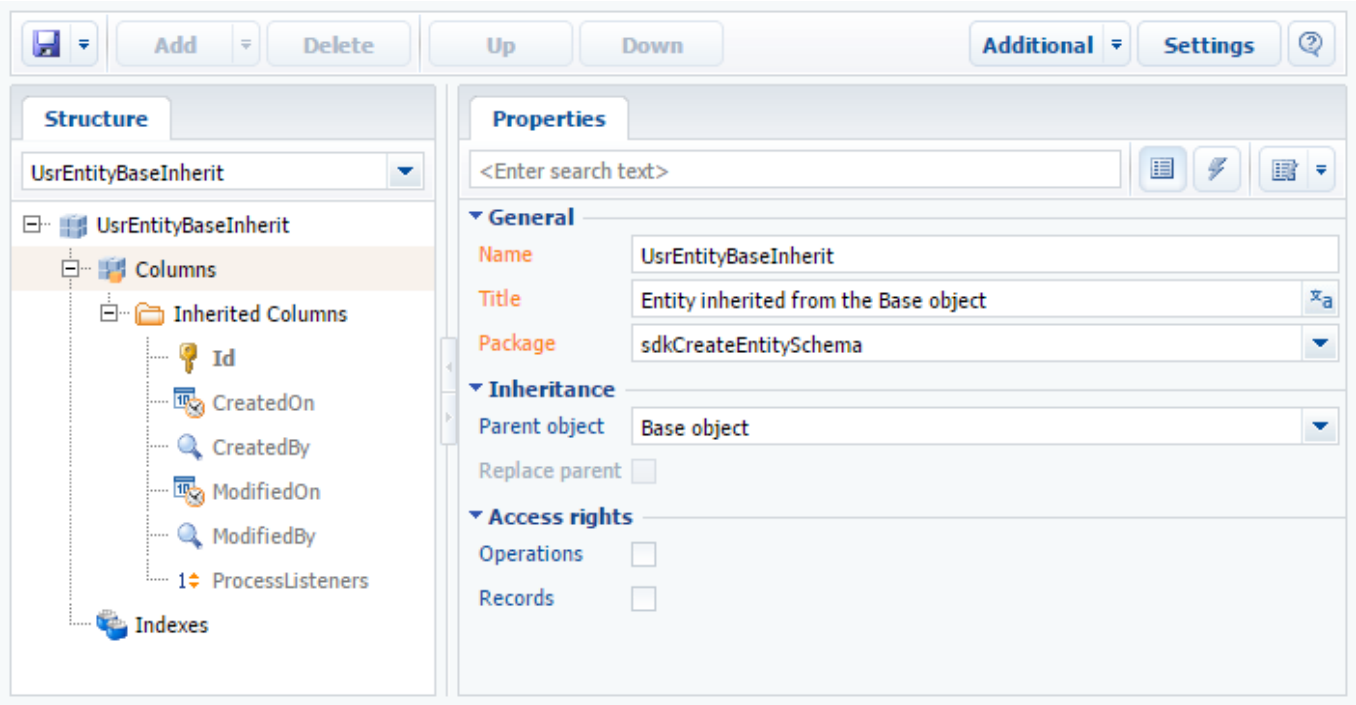


Fig. 7. Columns inherited in the entity structure



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.

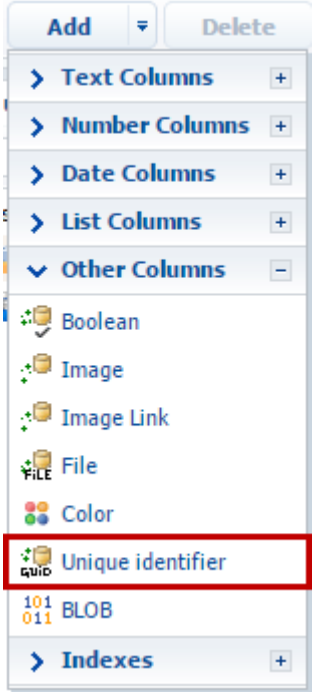
Save the object schema to preserve the metadata changes. Publish the schema to create the corresponding table in the database and make the changes available to bpm'online users.

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

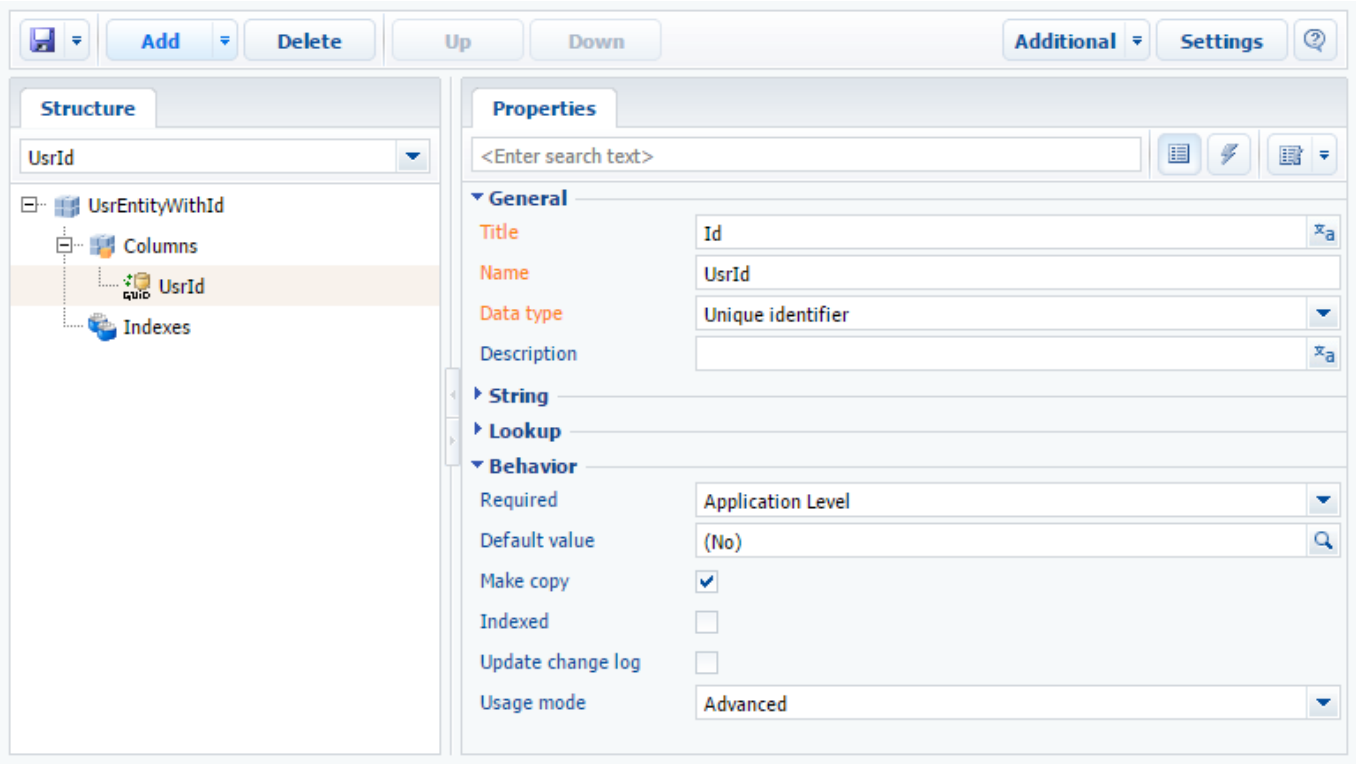
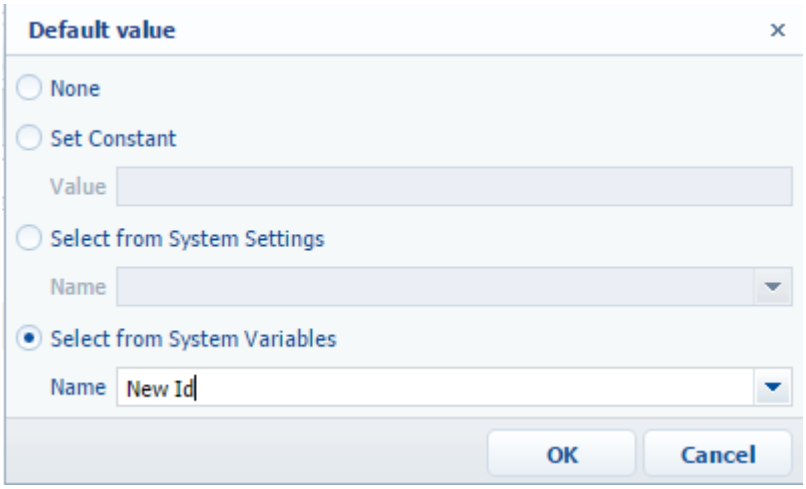


Fig. 10. Setting the default value



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

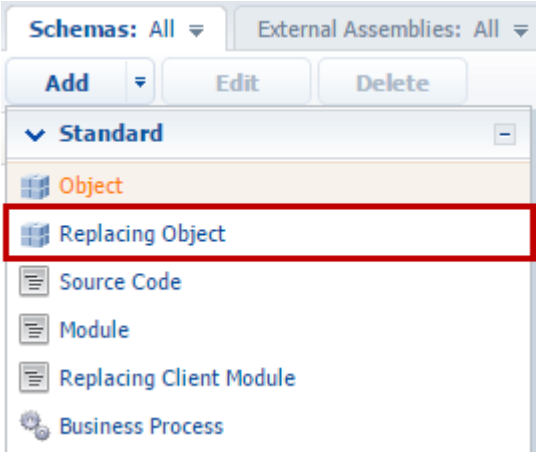
[Indexes] element. Select the object column and specify the sorting direction for the added indexed column.

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).

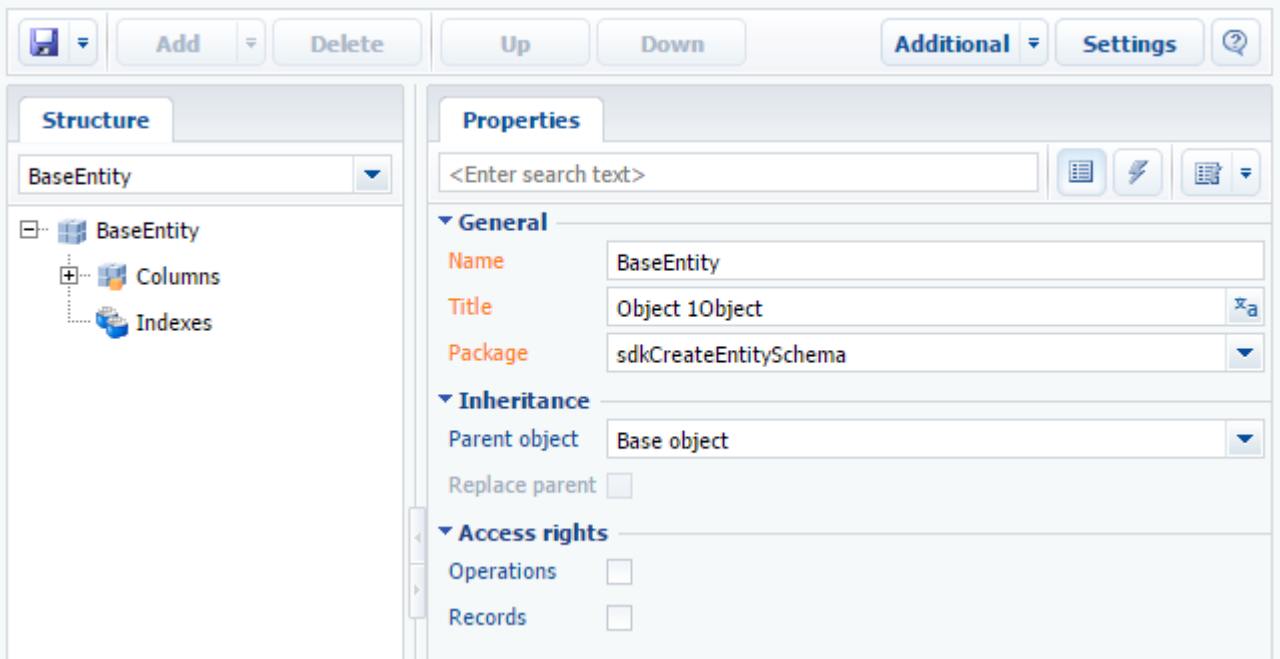
Fig. 11. Command for creating a replacing object schema



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.

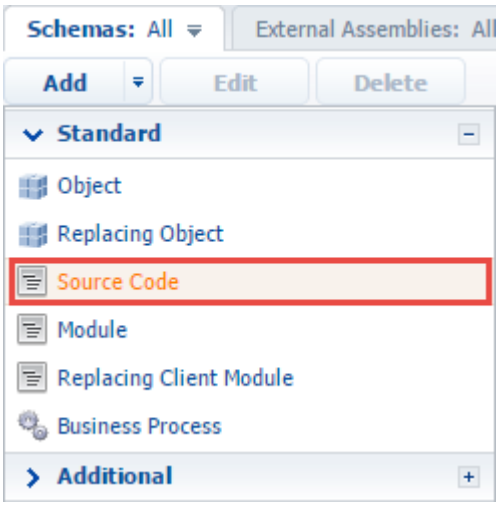
Creating the [Source code] schema

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.

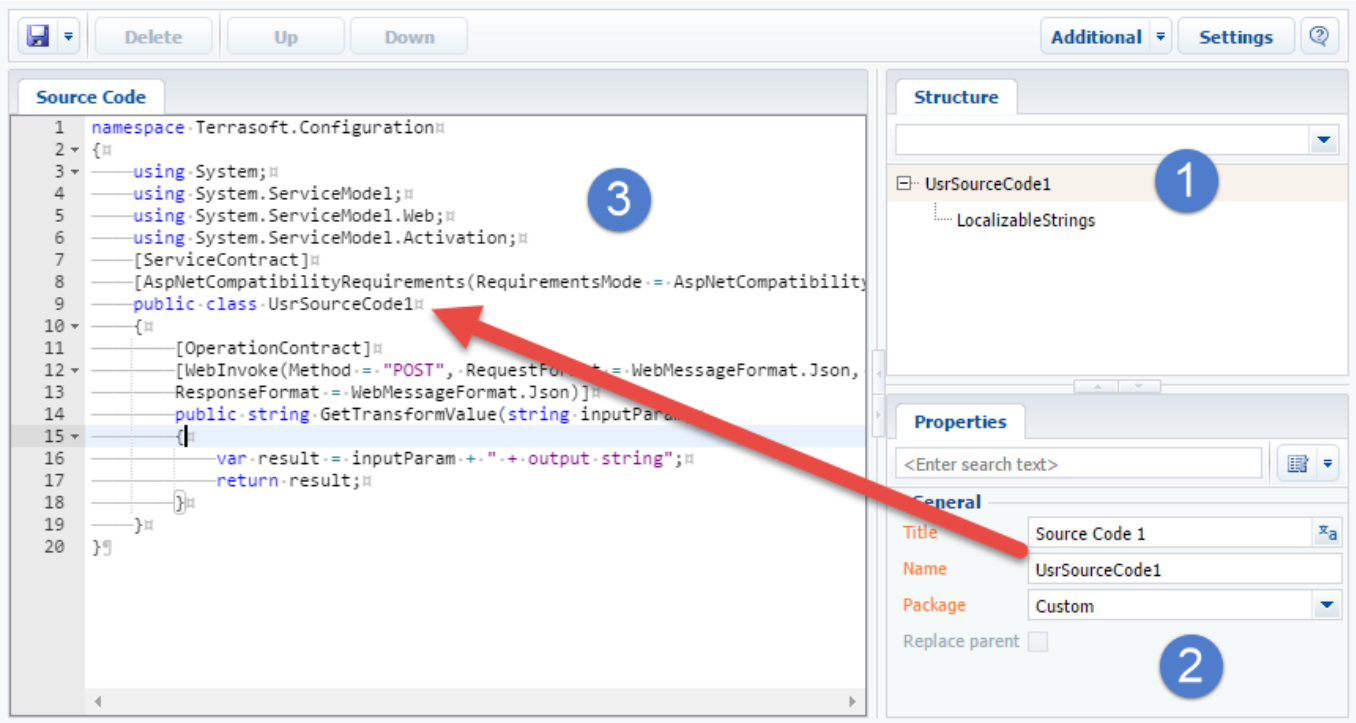
2. On the [Schemas] tab, run the Add > Source Code command (Figure 1).

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



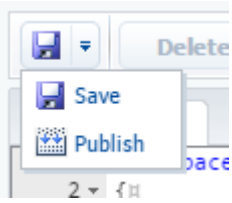
Main [Source code] schema properties:

- [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

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

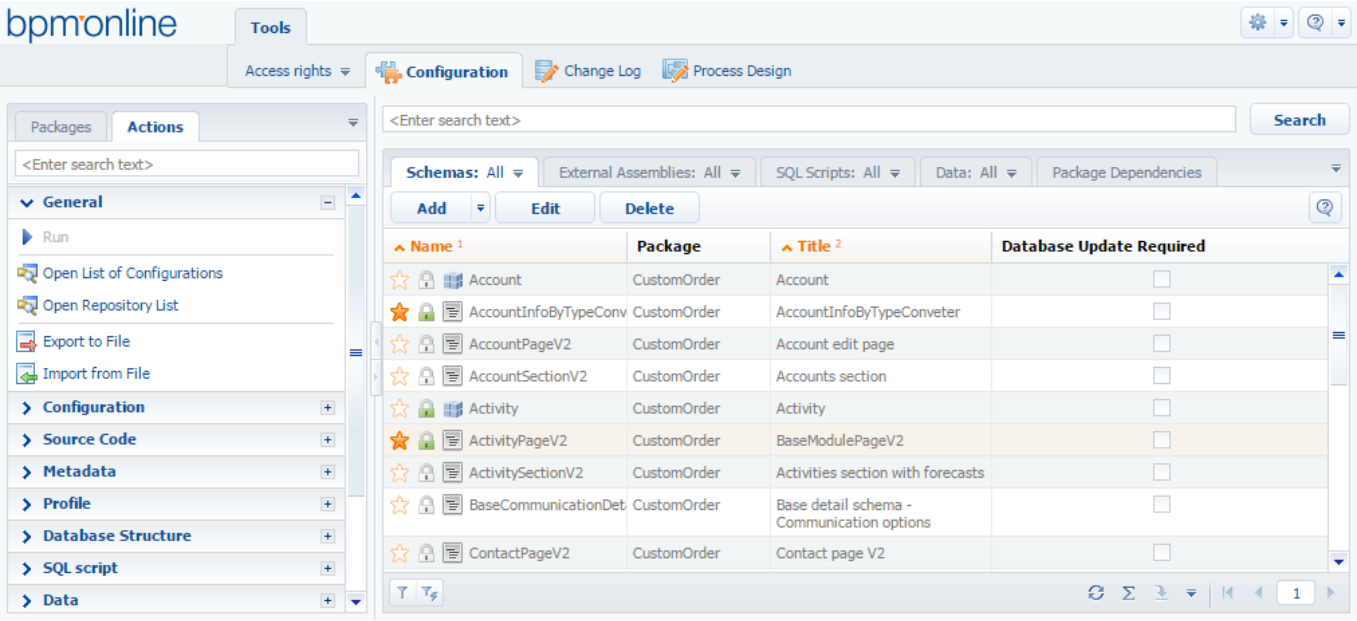
Built-in development tools

- **General information about the [Configuration] section**
- **Source code and metadata viewport**
- **Designers of configuration items**

General information about the [Configuration] section

Section [Configuration] (figure 1) is designed for control of configuration items, used in implementation of system functionality.

Figure 1. — Section [Configuration]



Section [Configuration] allows the following:

- creation and adjustment of workplaces;
- management of version control system repositories;
- control of packages, being a component parts of configuration;
- extension and modification of system functionality;
- control of parallel development of functionality and transfer of modifications between workspaces.

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

```
AccountInfoByTypeConveter
26
27        public string Evaluate(object value, string arguments = "") {
28            try {
29                _userConnection = (UserConnection)HttpContext.Current.Session["Us
30                Guid accountId = new Guid(value.ToString());
31                return getAccountInfo(accountId);
32            } catch (Exception err) {
33                if (arguments == "DEBUG") {
34                    return err.Message;
35                } else {
36                    return "mistake";
37                }
38            }
39        }
40
41        private string getAccountInfo(Guid accountId) {
42            try {
43                EntitySchemaQuery esq = new EntitySchemaQuery(_userConnection.Ent
44
45
```

Cancel

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]

Metadata | Metadata (Read-Only) | Modifications Package


```
{
 "MetaData": {
  "Schema": {
   "ManagerName": "ClientUnitSchemaManager",
   "UId": "fb72b2b4-3c1e-4a1a-8365-ef1126e49a06",
   "A2": "AccountPageV2",
   "A5": "6f5cc362-2b5b-4221-a1d3-e16886f0c981",
   "B1": [],
   "B2": [
    {
     "UId": "f7b9fe23-7644-4a73-9fc8-49cedceb36ad",
     "A2": "SaveButtonCaption",
     "A3": "d3cc497c-f286-4f13-99c1-751c468733c0",
     "A4": "d3cc497c-f286-4f13-99c1-751c468733c0",
     "A5": "e502d56a-e797-4315-8a0b-0fab9b8f01ab"
    },
    {
     "UId": "29a8e1d9-565a-4843-bc94-ad3311a87a83",
     "A2": "CancelButtonCaption",
     "A3": "d3cc497c-f286-4f13-99c1-751c468733c0",
     "A4": "d3cc497c-f286-4f13-99c1-751c468733c0",
```

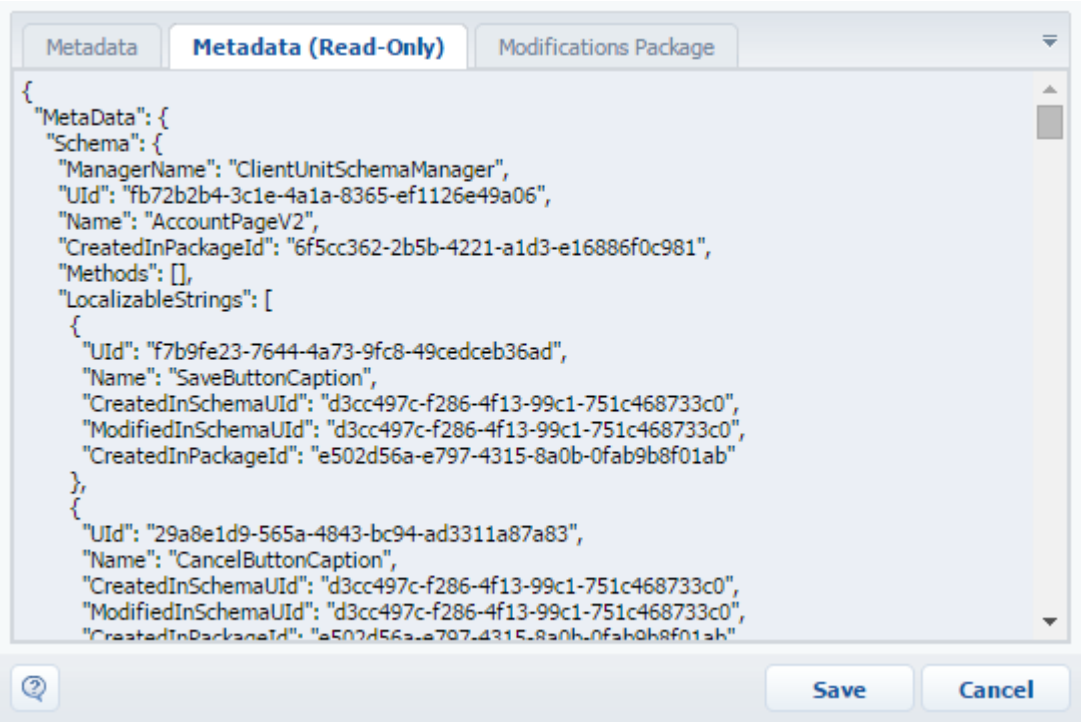

Save Cancel

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.

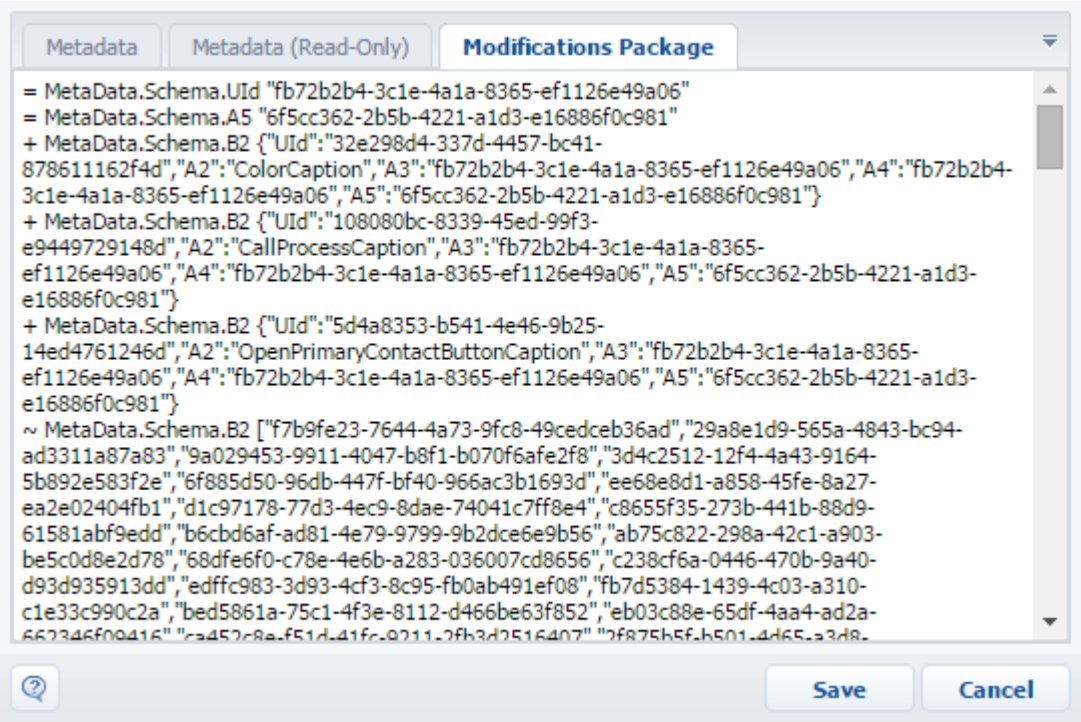
The *[Metadata (Read)]* (Figure 3) displays data that is similar to that displayed by the *[Metadata]* tab, but in a form suitable for reading. Internal identifiers (for example, "A2") are replaced with actual values of items, specified in the [Name] property field (for example, "AccountName"). This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



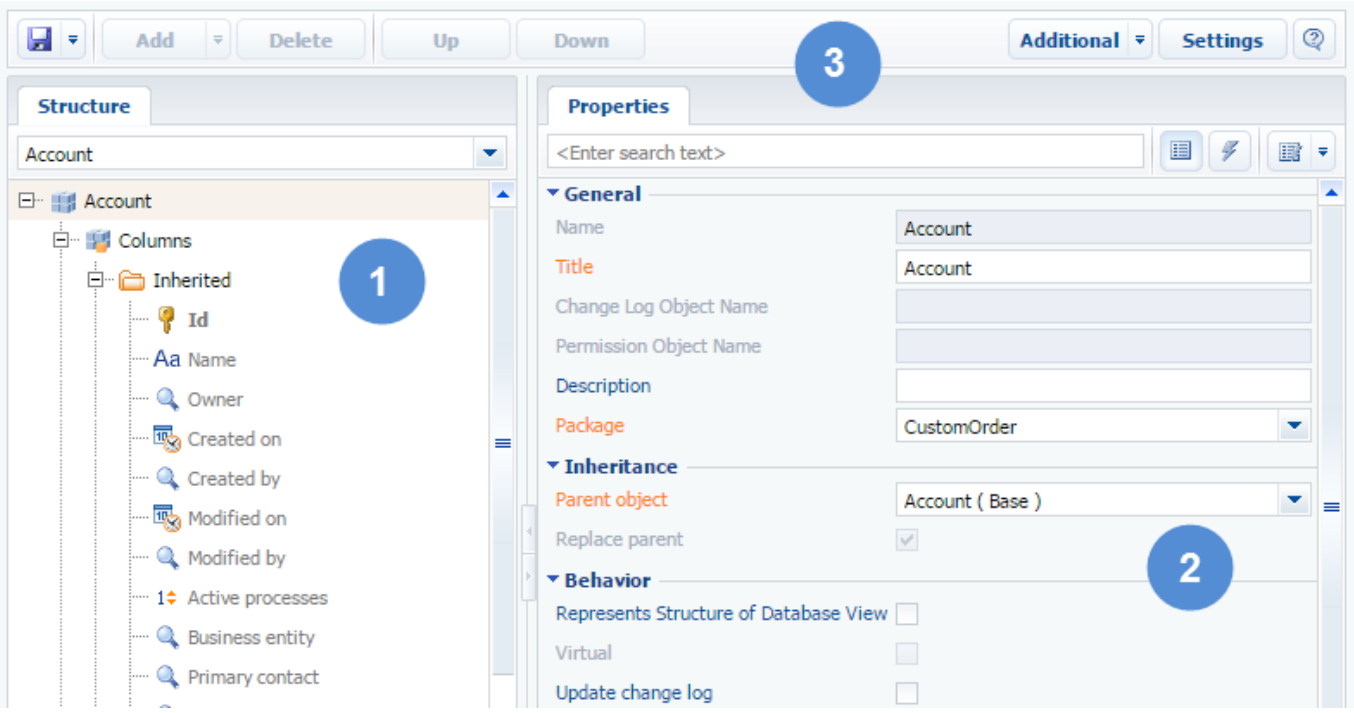
Designers of configuration items

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

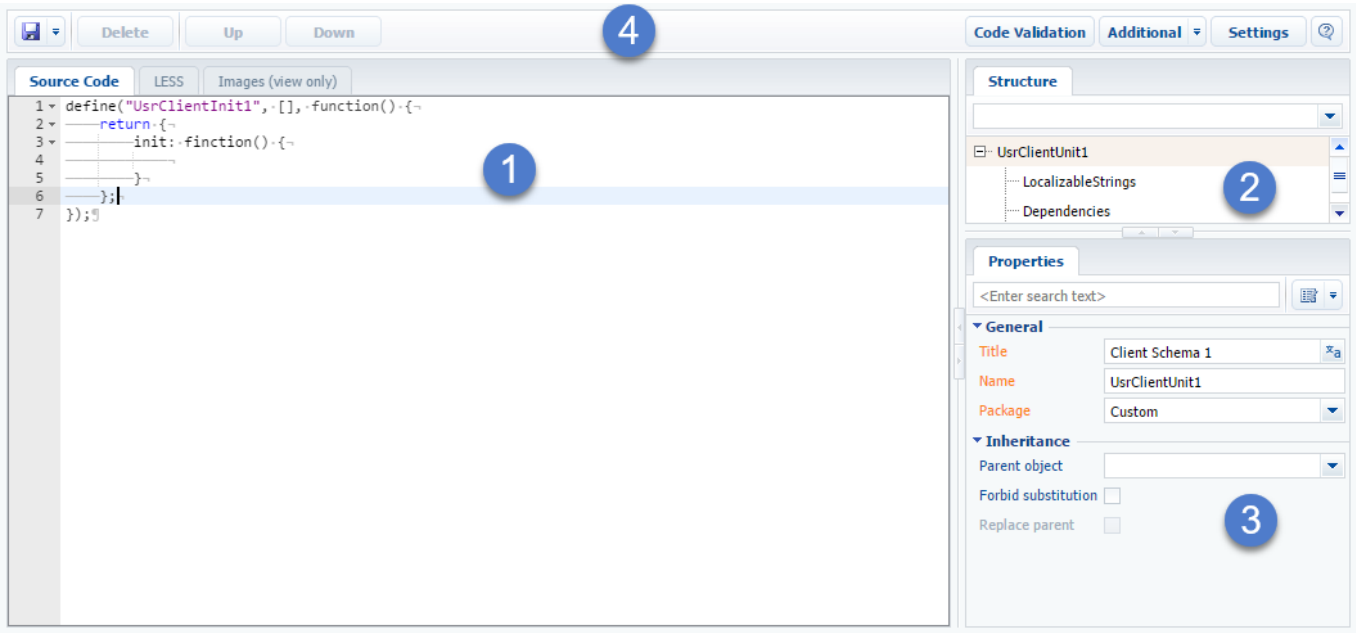
Module designer

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

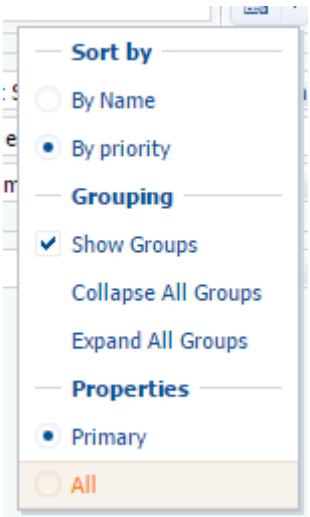
The schema structure area (2) is used to display the schema structure; the root element, resources, dependencies, messages and parameters.

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

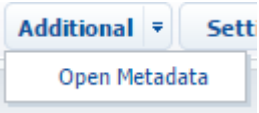


Toolbar

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

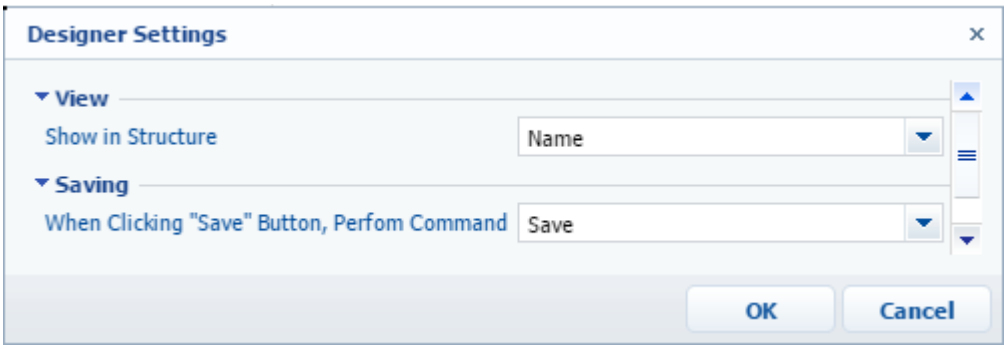


Settings window

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



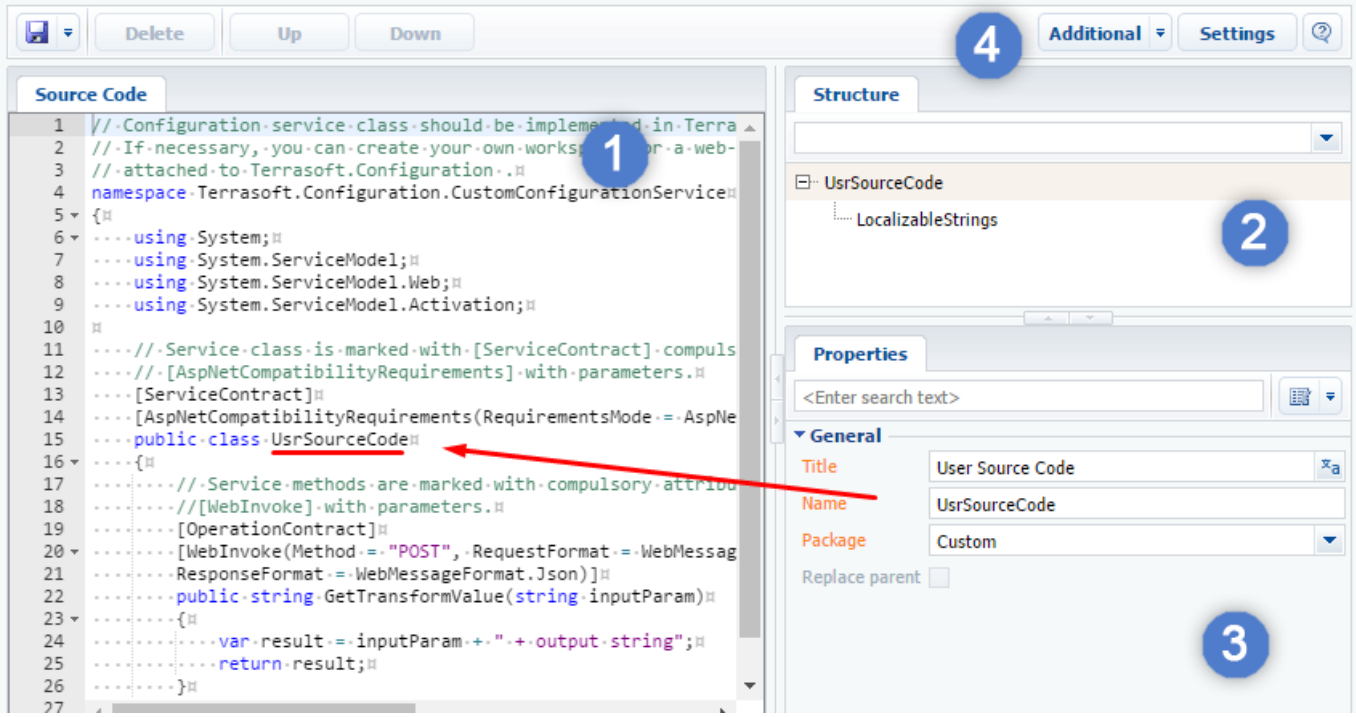
Source code designer

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

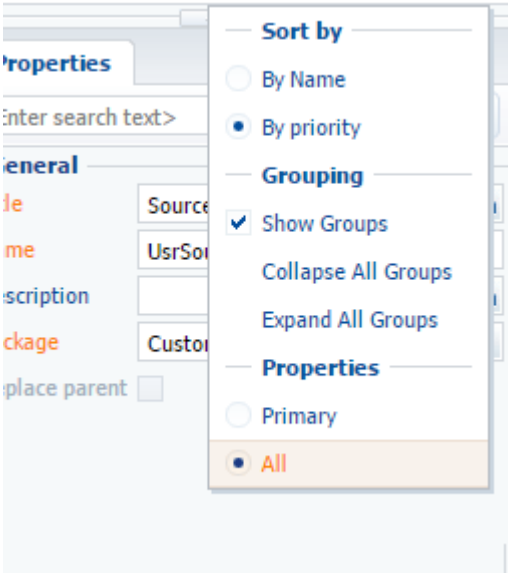
The schema structure (the root element and localizable strings of the source code) is located in the structure area (2).

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

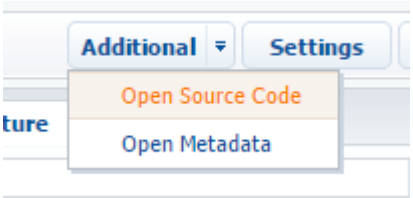


Toolbar

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



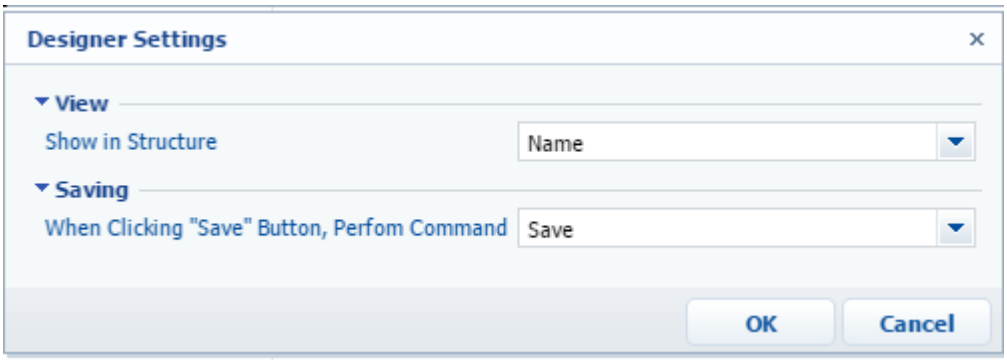
Settings window

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"	Choosing what happens when you click the [Save] button. Possible options – "Save"

Button, Perform Command	or "Publish" the schema.

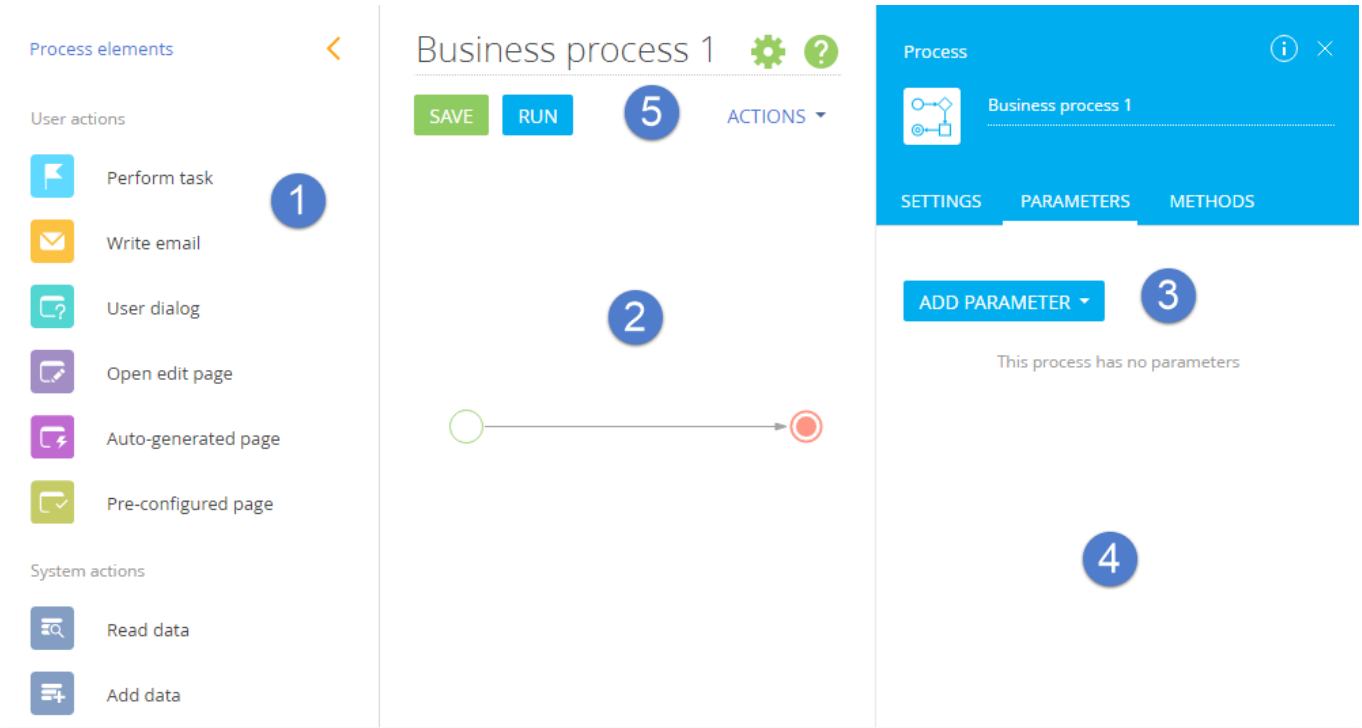
Fig. 4. Source code designer settings



Process designer workspace

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:

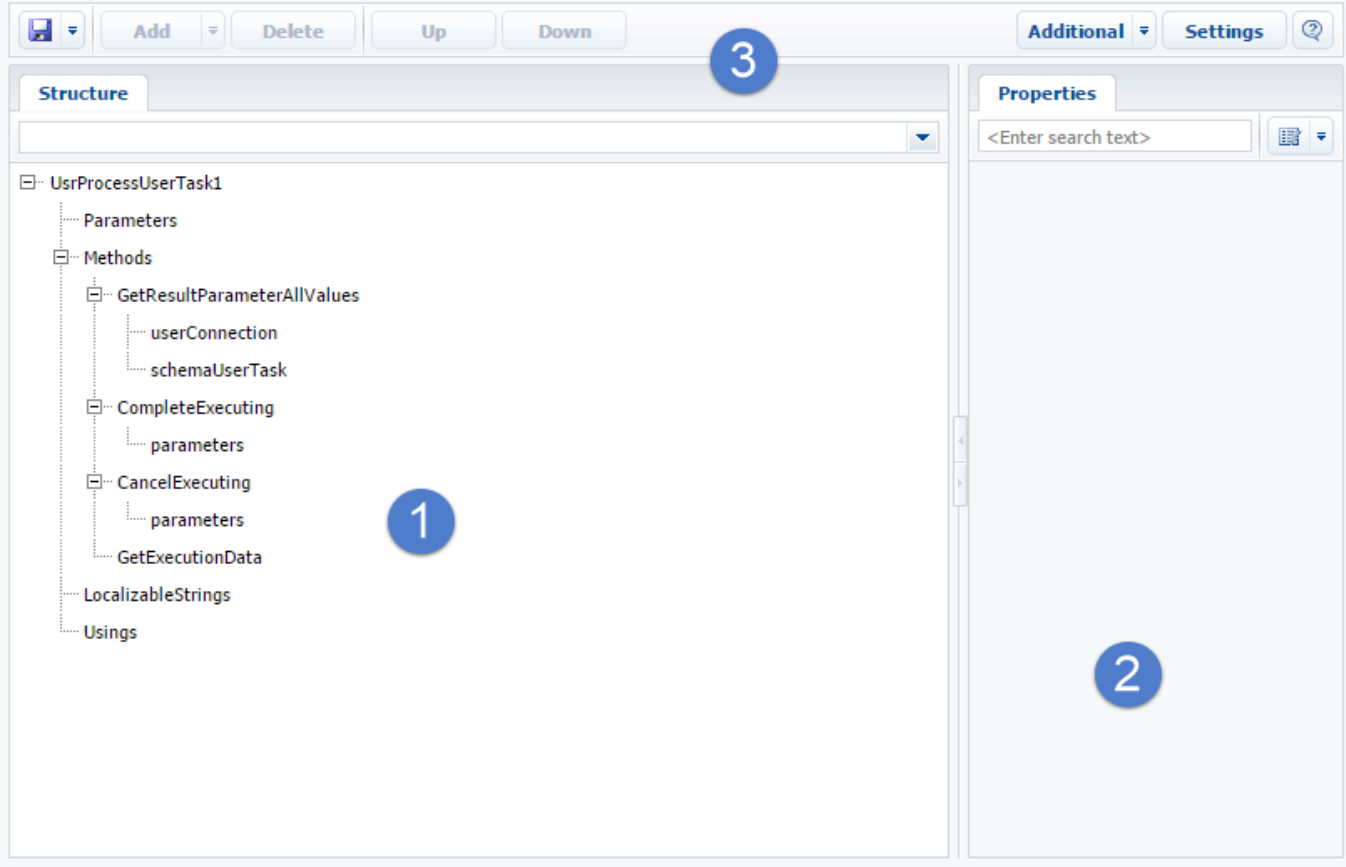
• [Open source code] — opens a window with the process source code.

• [Open metadata] — opens a window of the process metadata.

User task designer workspace

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.

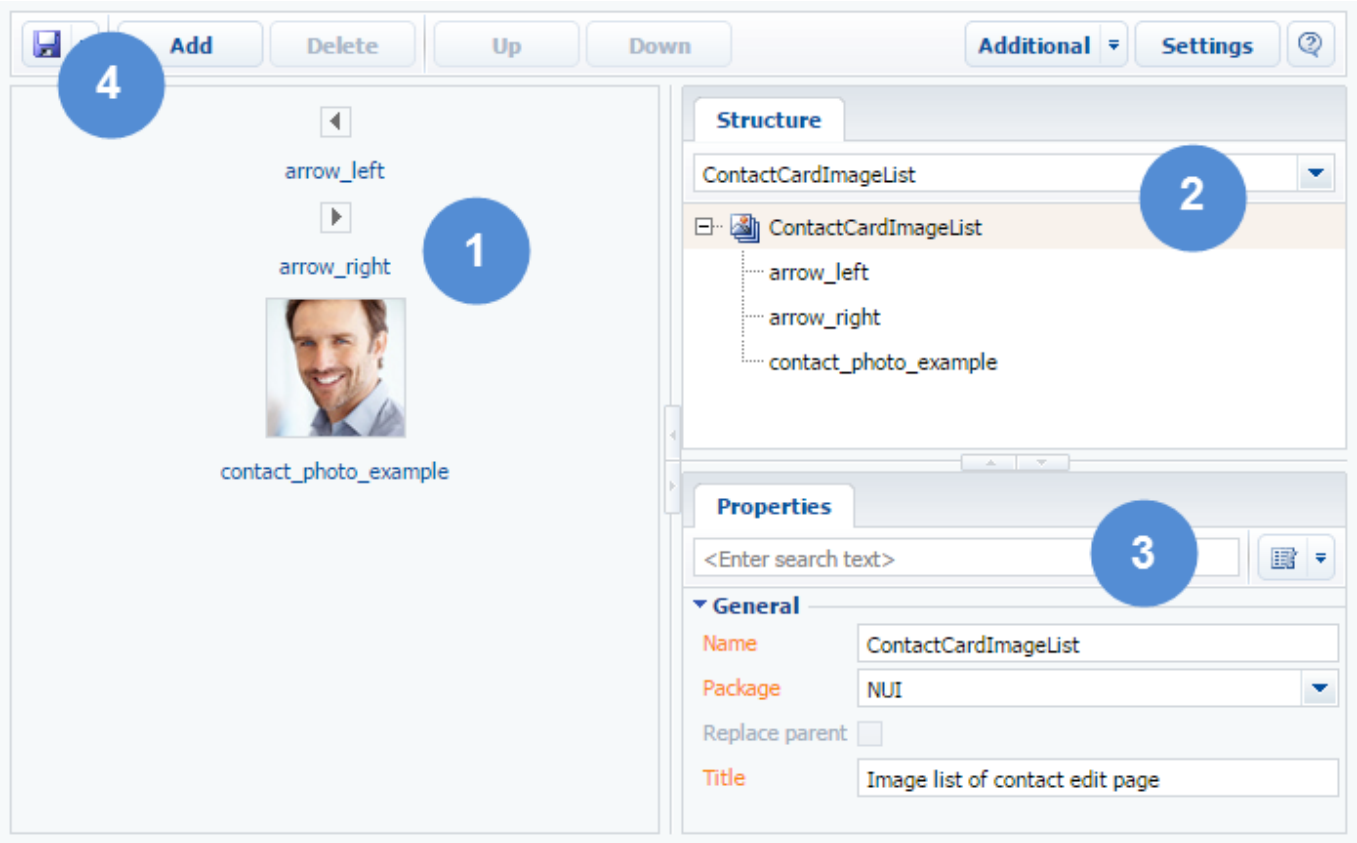
[Up] – moves an item up the list in the user task structure.

[Down] – moves an item down the list in the user task structure.

Workspace of image list designer

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.

DevExpress report designer

The [Report] configuration elements are used to form analytic reports to build printables.

Report designer is a separate application nased on the DevExpress Report Suite and 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 downoloaded by this link
> http://ftp.bpmonline.com/support/downloads/DevExpress/DevExpressReportDesigner_7_10_en.zip

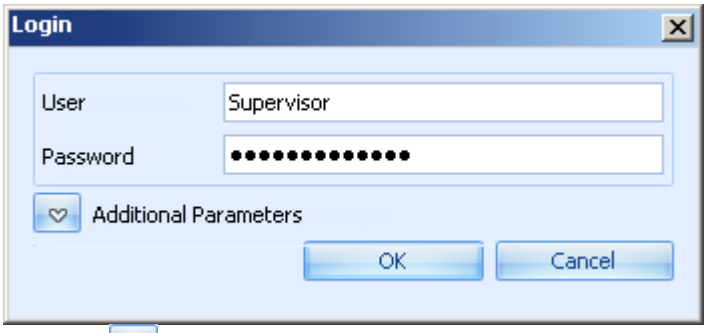
See also

- **Setting up the report designer connection with server**
- **Report designer workspace**
- **Report designer features**

Setting up the report designer connection with server

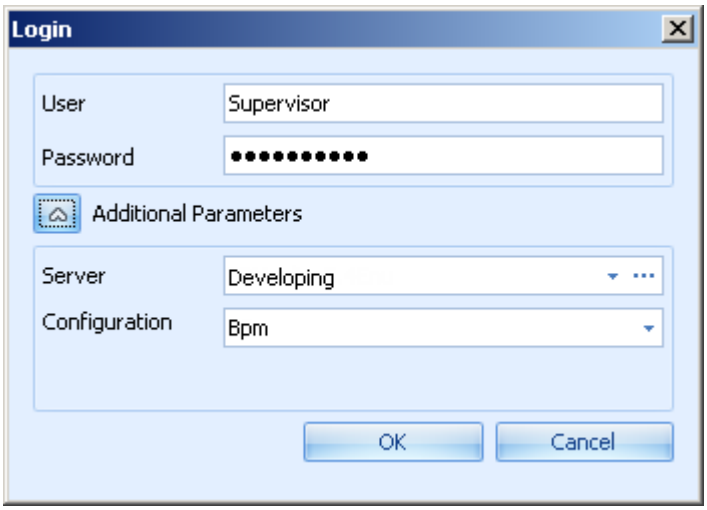
The report designer application starts with the login window (Fig. 1).

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

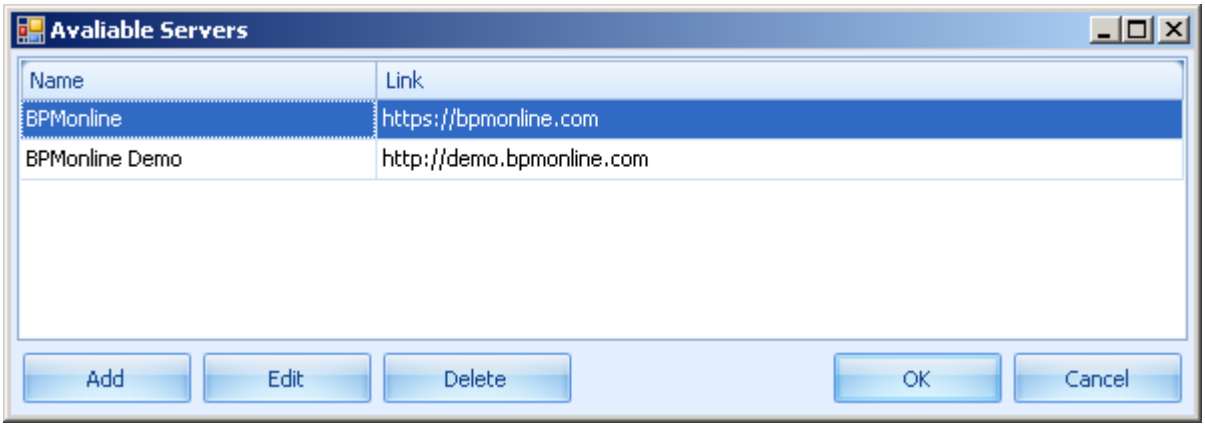
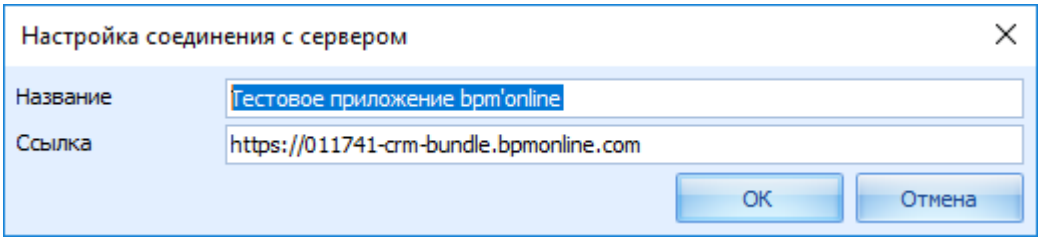


Table 2. Edit commands of the available servers list

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

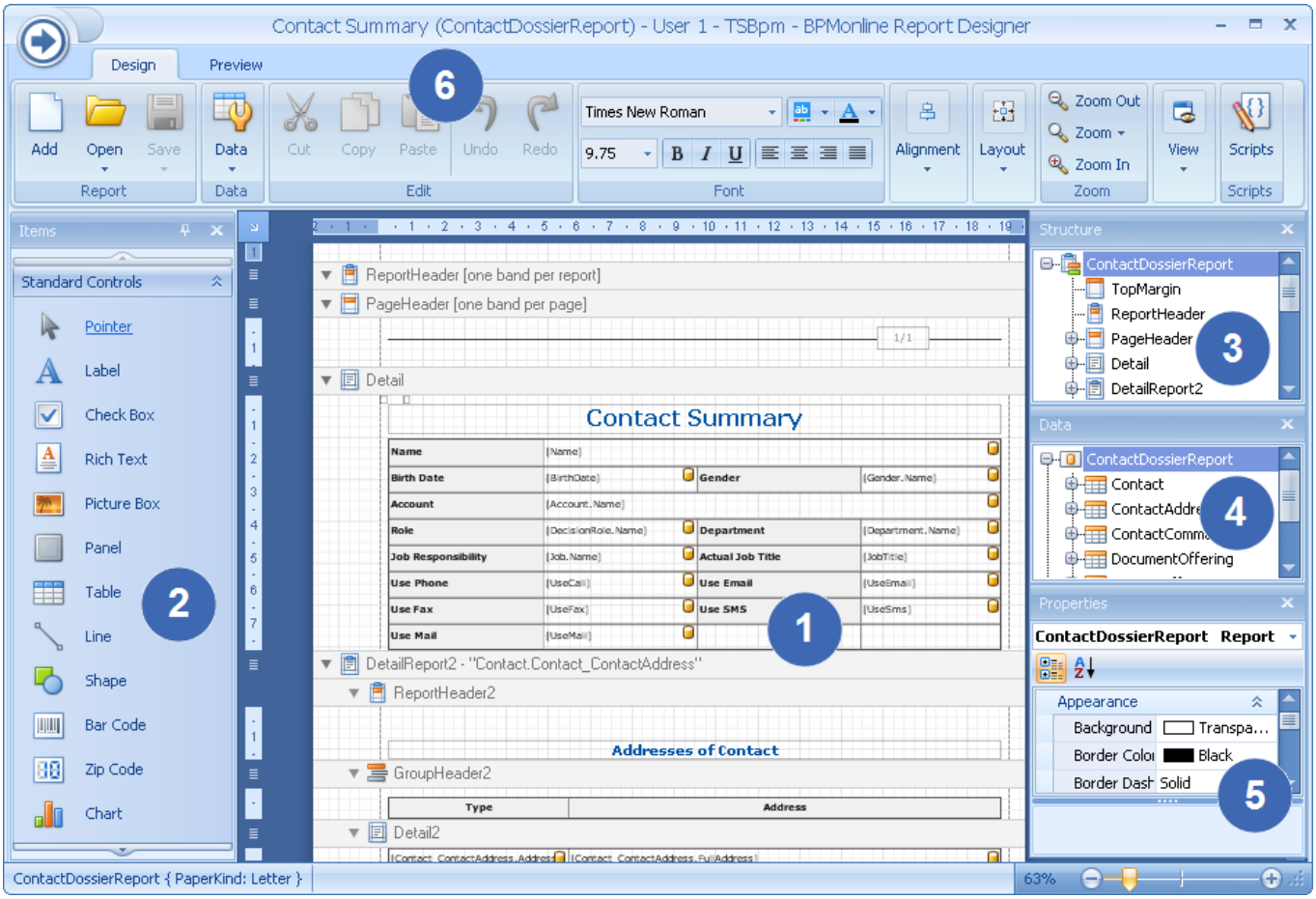


Report designer workspace

> ATTENTION
> A Zip archive with the report designer can be downloaded via this link
> http://ftp.bpmonline.com/support/downloads/DevExpress/DevExpressReportDesigner_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



Most of the report designer controls are standard for the DevExpress Report Suite.

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

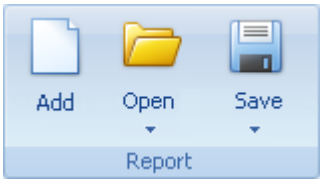


Table 1. Commands of the [Report] toolbar

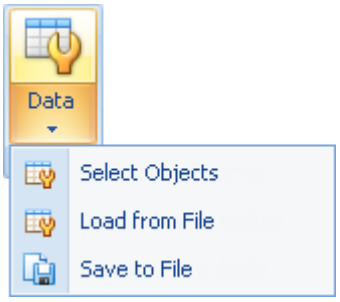
Command	Description
Create	Creates a new report If another report was opened when the [Create] button was clicked, you will receive a warning message to save the opened report before creating a new one. The report will be saved under its current name.
Open	Opens the selected report in the designer. Button 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.

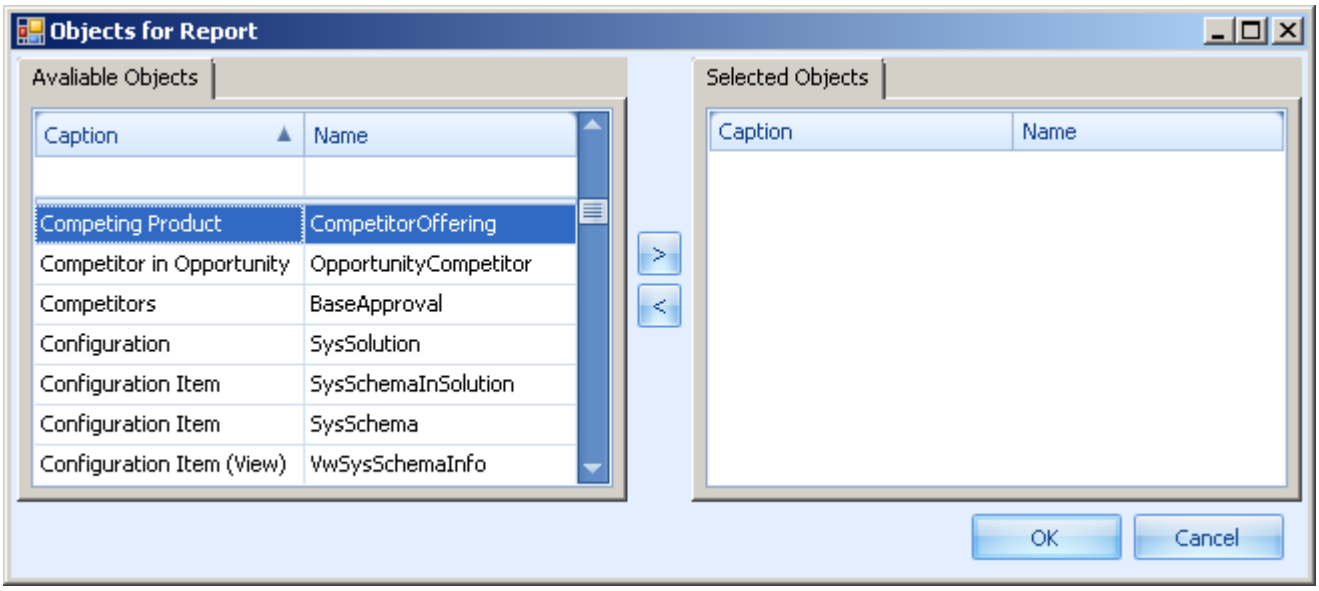
The [Objects for Report] window will open by clicking the [Data] button.

> 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

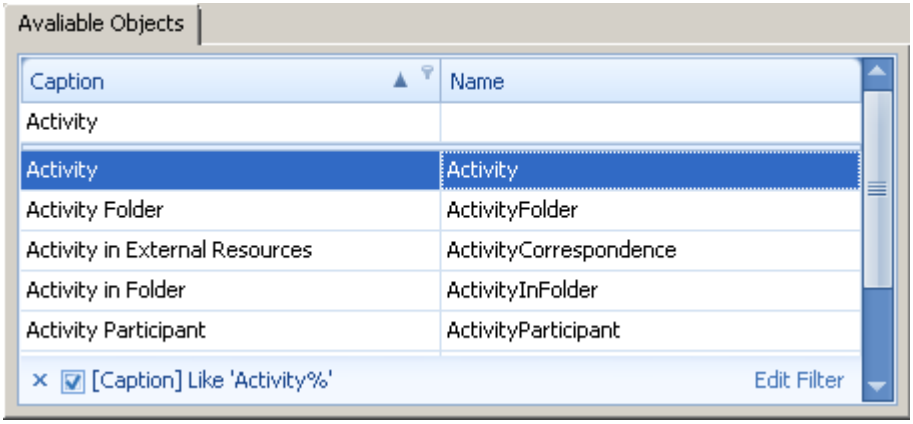


The [Available objects] area

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.

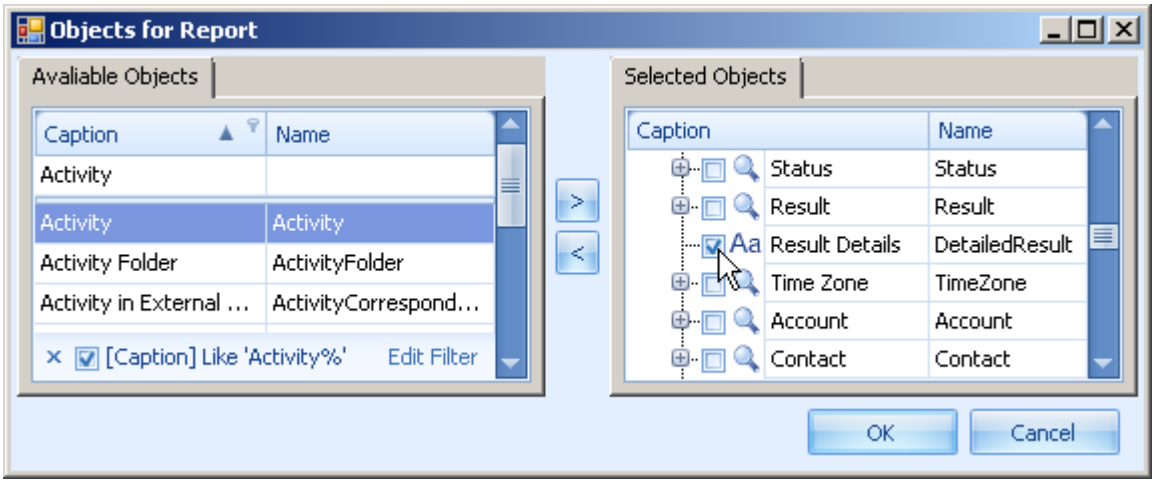
The [Selected objects] area

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.

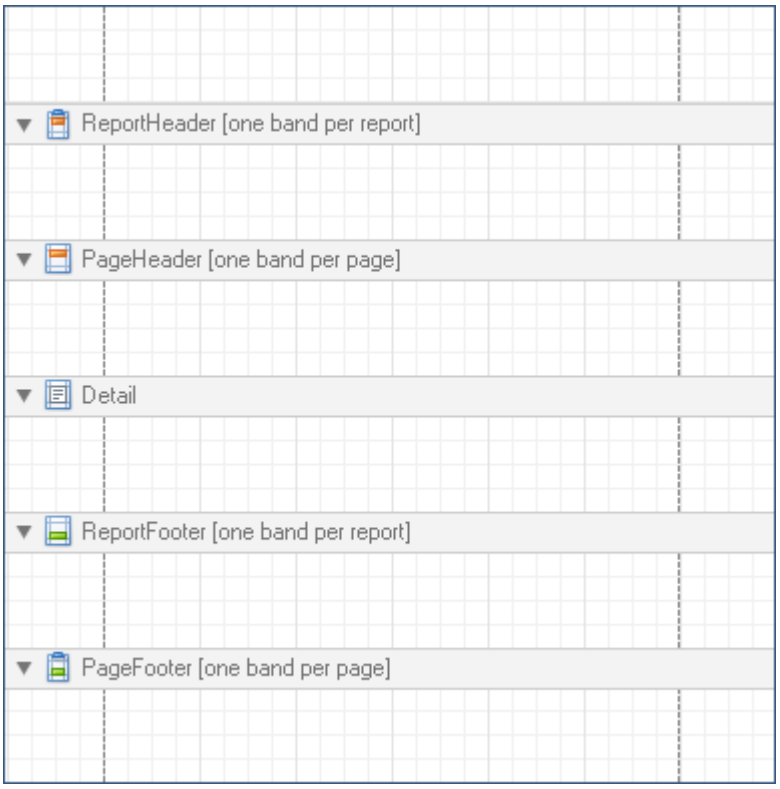
> ATTENTION
>
> When creating the report, the mechanism for filling out reports with data must be manually set by editing the data source of the report.

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.

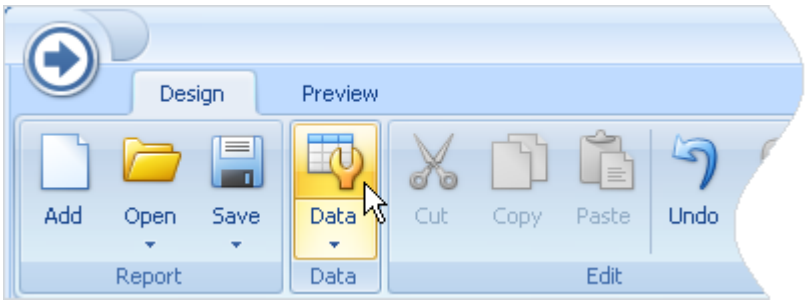
The [Detail] band is used to display a table in the report and is considered a table row template. One report can only have one [Detail] band.

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

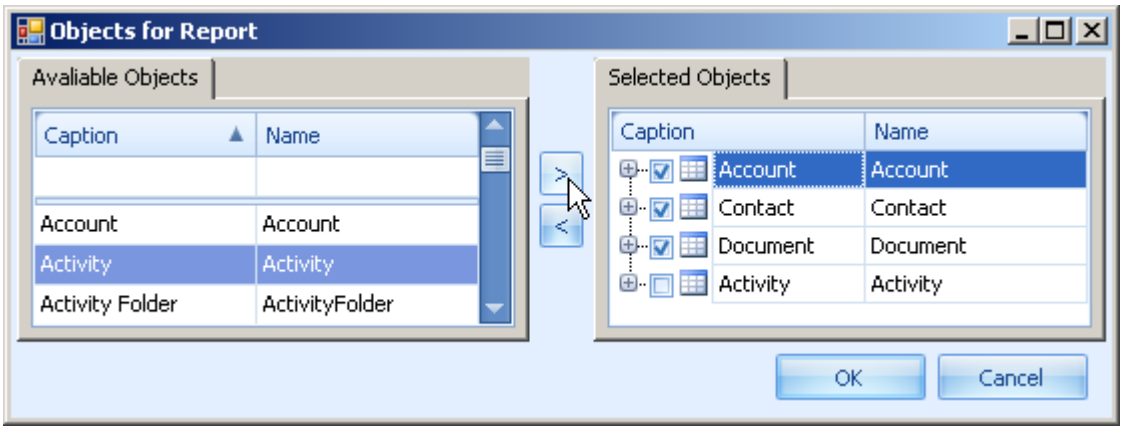
1. Click the [Data] button on the control panel (Fig. 2).

Fig. 2. Adding data to a report



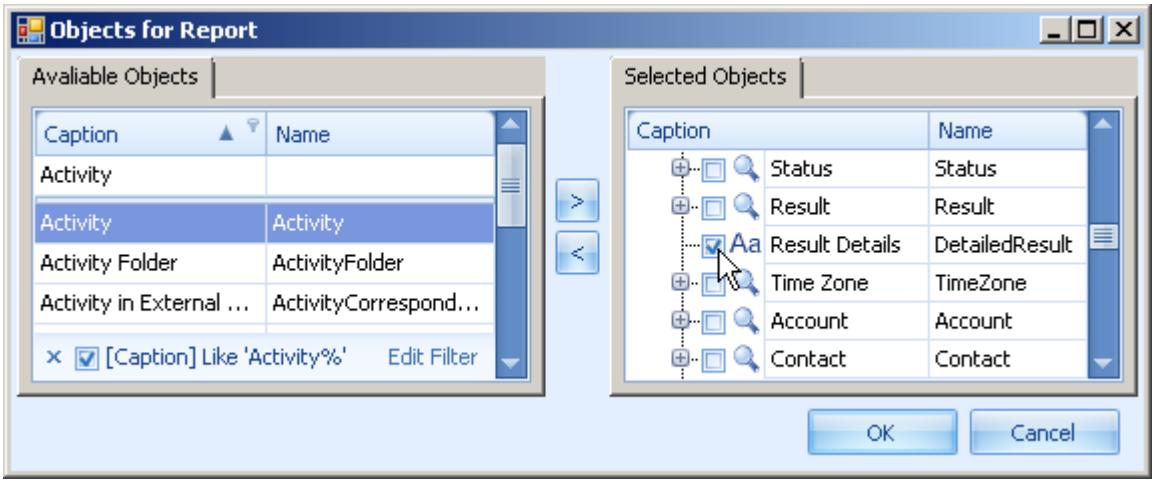
2. Add all objects the data of which will be used in the report to the [Selected objects] area, and enable checkboxes for them (Fig. 3).

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.

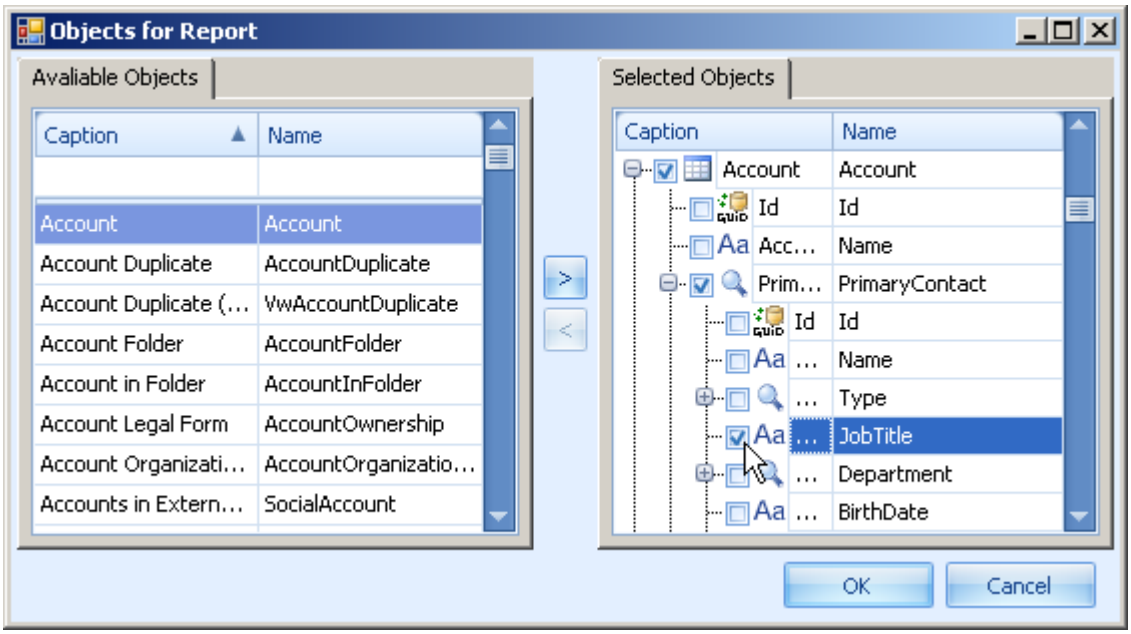
Added objects and their columns will be displayed in the [Data] window.

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

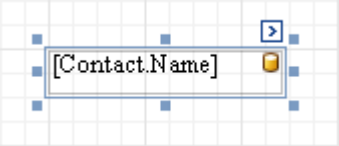
When you add connected data, enable the checkbox for the object the field of which you would like to use. Otherwise the added data will not be connected.

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).

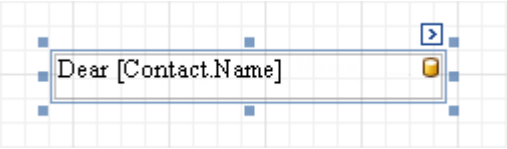
Fig. 6. Text object connected with the object column of the report data



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

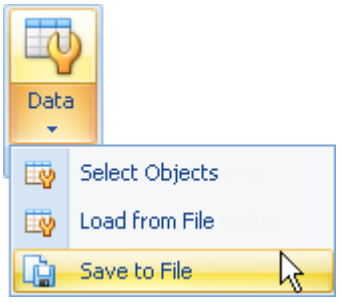
Edit the data source manually to fill the report with data from the database and for the advanced report configuration. For example, you can use columns that are not enabled in the system database in the report.

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.

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.

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

Attention!

Before version 7.10.0, bpm'online used the *UseFileContent* attribute of the *clientUnits* element in the Web.Config (located in the application catalog) for exporting source code files to the file system. Starting with version 7.10.0, the *UseFileContent* has been removed from Web.Config. Use the procedure covered in this article to export source code.

If the new procedure is used, the files with client schema source code from the pre-installed packages will also be exported to the file system (see description of the *Autogenerated\Src* folder below).

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" UseIDEForCompilation="false" />
```

NOTE

When developing source code schemas in C #, it is recommended to compile them directly in the Visual Studio. For this, set the *UseIDEForCompilation* attribute to *true*.

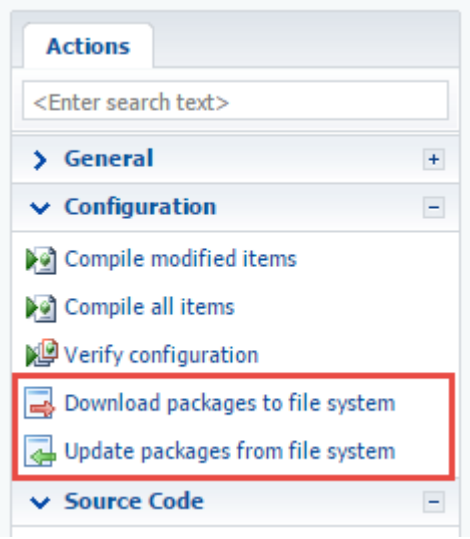
```
<fileDesignMode enabled="true" UseIDEForCompilation="true" />
```

More information about compilation and debugging in the Visual Studio can be found in the "**Working with the server side source code in Visual Studio**" article.

After enabling the development in the file system, two buttons will appear on the [Actions] tab in the [Configuration] section.

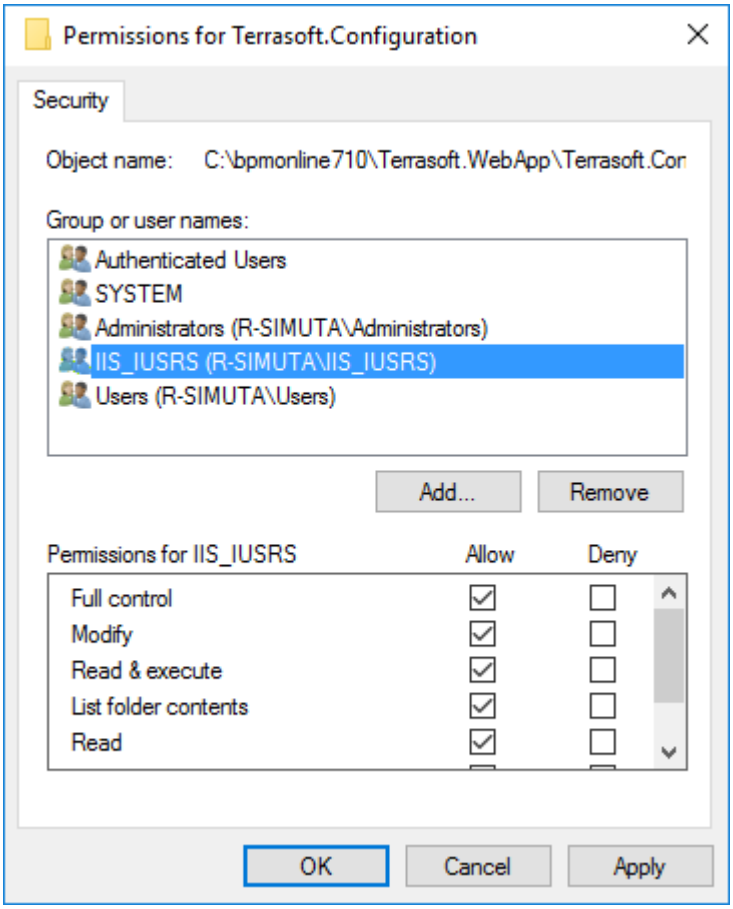
- [Download packages to file system] – exports the packages from the database to the following folder: *[path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg*.
- [Update packages from file system] – uploads the packages to the database from the following folder: *[path to the 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 2012 and up: *[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.

Visual Studio settings for development in the file system

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 2015 (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* and *UseIDEForCompilation* attributes of the *fileDesignMode* element to *true*.

```
<fileDesignMode enabled="true" UseIDEForCompilation="true" />
```

> Attention!
>
> Enable the **development mode in the file system** to compile in the Visual Studio.

The *UseIDEForCompilation* attribute specifies the location of the compiled *Terrasoft.Configuration.dll* class library. If this attribute set to *false*, the IIS will use the library from the database. If it is set to *true*, the IIS will use the library from the file system.

> NOTE
>
> Starting with version 7.10.1 this attribute has been deleted from the Web.Config. If the development mode in the file system is enabled, IIS will use the *Terrasoft.Configuration.dll* library only from the file system.

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" UseIDEForCompilation="true" />
```

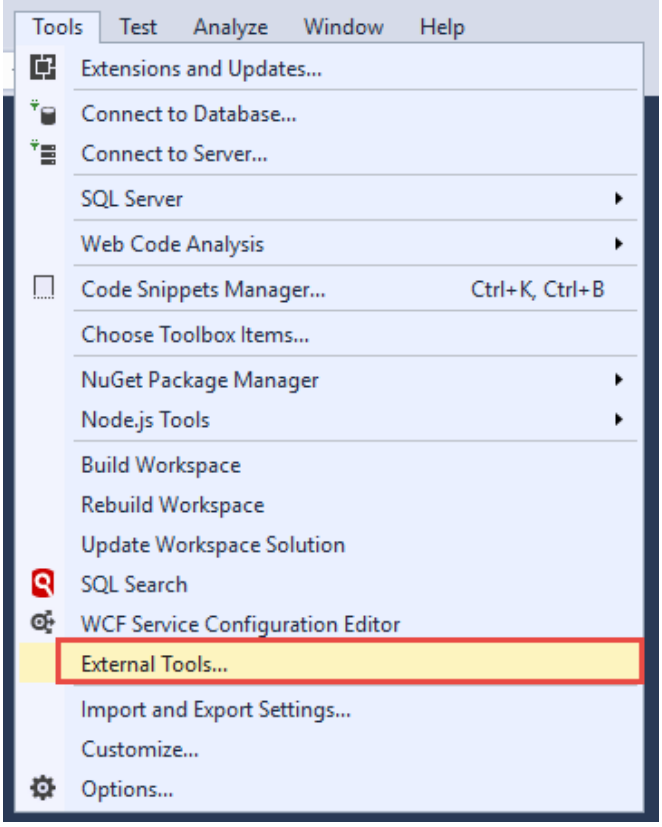
To speed up the compilation by splitting the configuration assembly into independent compiled modules, set the *CompileByManagerDependencies* setting to "*true*" in the "internal" Web.Config (located in the Terrasoft.WebApp directory) and in the Terrasoft.Tools.WorkspaceConsole.exe.config file of the 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

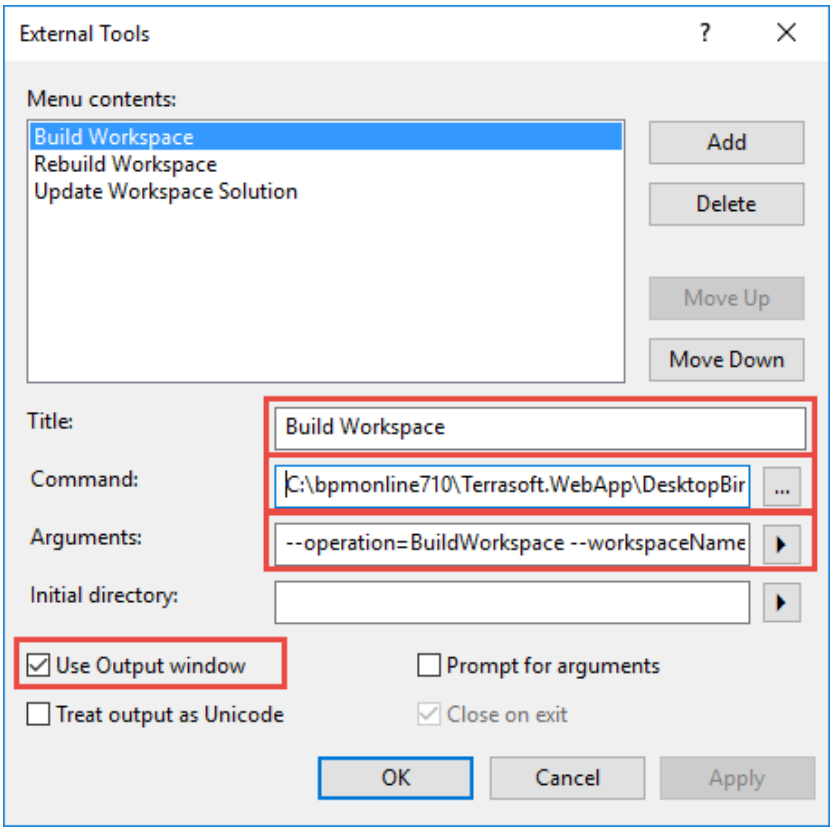


Table 1. Update Workspace Solution command properties

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

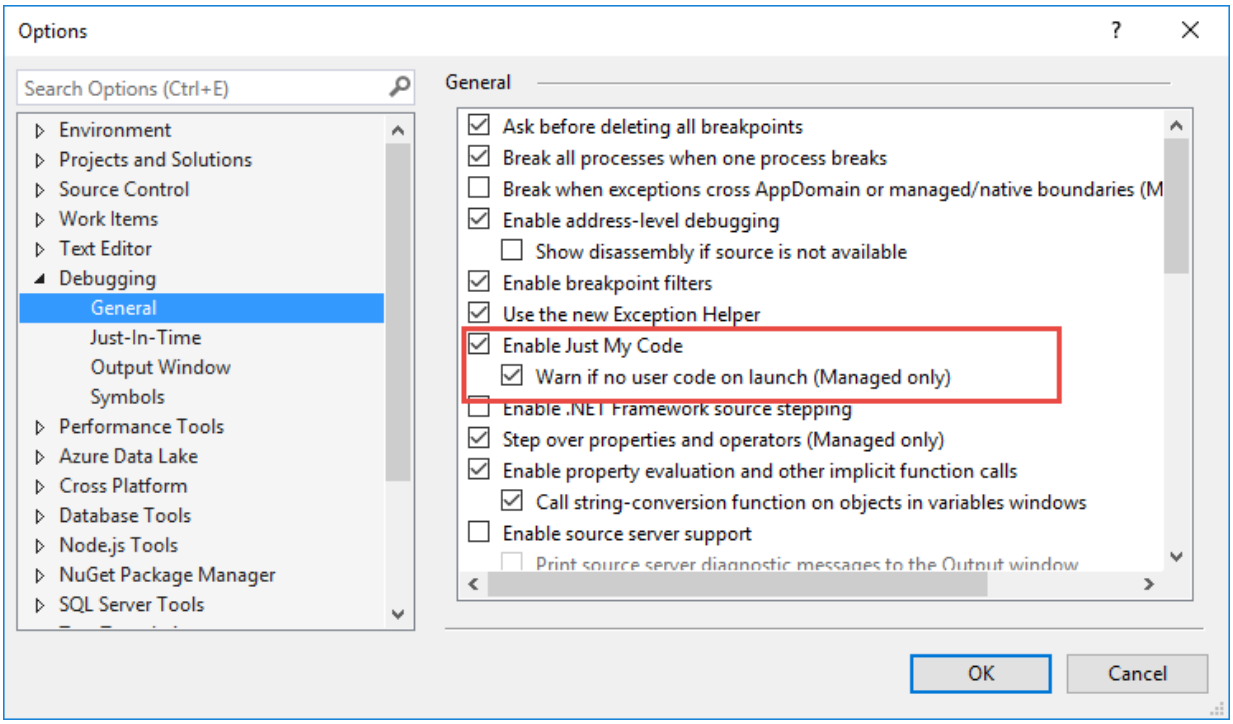
Title	Build Workspace
Command	[Path to the catalog with installed application]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe Example: C:\bpmonline710\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe
Arguments	--operation=BuildWorkspace --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\"

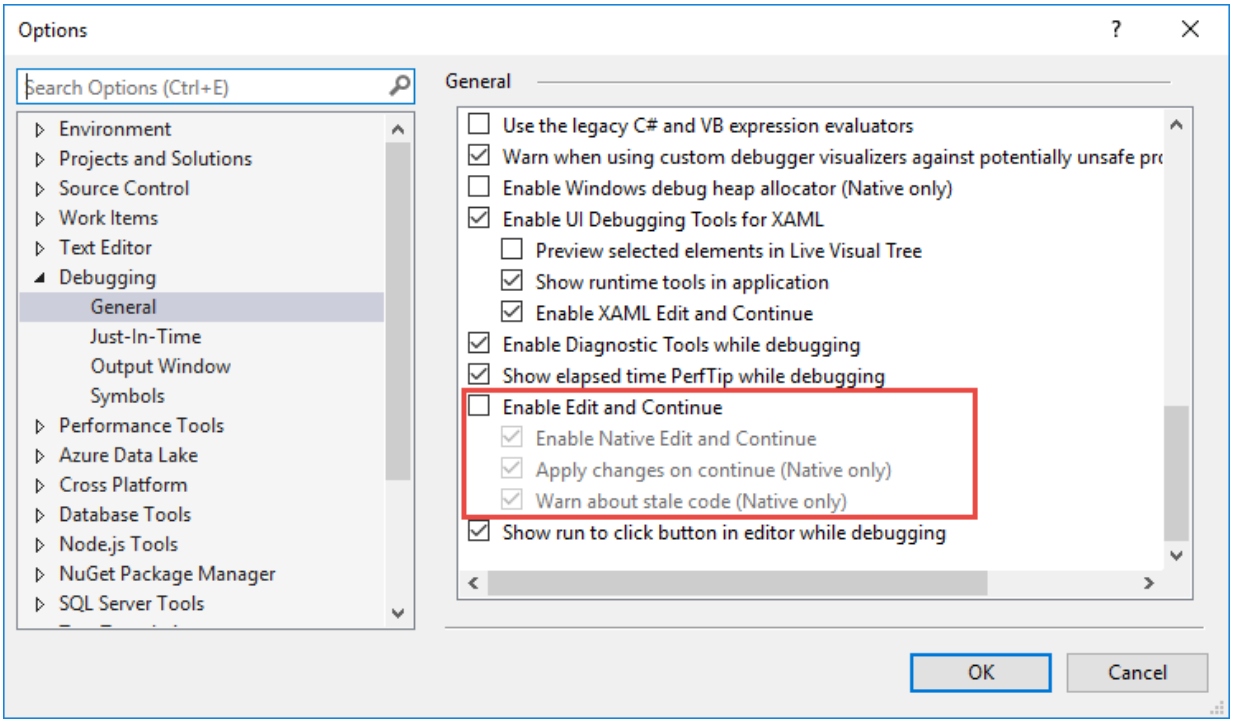
To prevent the debugger from accessing the source code that is disabled in the project, execute the Debug > Options... menu command and enable the *Enable Just My Code* option in the opened dialog (Fig. 3). For more information about the *Enable Just My Code*, please refer to this page.

Fig. 3 Enable Just My Code option



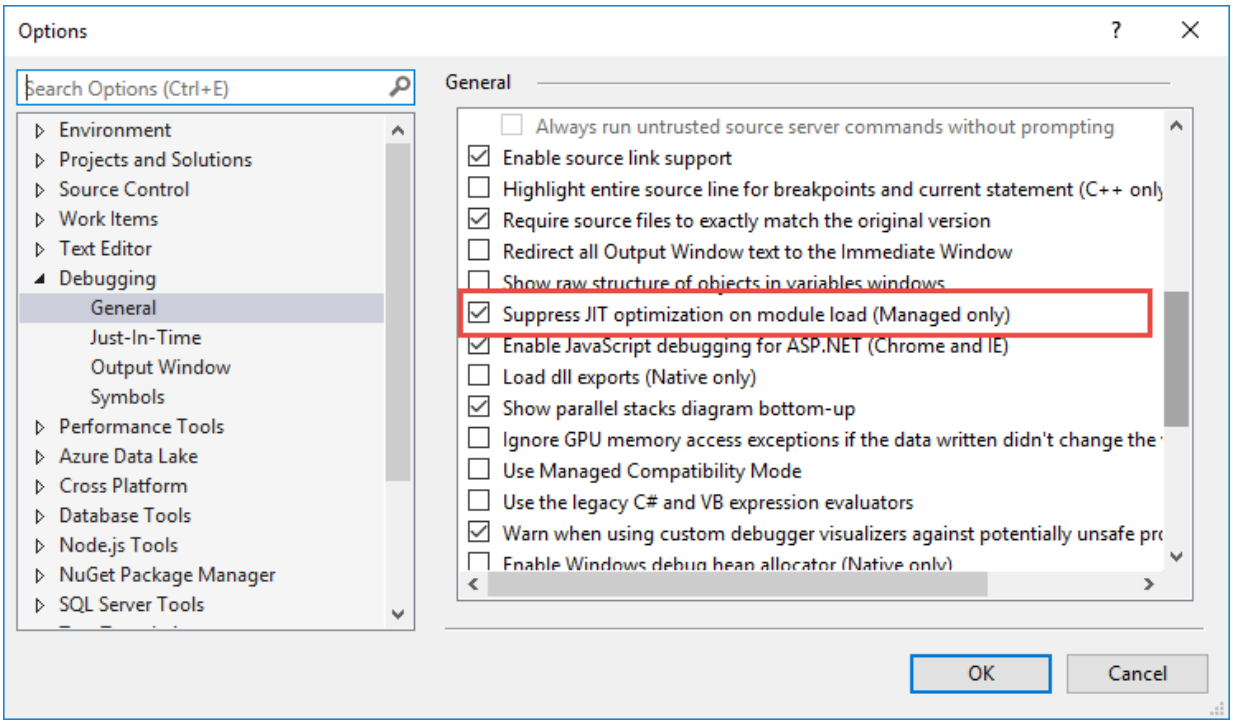
After the configuration is compiled, the application is automatically restarted. The *Enable Edit and Continue* option is not supported and should be disabled (Fig. 4).

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

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.

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.

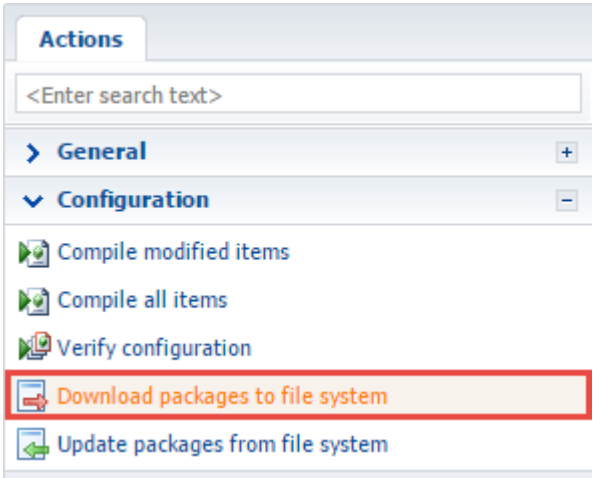
3. Create a [Source code] schema

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

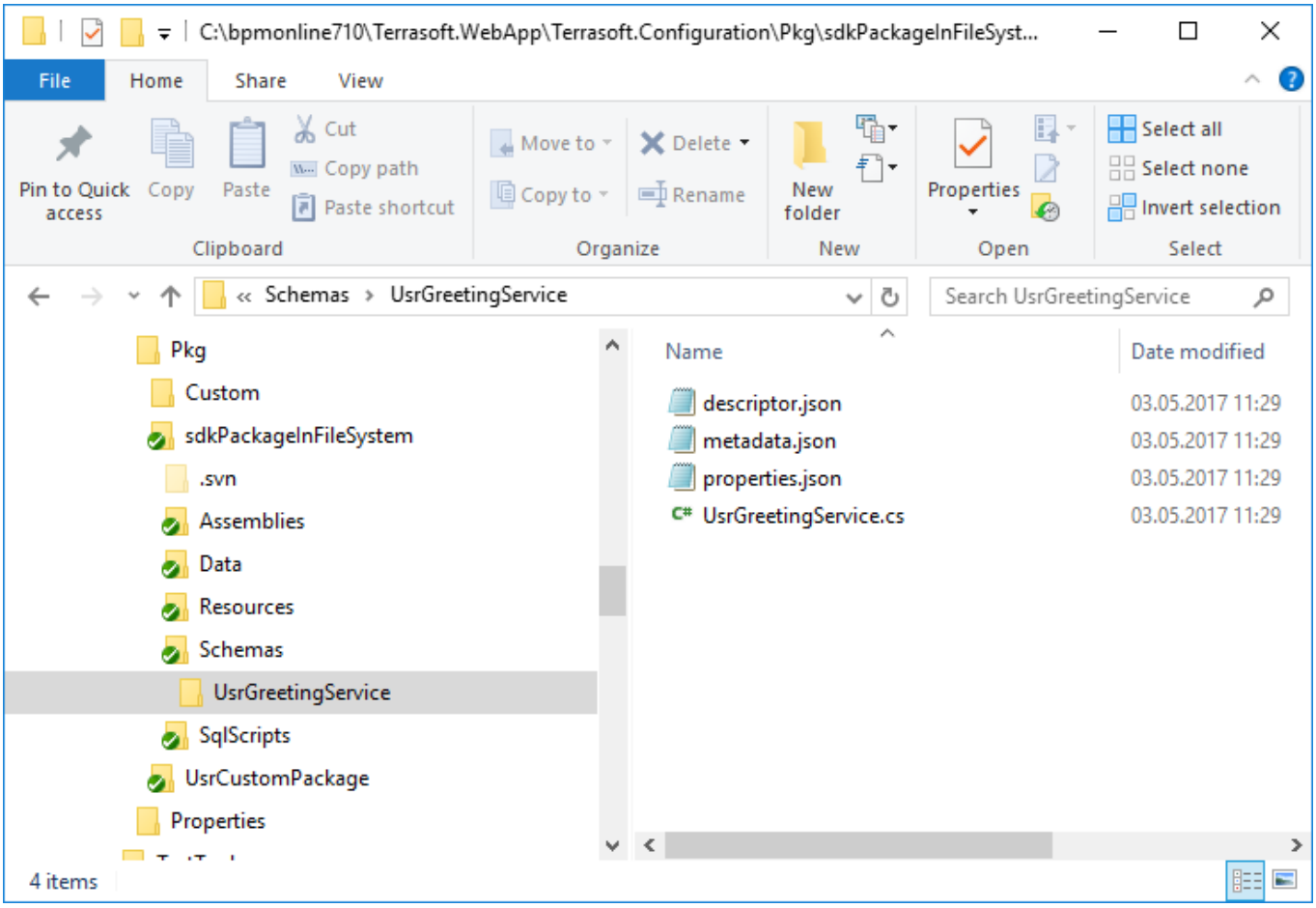
Before you start development in Visual Studio, make sure that you export existing schemas from the database to the file system. Before you start development in Visual Studio, make sure that you export existing schemas from the database to the file system.

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

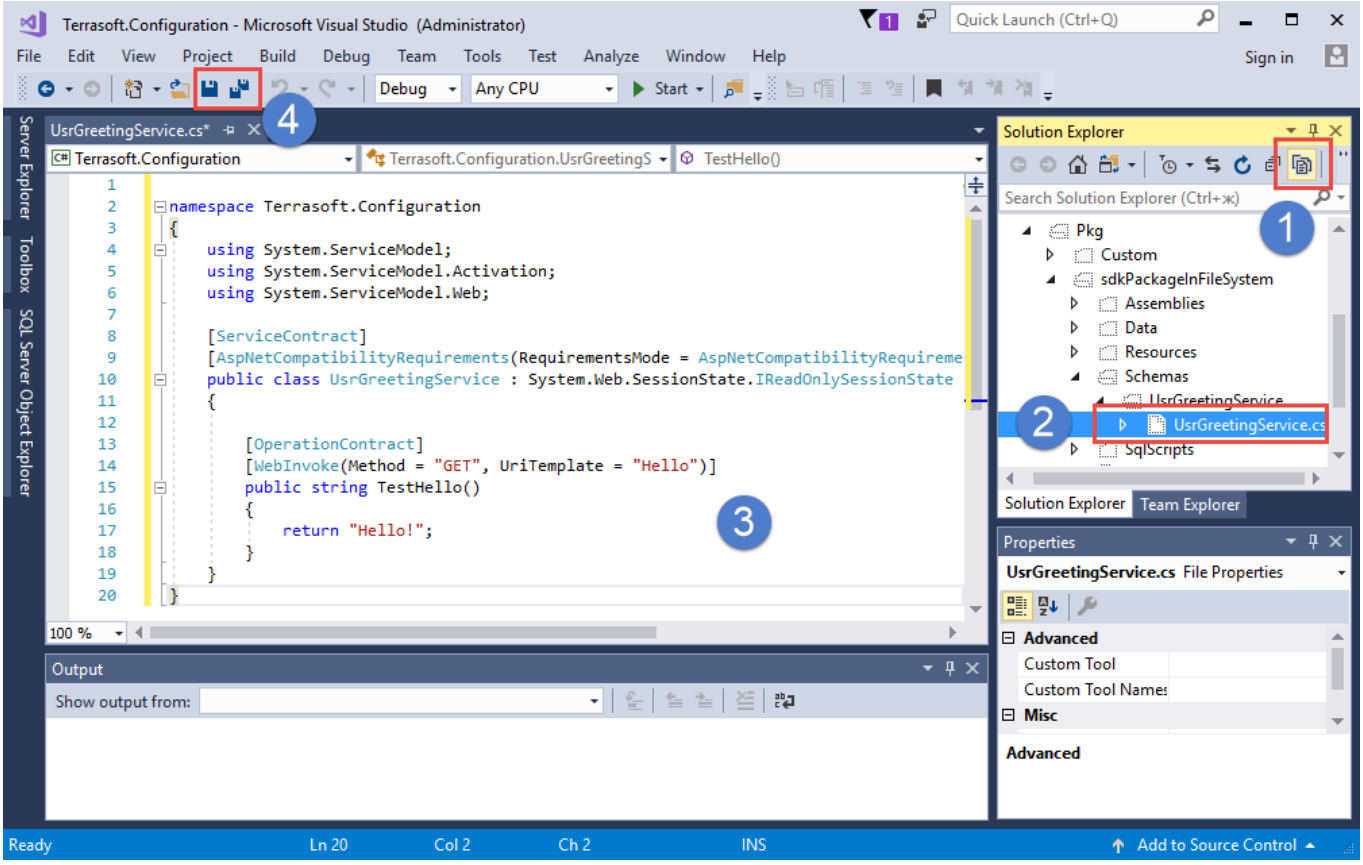


NOTE

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 own configuration service**".

5. Save, compile and debug source code

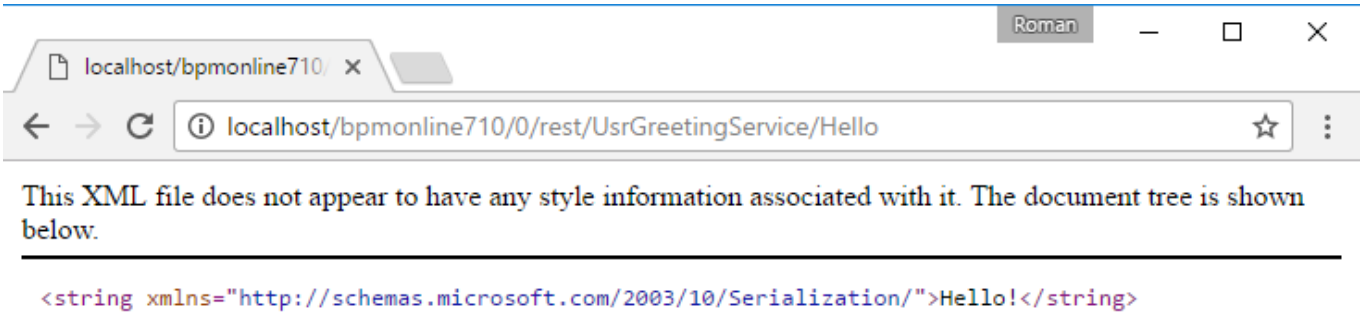
After modifying the source code, and before compiling and debugging it, be sure to save the code. Normally, this is done by Visual Studio automatically, but since Visual Studio compiler is not used, the developer must save the code manually.

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):

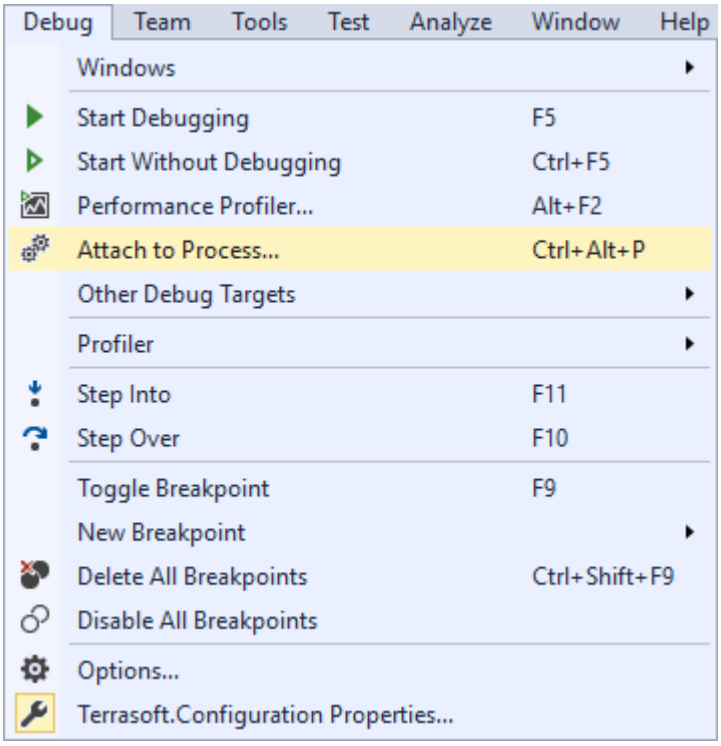
```
http://[Application URL]/0/rest/UsrGreetingService/Hello
```

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

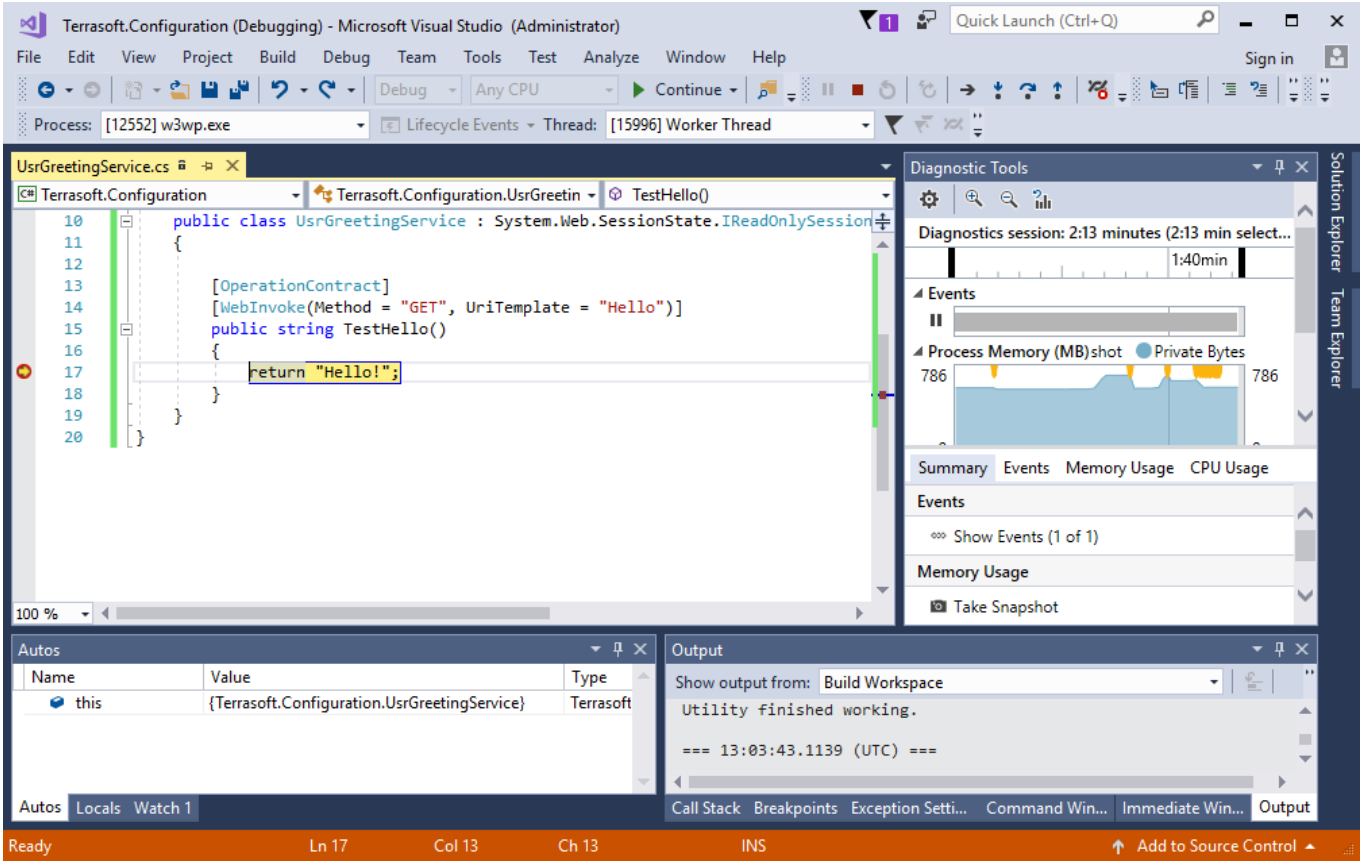
The name of the working process can be different, depending on the configuration of the IIS server being used. With a regular IIS, the process is *w3wp.exe*, but with IIS Express, the process name is *iisexpress.exe*.

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

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.

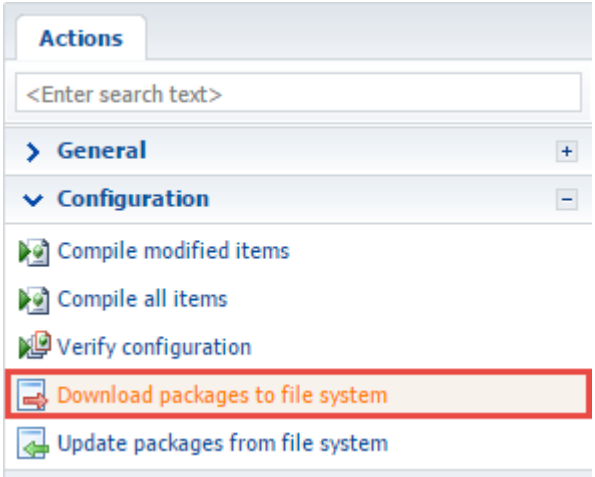
3. Create a custom schema for development

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

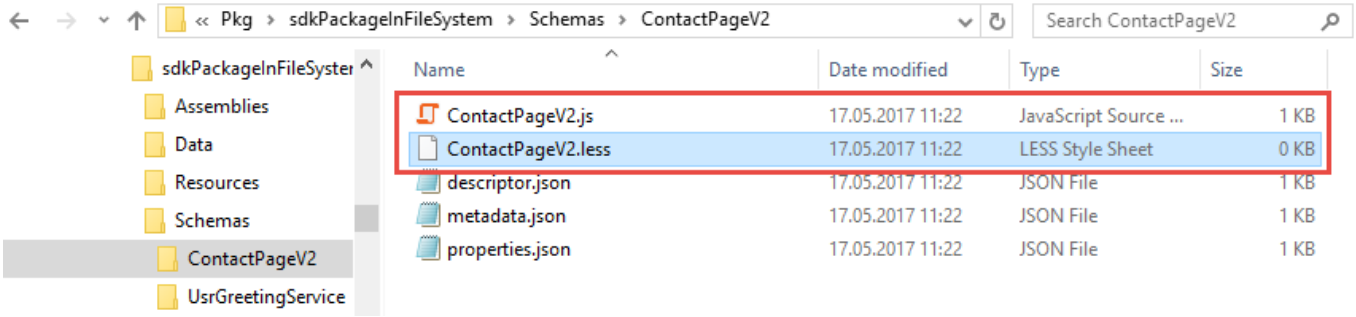
To do this, use the [Download packages to file system] command (Fig. 1) in the [Configuration] section.

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).

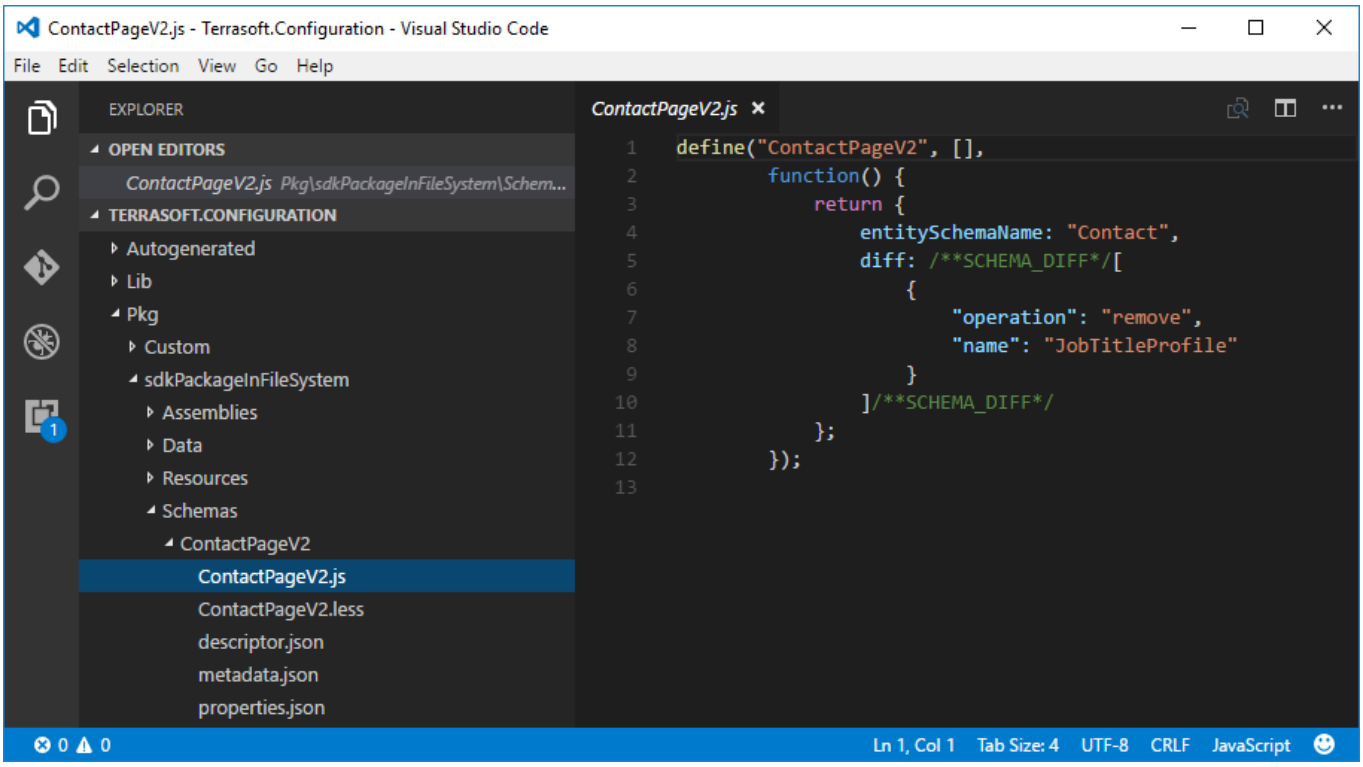
Fig. 2. The source code schema file



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:

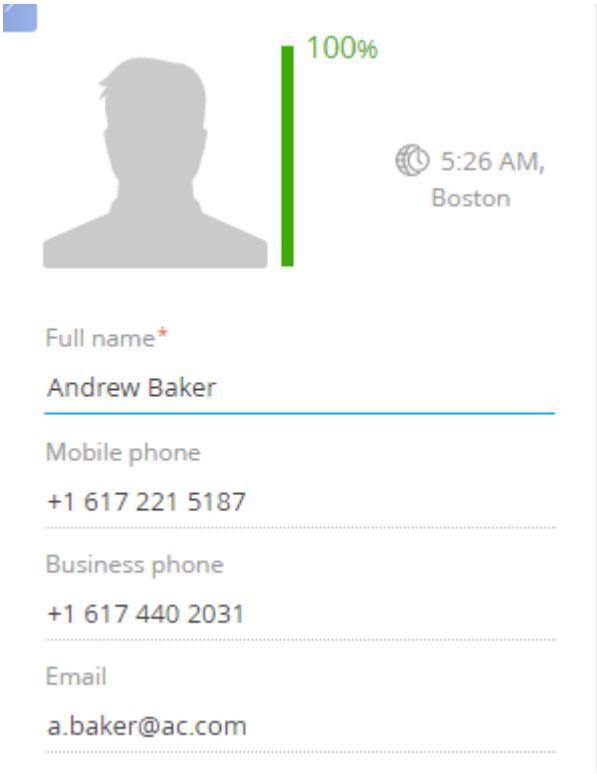
```
define("ContactPageV2", [],
        function() {
            return {
                entitySchemaName: "Contact",
                diff: /**SCHEMA_DIFF*/[
                    {
                        "operation": "remove",
                        "name": "JobTitleProfile"
                    }
                ]/**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

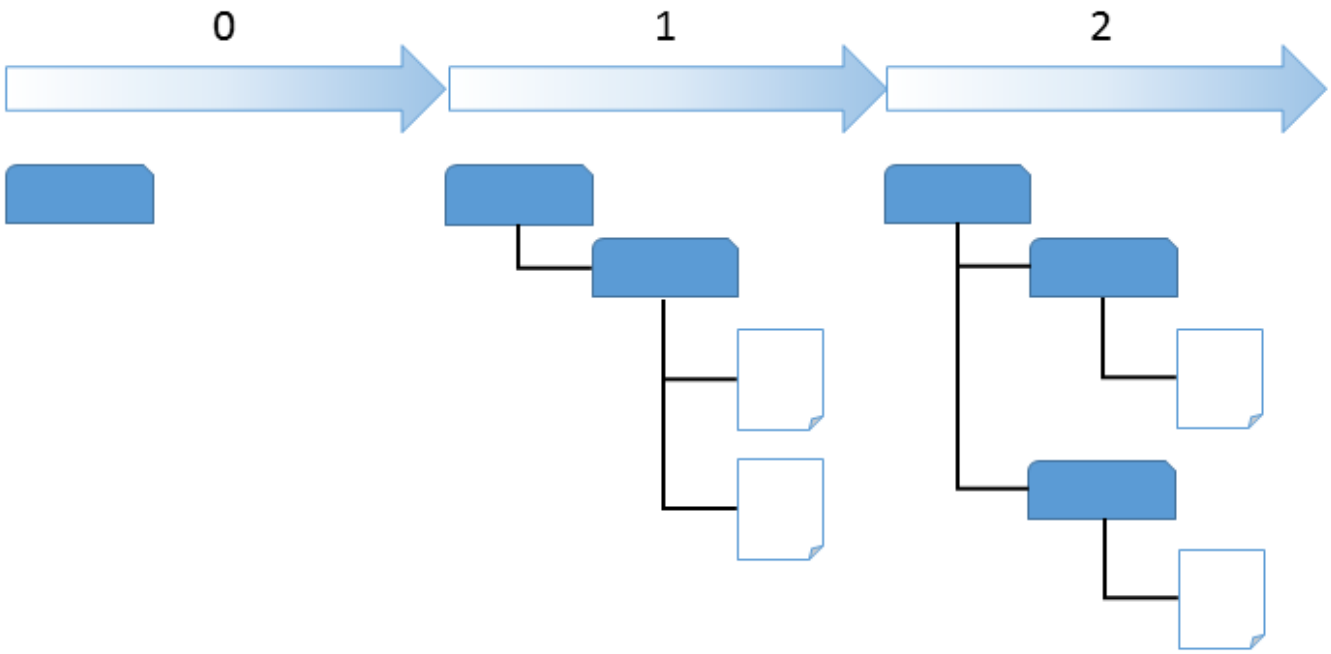
copy does not include the project's history, but does contain the copies of files that were located in the repository prior to any changes made by the developer. This enables complete visibility over any changes made to the files.

> 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 1.8 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.

Working with WorkspaceConsole

- **WorkspaceConsole settings**
- **WorkspaceConsole parameters**
- **Exporting packages from database**
- **Saving packages to the database**
- **Saving SVN packages**

WorkspaceConsole settings

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 workspaces and migrate them between applications.
- 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 with the target build version. For example, if the current version of the bpm'online build is 7.8.2.1035, and you need to update the packages to 7.8.2.1561, then you must use version 7.8.2.1561 of the utility.

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.

To do this – find the name of the current database connection string in the <db> XML element of the configuration file. Change the value of the connectionString attribute in the <connectionStrings> element so that it corresponds with the value found in the *ConnectionStrings.config* configuration file. For more information on modifying the settings in ConnectionStrings.config file, as well as their purpose, please refer to the "**How to deploy bpm'online on-site**" article. Below is an example of a configured *db* section of the Terrasoft.Tools.WorkspaceConsole.exe.config file:

```
<terrasoft>
 ...
  <db>
```

```
    <general connectionStringName="db" ... />
  </db>
</terrasoft>
<connectionStrings>
 ...
  <add name="db" connectionString="Data Source=DBServer; Integrated Security=SSPI;
Initial Catalog=DBName; MultipleActiveResultSets=True; Pooling=true; Async=true" />
</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.

The utility requires access to bpm'online libraries to work properly. Before you begin working with the utility, copy the contents of *[Path to application directory]\Terrasoft.WebApp\bin* to *[Path to application directory]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole*.

Select the version of *Terrasoft.Tools.WorkspaceConsole.exe* depending on whether you are using a 32-bit or 64-bit operating system. 32-bit and 64-bit files are located in *[Path to application directory]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\x86* and *[Path to application directory]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\x64* folders respectively.

To avoid any possible mistakes while copying the libraries manually, the *PrepareWorkspaceConsole.x86.bat* and *PrepareWorkspaceConsole.x64.bat* batch files are supplied with the *WorkspaceConsole* utility. A batch file that corresponds with the operating system version will prepare the *WorkspaceConsole* utility automatically by copying all the necessary libraries.

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.
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.
InstallFromSvn	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. Saves the latest version of the package structure and metadata from the zip archives into the configuration.
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.
InstallBundlePackages	Installs the set of comma-separated packages specified in the *-packageName* parameter to the workspace specified in the *-workspaceName* parameter.
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.
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**").

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.

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.

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]
```

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:\bpmonline7.9.2 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:\bpmonline7.9.2\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.Worksp
aceConsole.exe -operation=SaveDBContent -contentTypes=Repository -
workspaceName=Default -destinationPath=C:\SavedPackages
pause
```

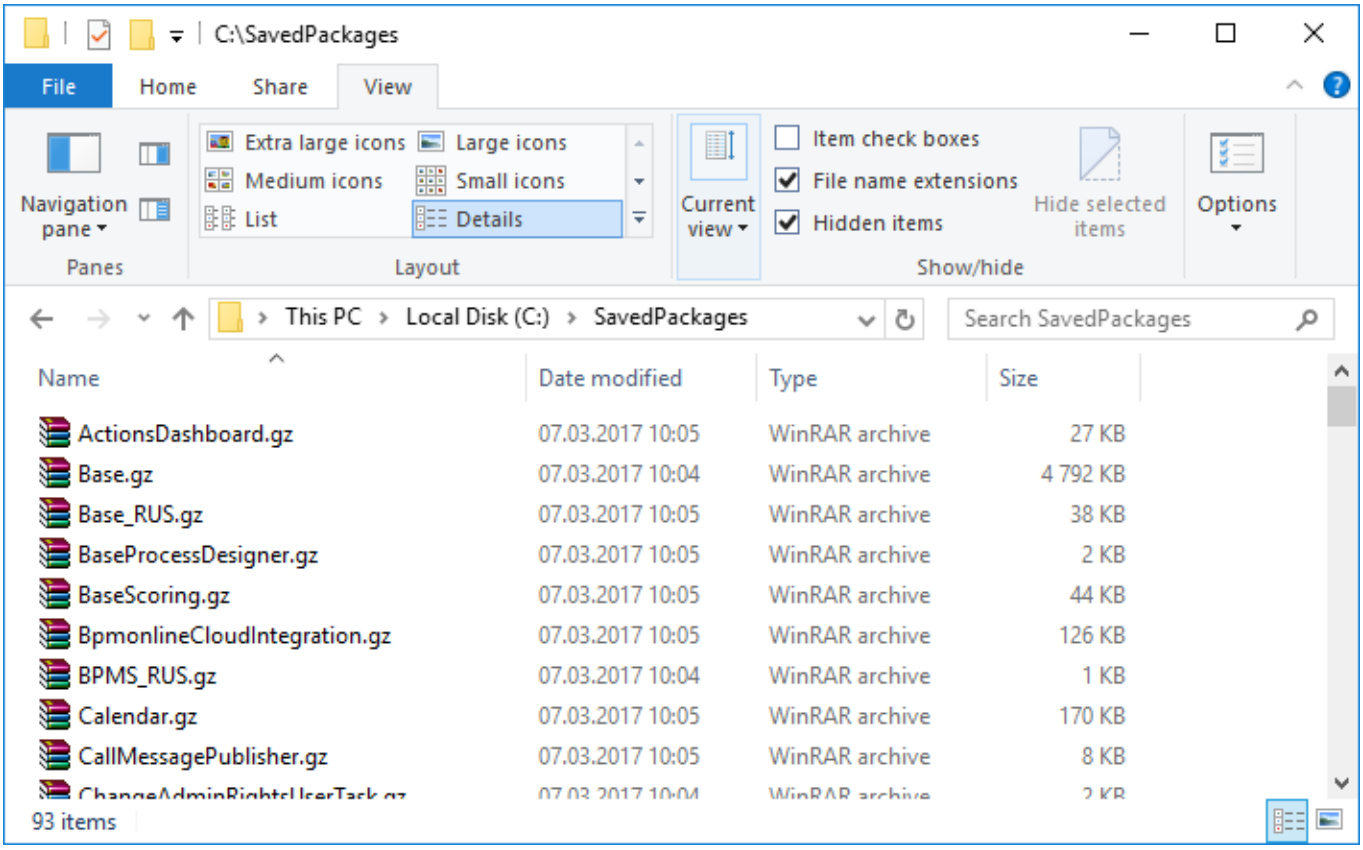
Upon saving the batch file and running it, a console window will appear, and the *WorkspaceConsole* execution process with specified parameter values will be displayed (Fig. 1).

Fig. 1. WorkspaceConsole execution

```
C:\WINDOWS\system32\cmd.exe - C:\bpmonline710\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe -opera...
c:\>C:\bpmonline710\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe -operation=SaveDBC
ontent -contentTypes=Repository -workspaceName=Default -destinationPath=C:\SavedPackages
Saving 'Default' workspace in repository 'C:\SavedPackages'
Added - schema 'SysPackageResourceChecksum' in package 'Base'
Added - schema 'AttributeValue' in package 'Base'
Added - schema 'ActivityResultEditPage' in package 'Base'
Added - schema 'Department' in package 'Base'
Added - schema 'ProductType' in package 'Base'
Added - schema 'Product' in package 'Base'
Added - schema 'UidBasedSinceSyncStrategy' in package 'Base'
Added - schema 'CreateSocialAccountUserTask' in package 'Base'
Added - schema 'VwSysEntitySchemaInPackage' in package 'Base'
Added - schema 'SysModuleAnalyticsReportLczOld' in package 'Base'
```

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

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.

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 -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**").

Saving packages to the database

Case description

The bpm'online application is installed in the C:\bpmonline7.9.2 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.

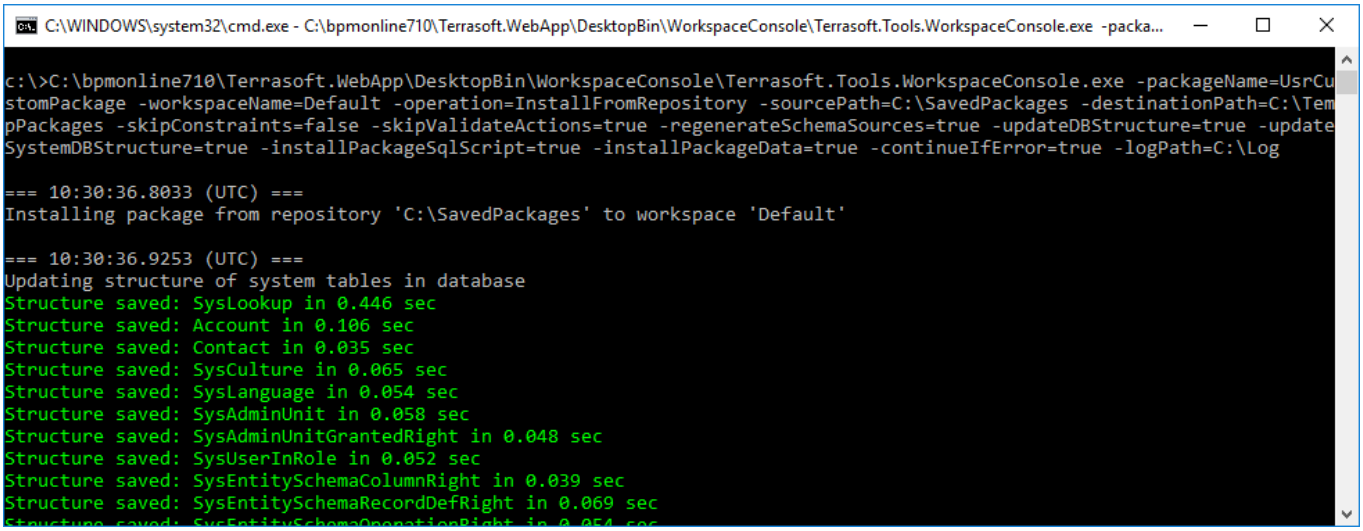
Case implementation:

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:\bpmonline7.9.2\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.Worksp
aceConsole.exe -packageName=userPackage -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 -
logPath=C:\Log
pause
```

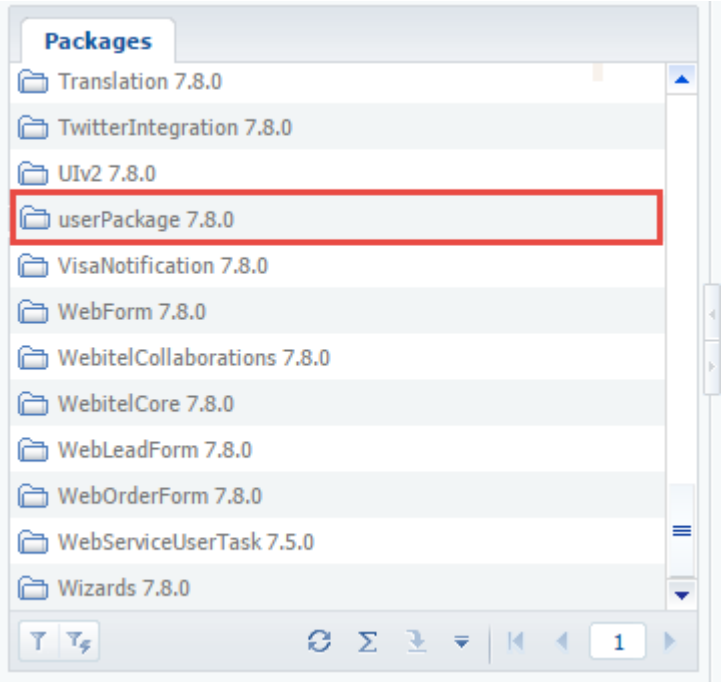
Upon saving the batch file and running it, a console window will appear, and the *WorkspaceConsole* execution process with specified parameter values will be displayed (Fig. 1).

Fig. 1. Saving a package to the application database



Run the command to load the *userPackage* package to the *Default* configuration.

Fig. 2. The userPackage package in the [Configuration] section



Saving SVN packages

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:

```
[Путь к WorkspaceConsole]\Terrasoft.Tools.WorkspaceConsole.exe -
operation=SaveVersionSvnContent -destinationPath=[Путь к локальному каталогу] -
workingCopyPath=[Путь к локальному каталогу] -sourcePath=[Путь к SVN-хранилищу] -
packageName=somePackage -packageVersion=7.8.0 -sourceControlLogin=User -
sourceControlPassword=Password -logPath=[Путь к локальному каталогу] -cultureName=ru-
RU -excludeDependentPackages=true
```

Uploading packages to a file system

Case description

Save the userPackage package to the C:\SavedPackages directory from the SVR repository, found at http://server-svn:8050/svn/Packages. Language culture - Spanish. 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:\bpmonline7.9.2 directory.

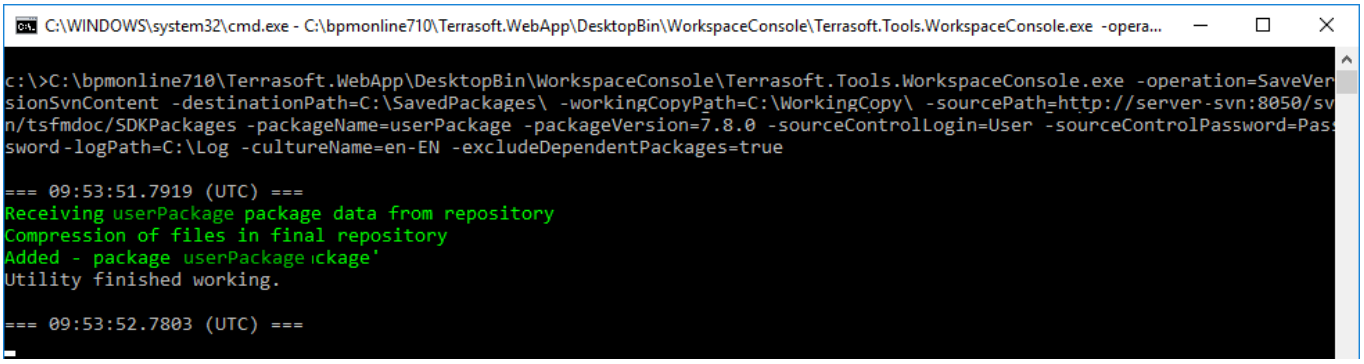
Case implementation:

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:\bpmonline7.9.2\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.Worksp
aceConsole.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
```

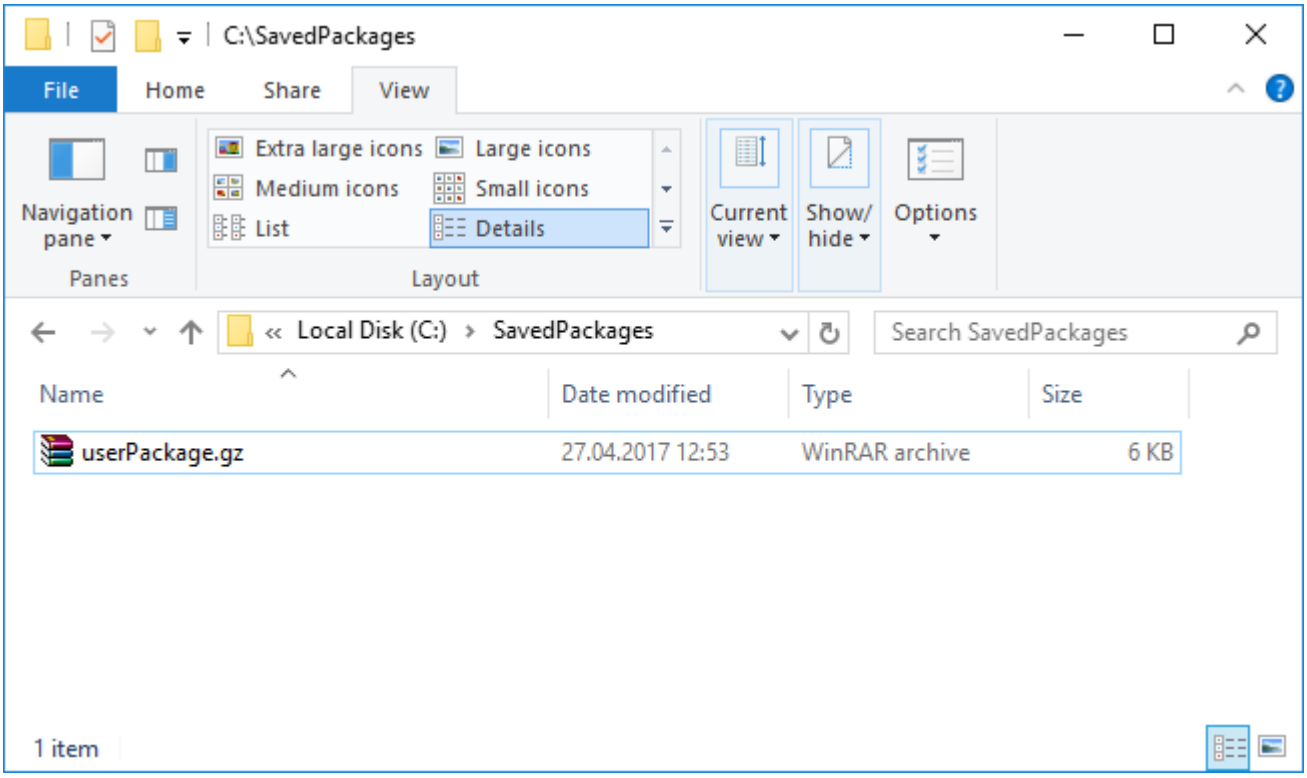
Upon saving the batch file and running it, a console window will appear, and the *WorkspaceConsole* execution process with specified parameter values will be displayed (Fig. 1).

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

General

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

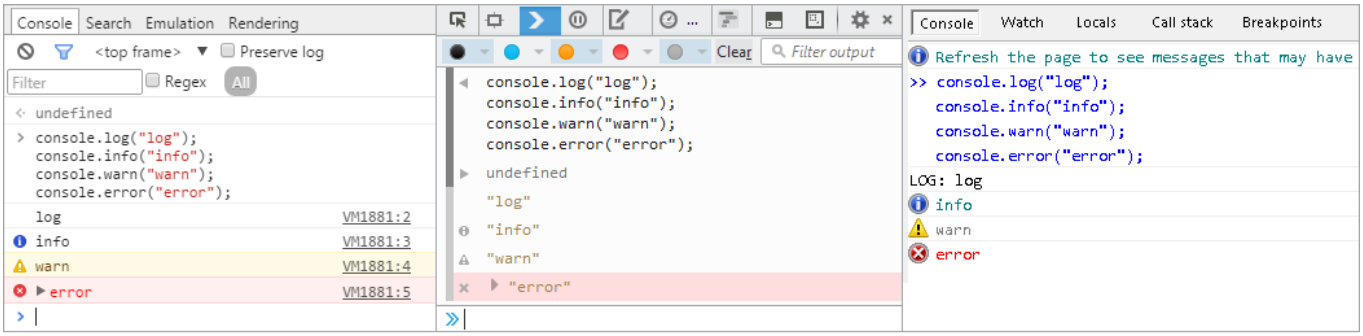
You can enable debugging information of a different nature, i.e. info messages, warnings and error messages, in the console. For this purpose you can use corresponding console object methods (table 1).

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**").

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:

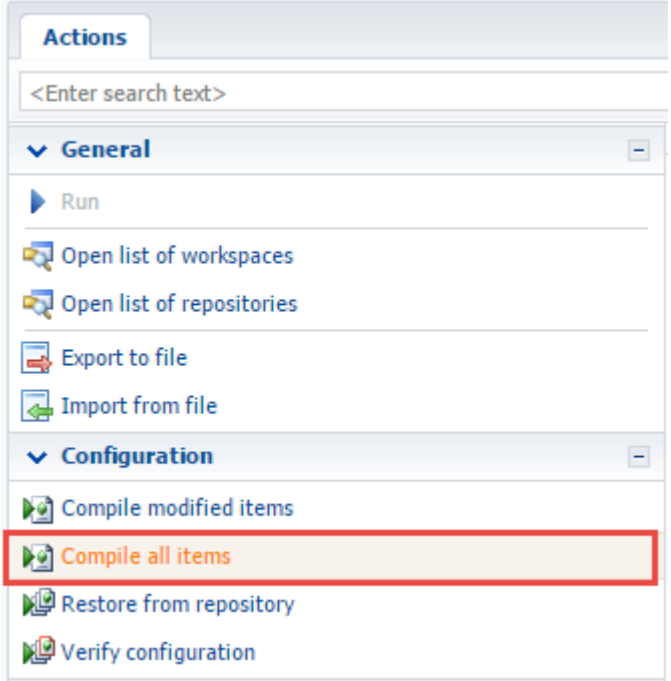
- To configure *IncludeDebugInformation*, specify the "*true*" value.
- To configure *CompilerSourcesTempFolderPath*, specify the path to the directory where the source files will be exported.
- To configure *ExtractAllCompilerSources*, set the value to *"true"* if you want to export all schemas when 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.

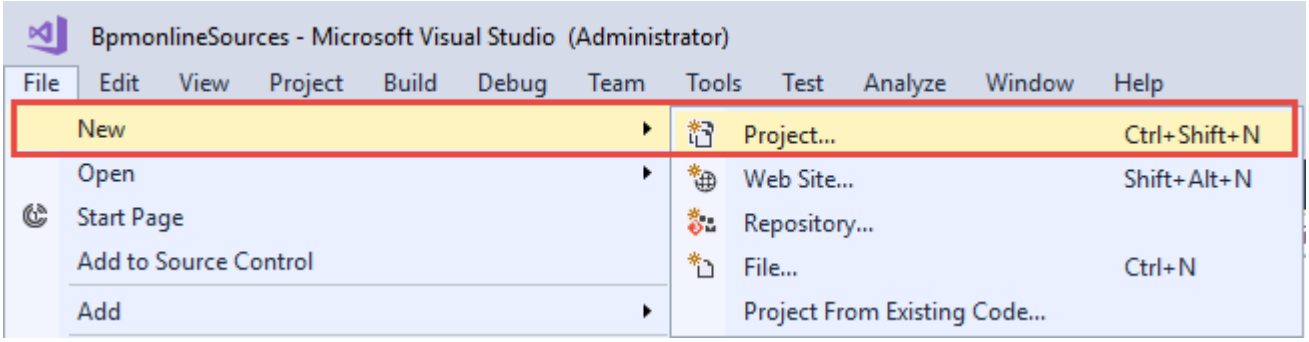
For example: Contact.Base_Entity.cs, ContractReport.Base_Report.cs.

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.

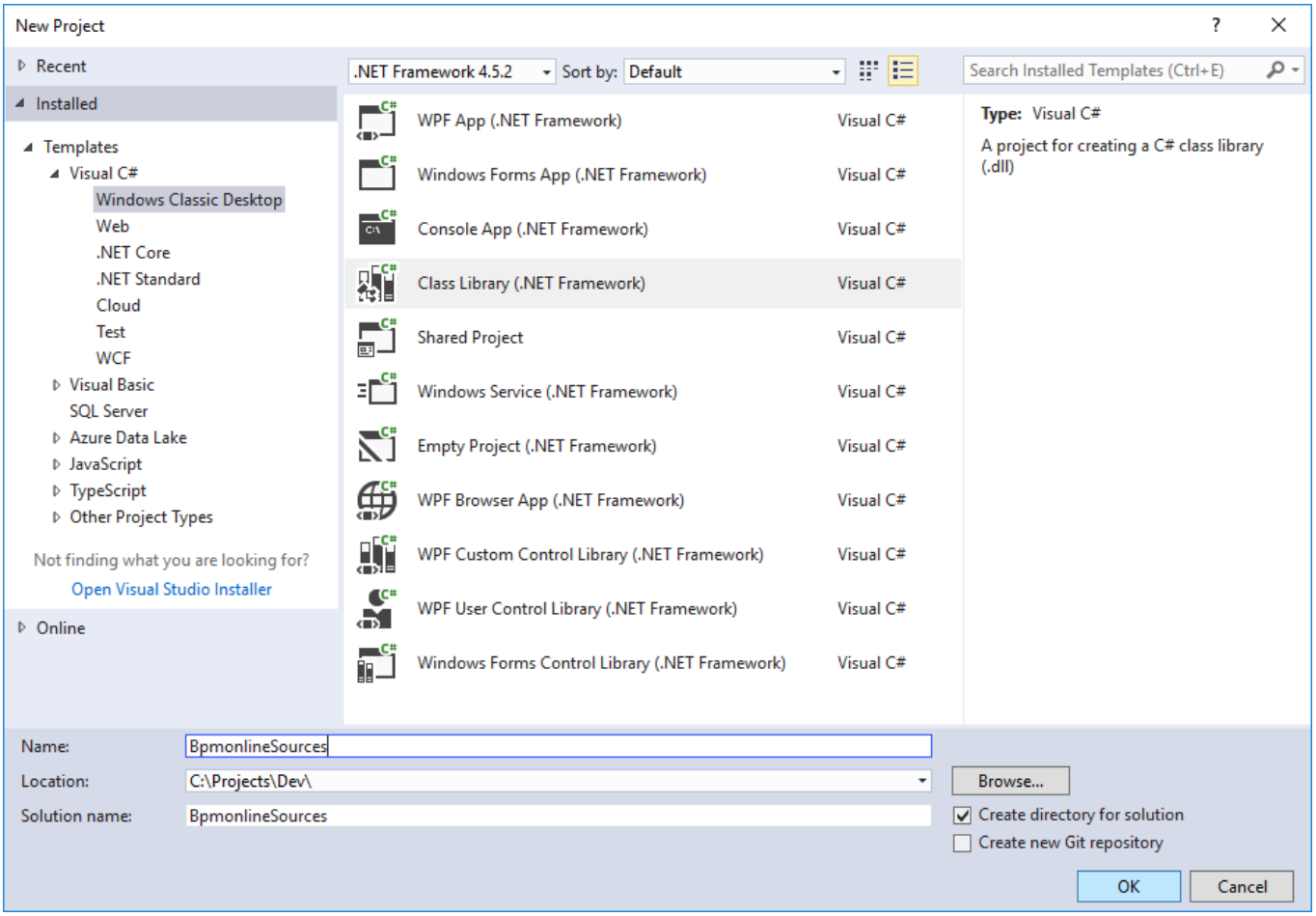
To create a project for debugging an application in Visual Studio, execute the *File* > *New* > *Project*menu command (Fig. 2).

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



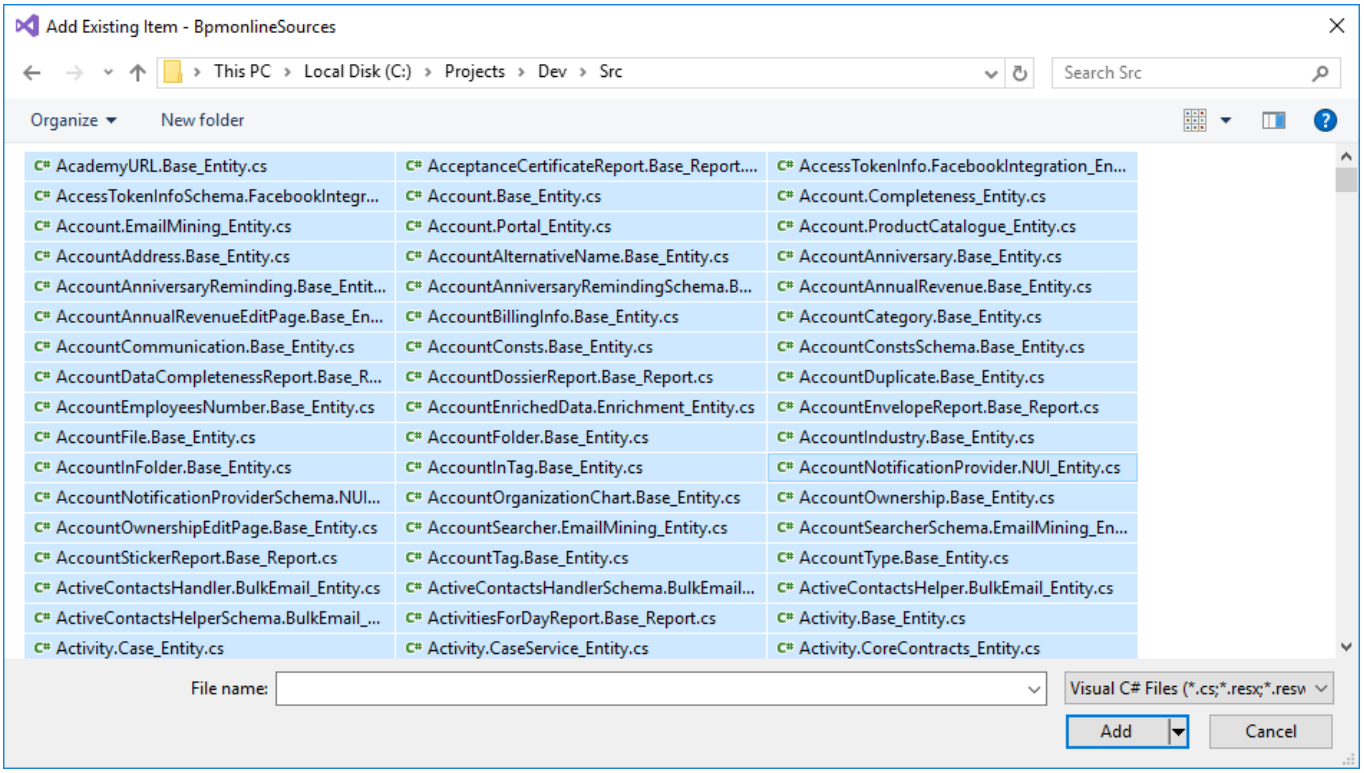
After creating the project, you need to remove an extra file from it (by default, the file Class1.cs is added to the new

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



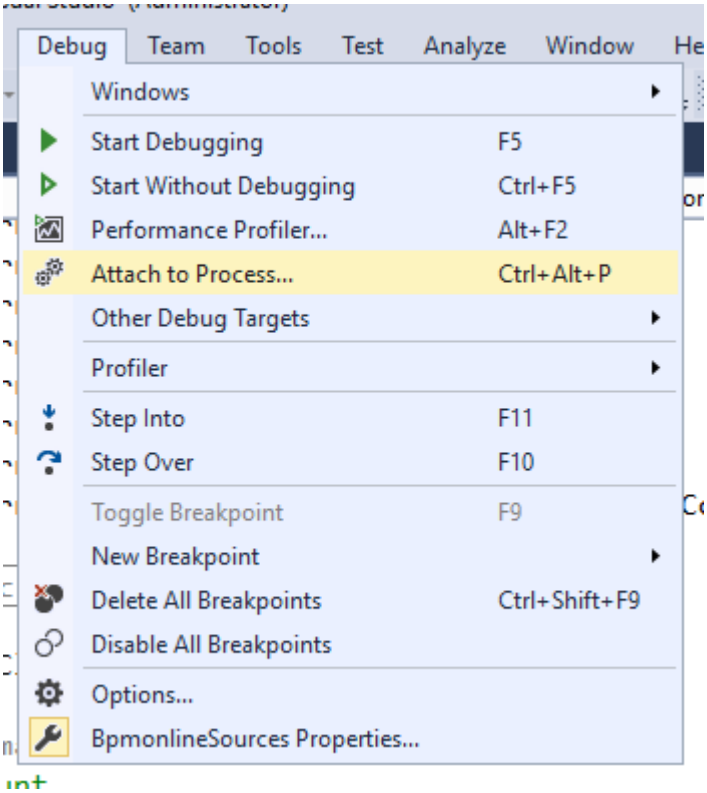
> NOTE
>
> Add only the files needed for debugging to the Visual Studio project. However, the transition between methods during debugging will be limited only by the methods of classes implemented in the files added to the project.

Save the project after adding the files.

4. Attaching to the IIS process for debugging

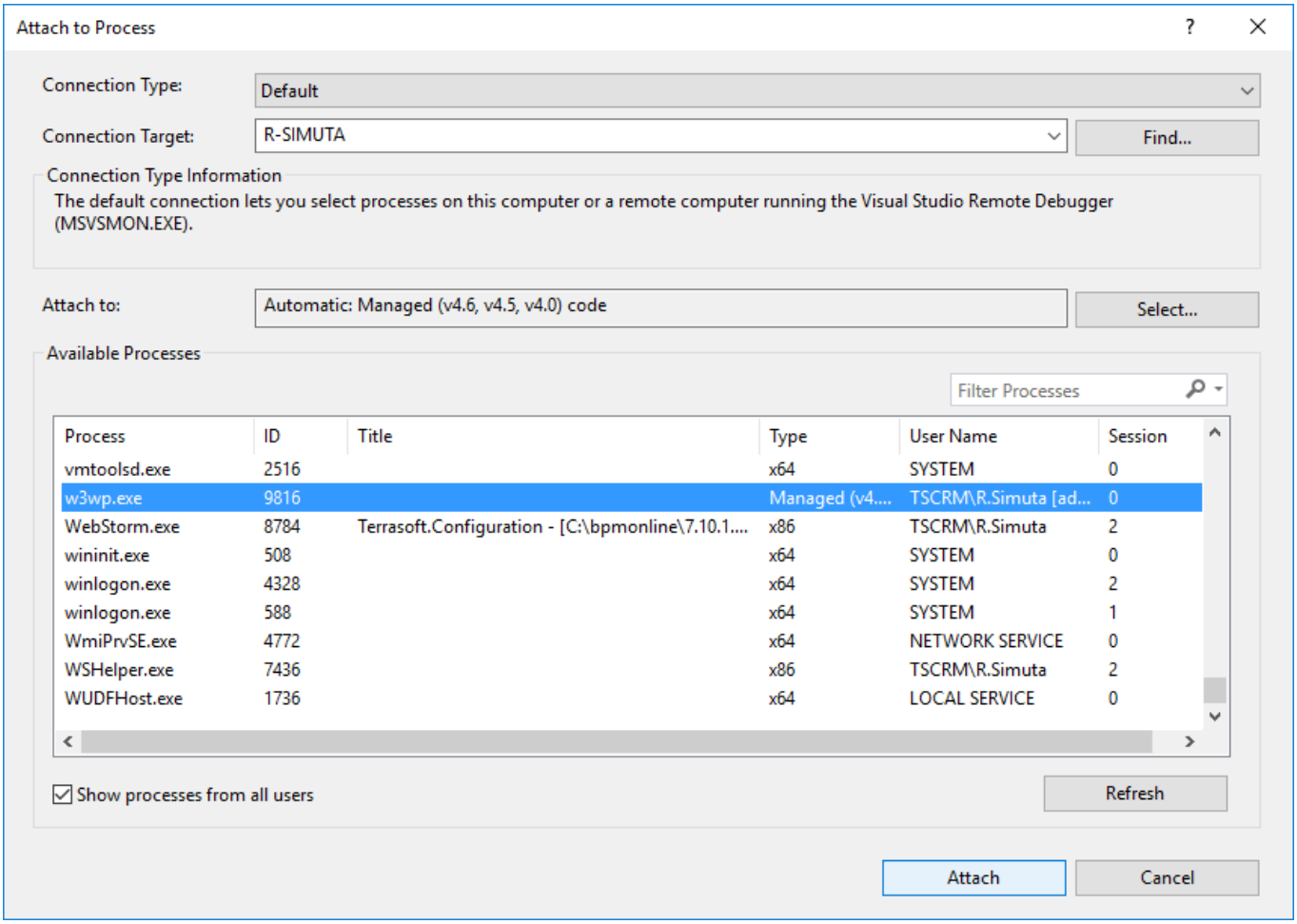
To begin debugging, attach to the IIS server process, where the application runs. To do this, select the *Debug* > *Attach to process* command in the Visual Studio menu (Fig. 5).

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



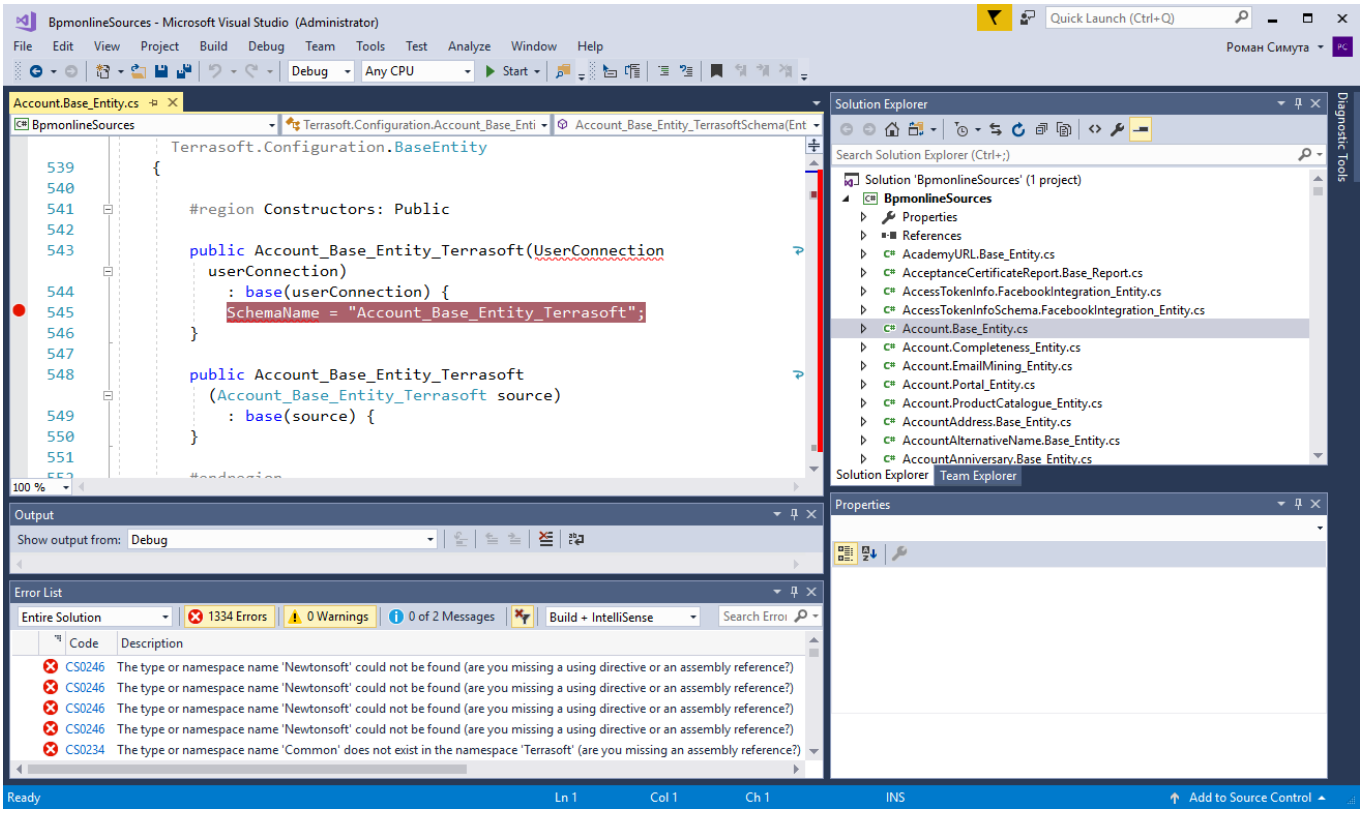
ATTENTION

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).

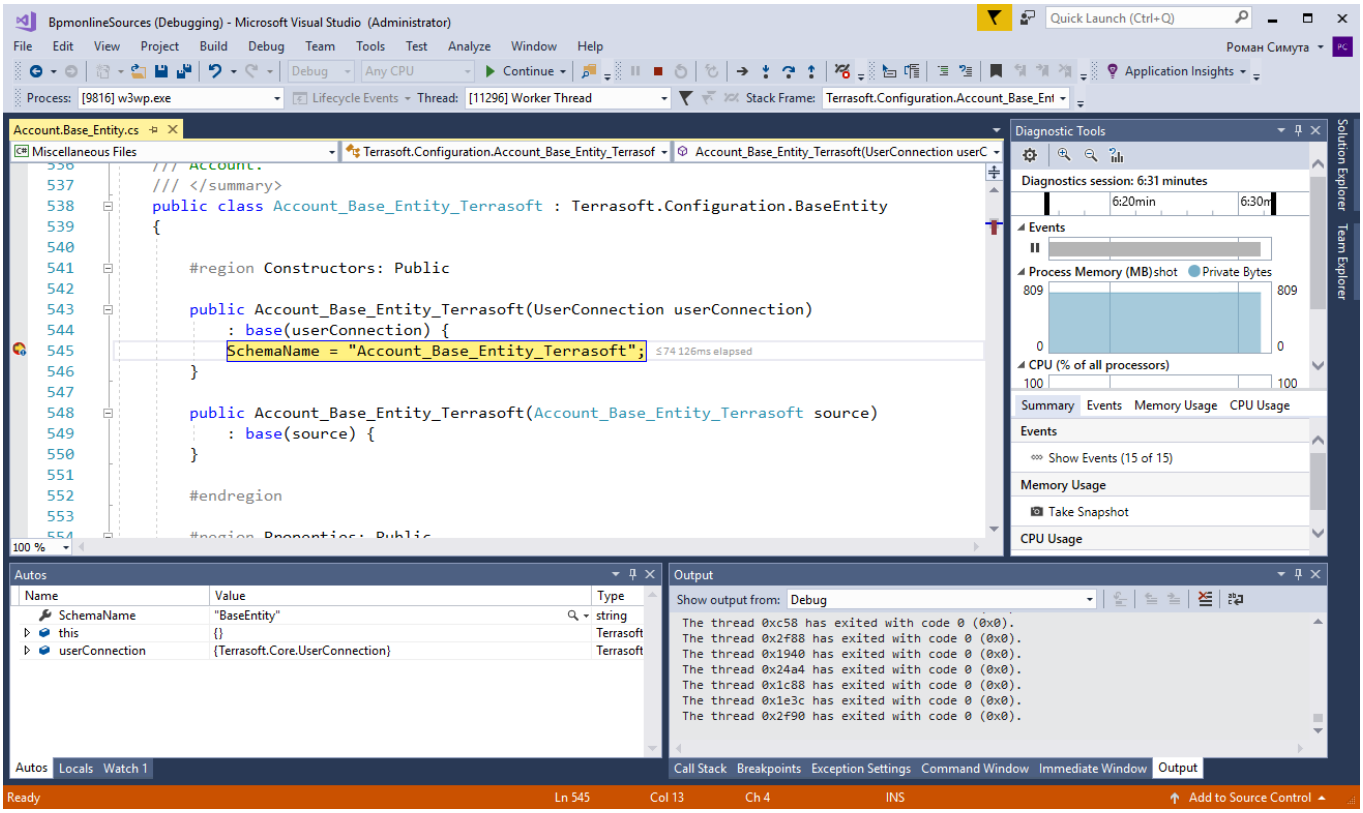
After you attach to the IIS process, you can start debugging. To do this, open the file with the desired source code and set a breakpoint (Fig. 7).

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).

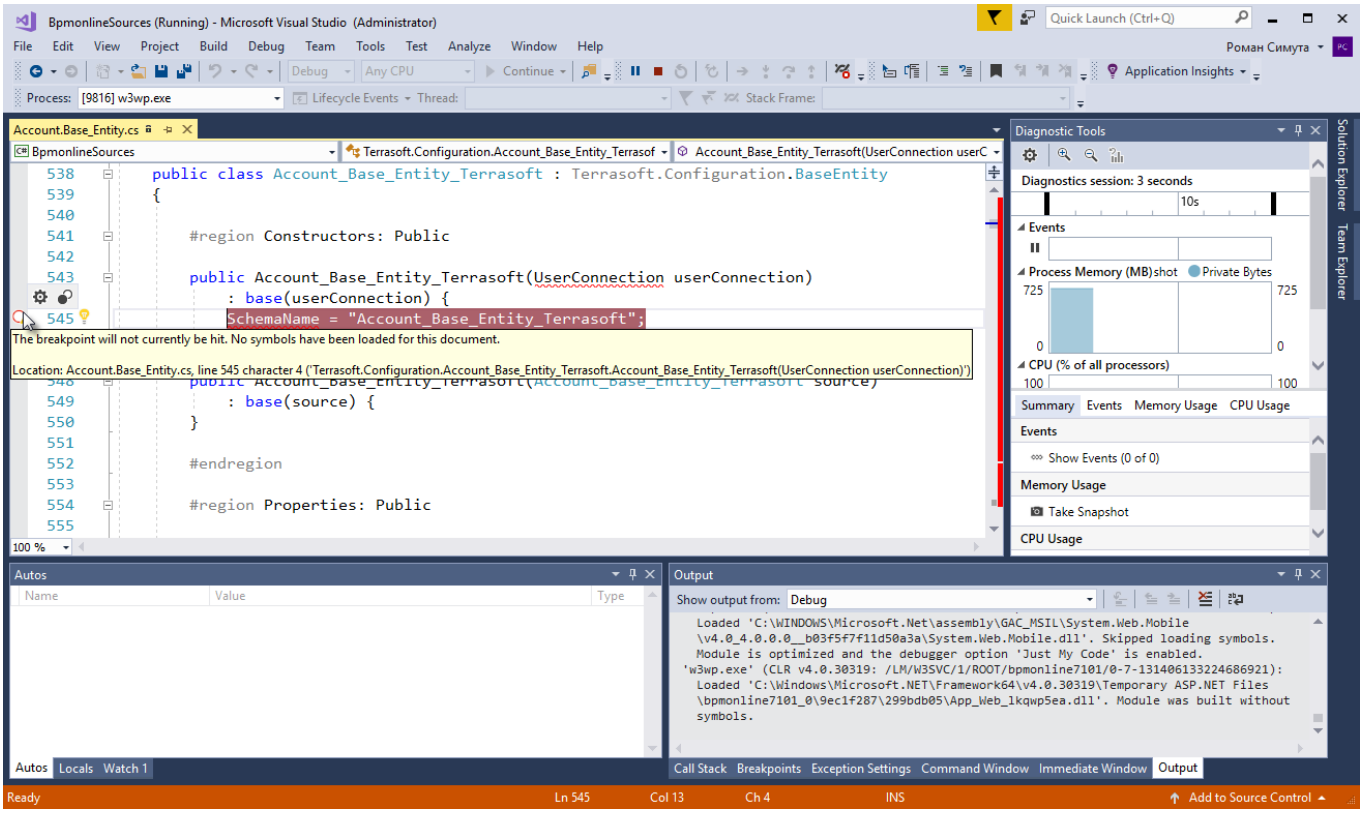
Fig. 8 Interrupting the execution of the program on the breakpoint



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



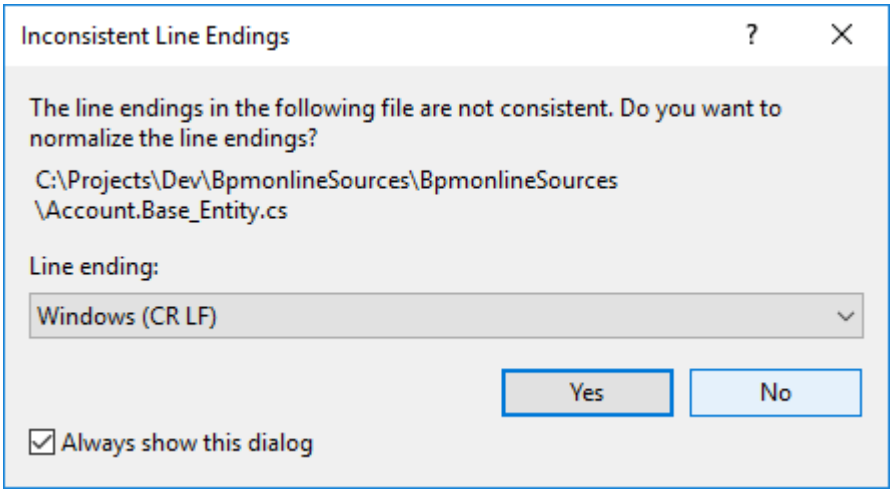
If the hint contains a message that the symbol information was not loaded (Fig. 8), it is necessary to do the following:

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.

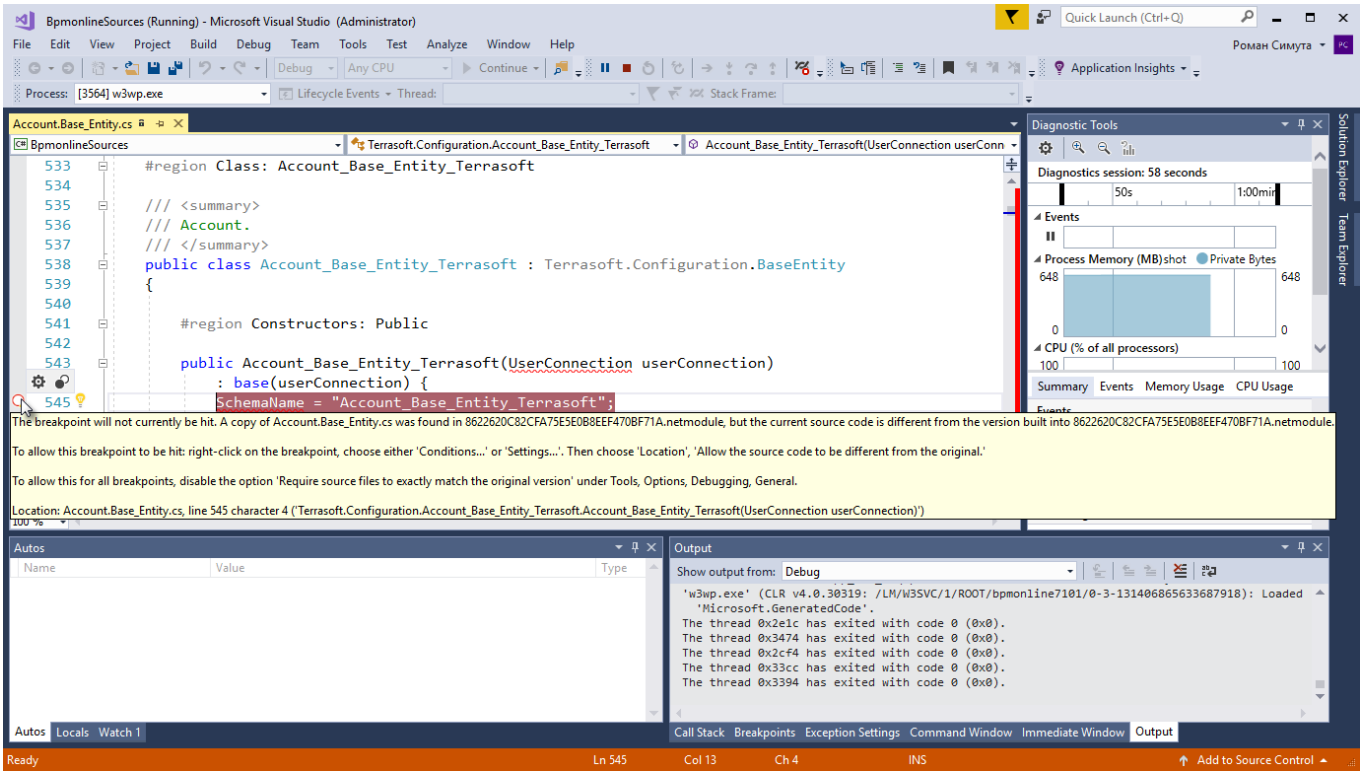
After the file with the source code is reopened, a message about non-uniform end-of-line characters may appear (Fig. 10).

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

- **Section business logic**
- **Page configuration**
- **Adding details**
- **Business processes**
- **Typical customizations**
- **Analytics**
- **Working with data**
- **Mobile application**
- **Service products customization**
- **Lending product customization**

Section business logic

- **Creating 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**
- **How to add a fixed filters block to the section**

Creating new section

General provisions

Creating new sections is one of the most common bpm'online development tasks. Use the Section Wizard to create new sections quickly. Using the wizard, not only can you add an existing section to a new workplace, but also set up list, record edit pages, and details.

As a result, new object, page and detail schemas will be added to the current custom package.

The list object schemas created by the wizard when adding new sections is available in table 1, and the list of created client schemas is available in table 2.

Table 1. Object schemas created by the section wizard

Naming rules	Purpose	Parent
[Section object name]	Section primary object	Base object
[Section object name]Folder	Object folder. Service object for correct grouping of section records. Forms general structure of section folders.	Base folder (BaseFolder)
[Section object name]inFolder	Object in folder. Service object for correct grouping of section records. Determines connection between section records and section folders, in which the records are included.	Base element in folder (BaseItemInFolder)
[Section object name]File	Object for [Attachments] detail of the section.	File (File)
[Section object name]Tag	Section tag.	Base tag (BaseTag)

[Section object name]inTag	Tag in the section object.	Base tag in base object (BaseEntityInTag)

Table 2. Client schemas created with section wizard

Naming rules	Purpose	Parent
[Section object name]Section	Section schema	Base section schema (BaseSectionV2)
[Section object name]Page	Section edit page schema	BaseModulePageV2
[Section object name]Detail	Section detail schema	Base detail with list schema (BaseGridDetail)

Sequence for adding sections

1. Create a section using the Section Wizard and add it to the required workplace.
2. Add required columns to the section object schema and set up their display in the list, page and details.

Case description

In the application, add a [Car showroom] workplace. Add a [Trucks] section to the workplace. The truck object schema must contain the following required columns:

- Name – string.
- Owner – [Contact] lookup.
- Organization – [Account] lookup.
- Price – string.

Case implementation algorithm

1. Create a section using the Section Wizard and add it to a workplace

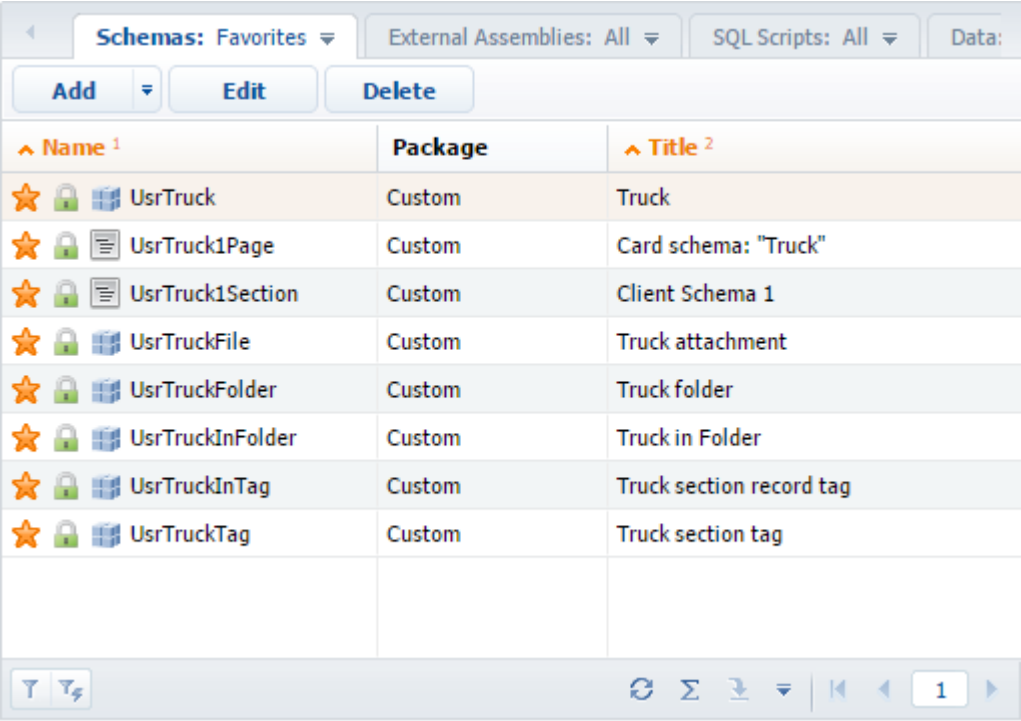
To create a new workplace, enter *"Car showroom"* in the [Name] field and add a user group that can access this workplace.

To create a new section, it is sufficient to complete the first page of the Wizard. The object of the new section will have the "Name" string column and the columns inherited from the base object. For the initial section setup, enter *"Truck*" in the [Code] field and *"Trucks"* in the [Title] field.

> NOTE
>
> If you did not change any default settings, the system will inform you that the Usr prefix is required when entering the [Section code] value. The required prefix is set in the "Object name prefix" system setting. If needed, you can modify prefix value to user prefix.

After saving the results in the custom package all schemas that are required for base section logic will be created automatically (Fig. 1).

Fig. 1. Schemas of the [Trucks] section in the custom package



2. Add required columns to section object schema and set up how they are displayed

There are two ways you can add new columns to an object schema.

The first method is to use the Section Wizard, namely the page field setup tab. With the Section Wizard, you can add a new column and add it to the edit page. The column will be automatically added to the section primary object schema. For more information on section page field setup, please refer to the "How to set up page fields" article.

The alternative method of adding columns is to use the Object Designer in the [Configuration] section to add columns to the page. For more information about the functions of the [Configuration] section, please see the "**General information about the [Configuration] section**" article.

Since you need to use the Section Wizard in both cases, the first method is more advisable.

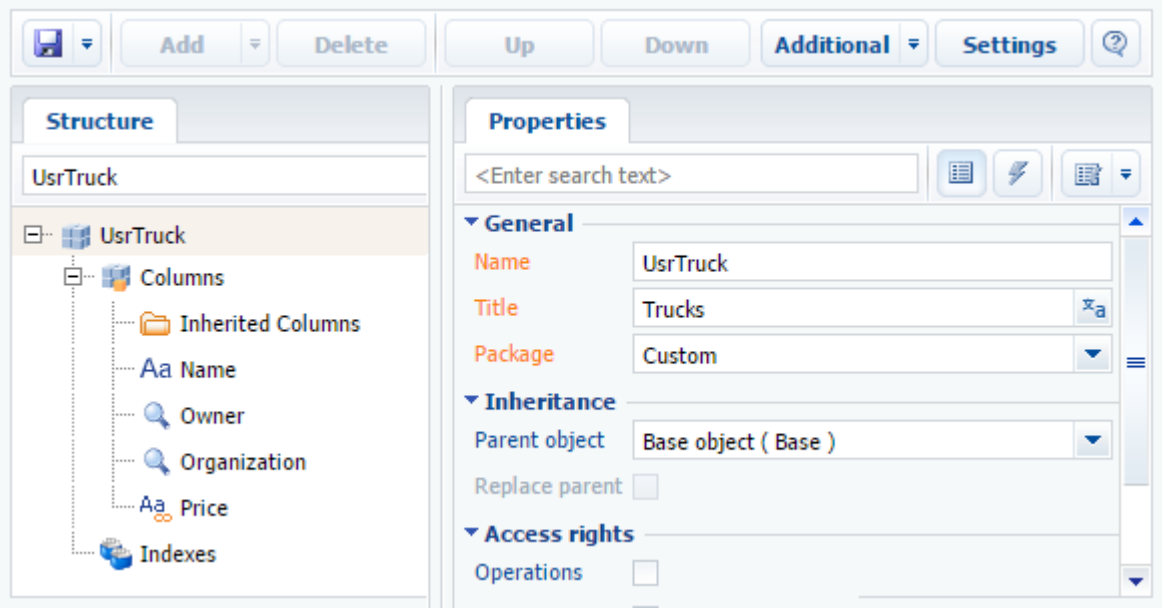
Since the [Name] column is added to the section pages automatically by the Section Wizard, you need to add the remaining columns manually.

When adding the [Owner] column, select the "Contact" lookup as the column type, and then enter the column name and title. Similarly, for the [Organization] column, select the "Account" as the lookup object, and then enter the title and name of the column. The [Price] column must be a string with the corresponding title and name.

Select the [Required] checkbox for all added columns.

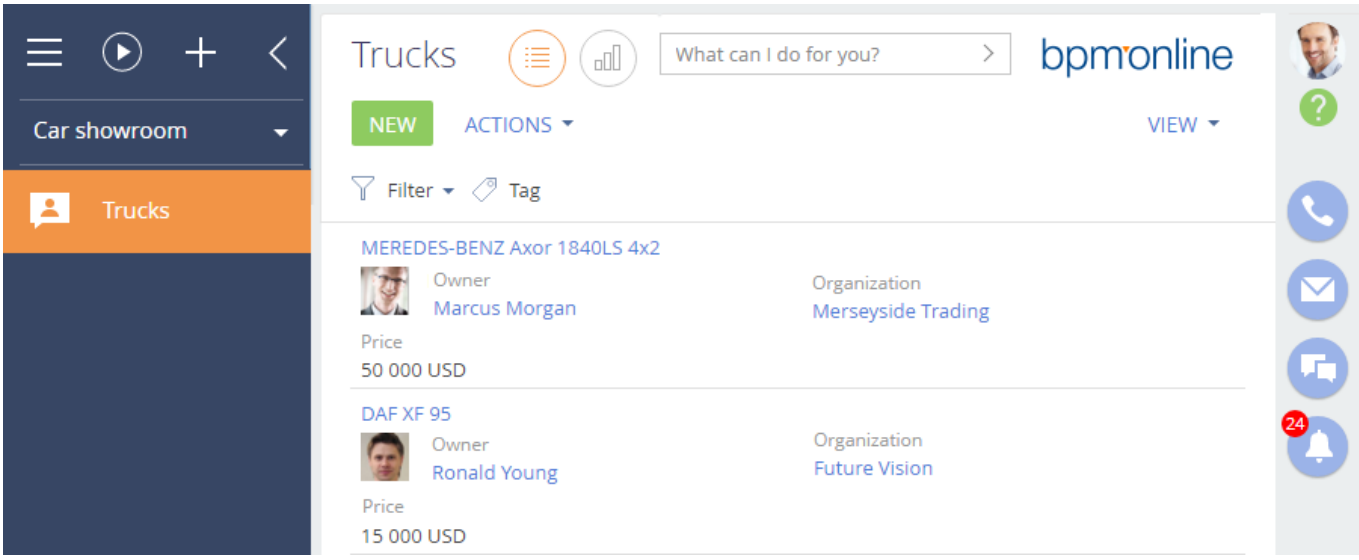
After saving the results, new columns will be added to the *Truck* object schema (Fig. 2), and the *diff* array of the *TruckPage* edit page schema will contain all corresponding configuration objects.

Fig. 2. [Trucks] section primary object schema



After adding new records, the section will look like the one in Fig. 3.

Fig. 3. The [Trucks] section in the [Car showroom] workplace



Adding an action to the list

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 the action displaying the order creation date in the data window for the [Orders] section. The action must be available only for orders at the [In progress] stage.

> NOTE
>
> A selected record is addressed via the ***ActiveRow*** attribute of the section view model which returns the value in the primary column of the selected record. This value may be used to receive values in other fields of the selected object, for example, from the list data collection stored in the ***GridData*** property of the list view model.

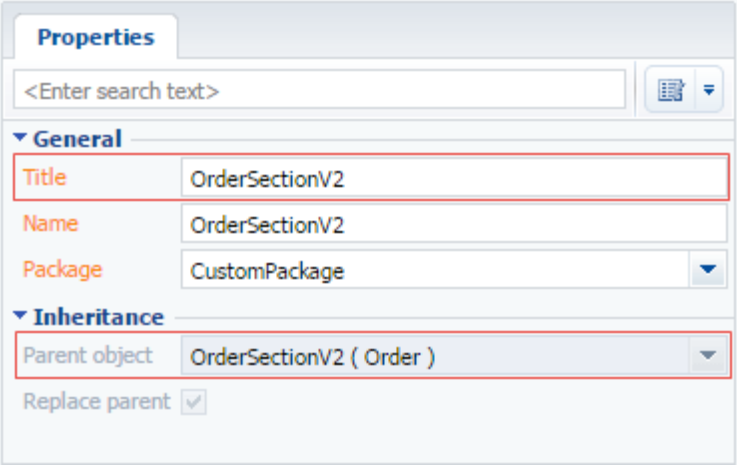
Case implementation algorithm

1. Create a replacing page for the [Orders] section in a custom package

A replacing client module must be created and [OrderSectionV2] must be specified as the parent object in it (Fig. 1).

The procedure for creating the replacing page is described in the article **Creating a custom client module schema**.

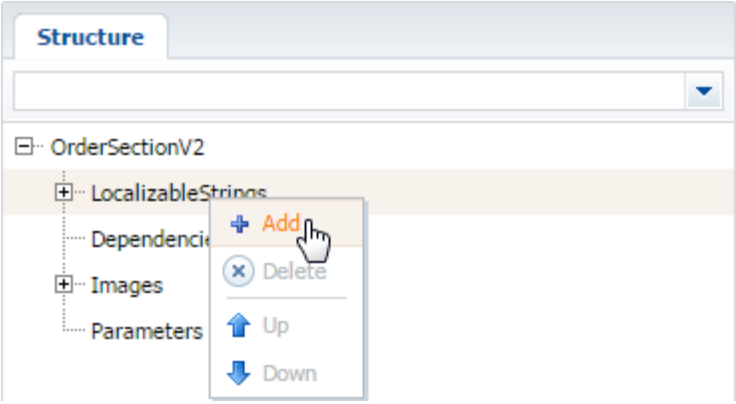
Fig. 1. — Properties of the replacing page for the [Orders] section



2. Add a string with the [Actions] menu item to the localizable string collection of the section replacing schema

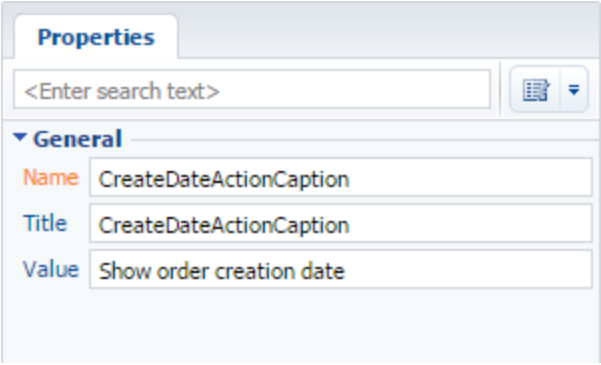
For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure node (Fig. 2).

Fig. 2 — Adding a localizable string to the schema



Fill in the properties for the created string 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

- isRunning checks whether the selected order is at the Execute stage;
- isCustomActionEnabled checks the availability of the added menu item;
- ShowOrderInfo action handler method displays the scheduled end date of the selected order in the data window.

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("OrderSectionV2", ["OrderConfigurationConstants"],
    function(OrderConfigurationConstants) {
        return {
            // Name of the section schema.
            entitySchemaName: "Order",
            // Methods collection of the section view model.
            methods: {
                // Method checking the stage, at which this order is.
                // A value of the primary column of the selected list record is
passed as a parameter.
               isRunning: function(activeRowId) {
                   // Receiving the collection of data on list view of the [Orders]
section list.
                   var gridData = this.get("GridData");
                   // Receiving a model of the order selected by the set value of the
primary column.
                   var selectedOrder = gridData.get(activeRowId);
                   // Receving the model property - status of the selected order.
                   var selectedOrderStatus = selectedOrder.get("Status");
                   // The value of the selected order status is compared to the value
of the "In progress" type and
                   // the method returns true or false depending on the comparison
```

```
result.
                    return selectedOrderStatus.value ===
OrderConfigurationConstants.Order.OrderStatus.Running;
                },
                // The method determines whether the menu item will be available.
                isCustomActionEnabled: function() {
                    // An attempt to receive the active record Id is made.
                    var activeRowId = this.get("ActiveRow");
                    // If the Id is determined (i.e. a record is selected in the list)
and the selected order has
                    // the "In progress" status, true is returned; otherwise false is
returned.
                    return activeRowId ? this.isRunning(activeRowId) : false;
                },
                // Action handler method displaying the order creation date in the
data window.
                showOrderInfo: function() {
                    var activeRowId = this.get("ActiveRow");
                    var gridData = this.get("GridData");
                    // Receiving the order creation date.
                    var dueDate = gridData.get(activeRowId).get("Date");
                    // Calling the standard system method to display the data window.
                    this.showInformationDialog(dueDate);
                }
            }
        };
});
```

4. Override the getSectionActions virtual method wherein the handler method must be bound to an action

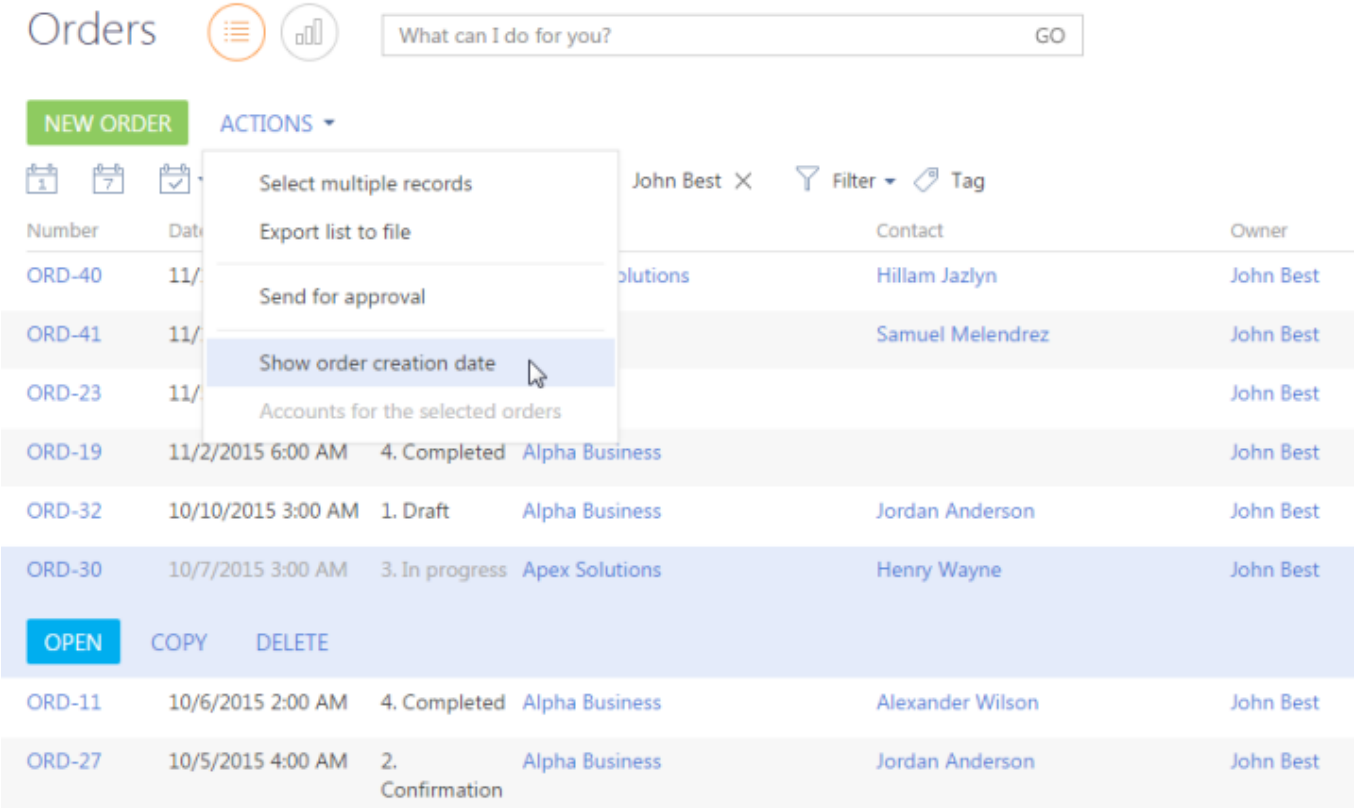
For this purpose, add the code of the overridden method to the methods collection.

```
// Overriding of the base virtual method which returns the section actions
collection.
getSectionActions: function() {
    // The parent implementation of the method is called to receive
    // the collection of initiatited actions for the base section.
    var actionMenuItems = this.callParent(arguments);
    // Adding a separator line for visual separation of a custom action from the list
    // of actions of the base section.
    actionMenuItems.addItem(this.getButtonMenuItem({
        Type: "Terrasoft.MenuSeparator",
        Caption: ""
    }));
    // Adding a menu item into the section actions list.
    actionMenuItems.addItem(this.getButtonMenuItem({
        // Binding the menu item title to the localizable schema string.
        "Caption": {bindTo: "Resources.Strings.CreateDateActionCaption"},
        // Binding the action handler method.
        "Click": {bindTo: "showOrderInfo"},
        // Binding the property of the menu item availability to the value which
returns the isCustomActionEnabled method.
        "Enabled": {bindTo: "isCustomActionEnabled"}
    }));
    return actionMenuItems;
}
```

5. Save the created replacing schema

When the schema is saved and the system web-page is updated, a new action will appear in the [Orders] section. It will be active when selecting an order at the "In progress" stage (Fig. 4).

Fig. 4. – Demonstrating the case implementation result



How to add a section action: handling the selection of several records

Mode of selecting multiple records

The mode of selecting one record is used by default for the list. To select simultaneously several active records in the list, the [Select multiple records] item must be selected in the [Actions] menu of the section. Then, the visual representation of the list changes and checkboxes for selecting records appear. The mode of selecting several records is cancelled using the same [Actions] menu, [Cancel multiple selection] item.

Case description

Implement the action for displaying accounts of several orders selected in the list in the data window for the [Orders] section.

> NOTE
>
> The value of primary columns for the selected records is stored in the ***SelectedRows*** property of the section view model. These values may be used to receive values in other fields of selected objects, for example, from the list registry data collection stored in the ***GridData*** property of the list view model.

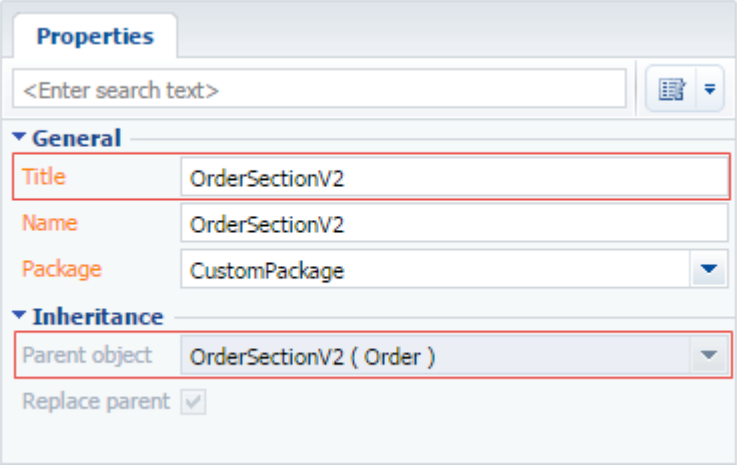
Case implementation algorithm

1. Create the [Orders] replacingsection page in a custom package

A replacing client module must be created and [OrderSectionV2] 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**.

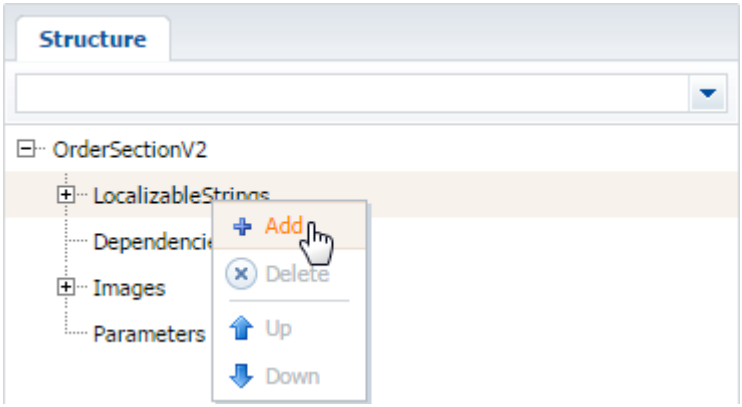
Fig. 1. — Properties of the [Orders] section replacing page



2. Add a string with the [Actions] menu item to the collection of localizable strings in the section replacement schemareplacing schema

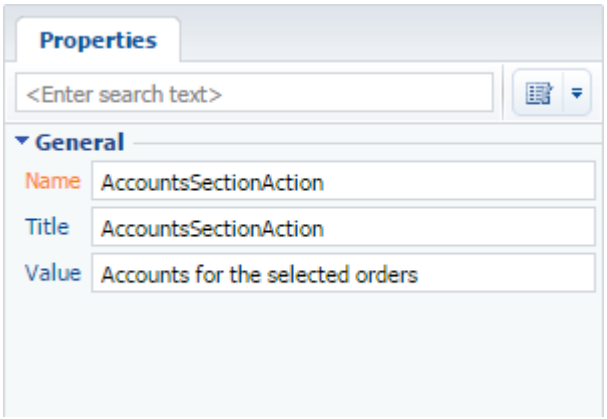
For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure node (Fig. 2).

Fig. 2 — Adding a localizable string to the schema



Fill in the properties for the created string as shown in Fig. 3.

Fig. 3. — Properties of a custom localized string



3. Add the method implementation to the methods collection of the section view

- isCustomActionEnabled checks the availability of the added menu item;
- ShowOrderInfo action handler method displays the scheduled end date of the selected order in the data window.

For this purpose, add the program code of the section replacement module to the source code tab. Required methods are added to the methods collection of the view model.

```
define("OrderSectionV2", ["OrderConfigurationConstants"],
    function(OrderConfigurationConstants) {
        return {
            // Section name schema.
            entitySchemaName: "Order",
            // Model methods collection of section view.
            methods: {
               // Method defines if the item menue is avalable.
               isCustomActionEnabled: function() {
                   // Trying to receive Id massive for the selected records.
                   var selectedRows = this.get("SelectedRows");
                   // If the massiv contains items (i.e. at least one record is
selected in the list),
                   // true value returns, othervise false value applies.
                   return selectedRows ? (selectedRows.length > 0) : false;
               },
               // Action method-handler that displays a list of accounts in the
informational window.
               showOrdersInfo: function() {
                   // Receiving a massive of initial values for the columns of the
selected records.
                   var selectedRows = this.get("SelectedRows");
                   // Receiving data collection of the list records.
                   var gridData = this.get("GridData");
                   // Variable for object model storage of the selected order.
                   var selectedOrder = null;
                   // Variable to store the name of the account of the selected
order.
                   var selectedOrderAccount = "";
                   // Variable to form the text with the accounts names for the
selected object.
                   var infoText = "";
                   // Handling massive values of initial columns for the selected
list records.
                   selectedRows.forEach(function(selectedRowId) {
                       // Receiving object model for the selected record.
                       selectedOrder = gridData.get(selectedRowId);
                       // Receiving account's name for the selected order.
                       selectedOrderAccount =
selectedOrder.get("Account").displayValue;
                       // Adding account's name to the string that will be displayed
at the info window.
                       infoText += "\n" + selectedOrderAccount;
                   });
                   // Standart system method calling to display info window.
                   this.showInformationDialog(dueDate);
               }
            }
        };
});
```

4. Redefine the getSectionActions virtual method wherein the handler method must be bound to an action

For this purpose, add the code of the redefined method to the methods collection.

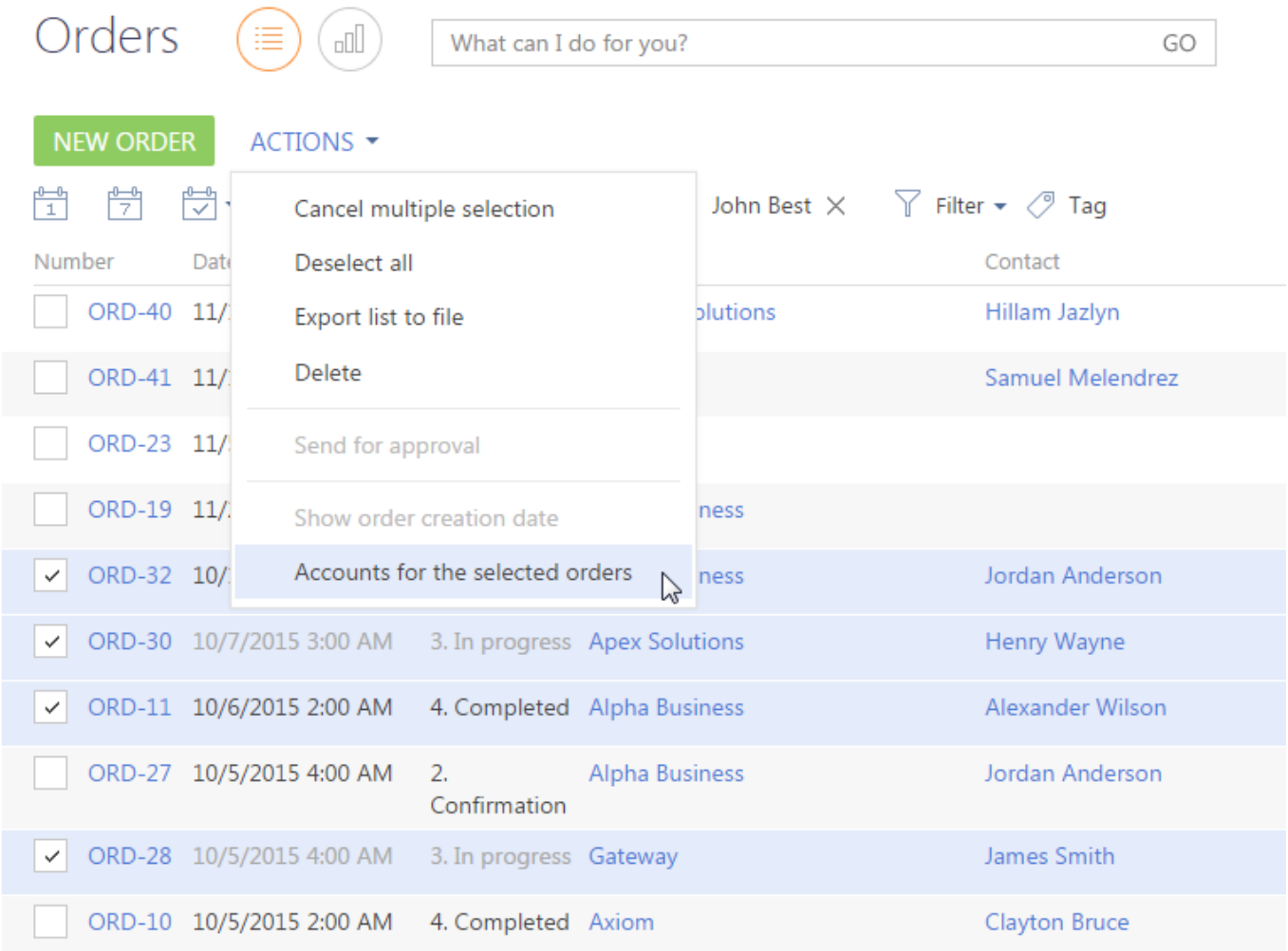
```
// Overriding of the base virtual method returning the section actions collection.
getSectionActions: function() {
    // The parent implementation of the method is called to receive
    // initiated actions collections of the base section.
    var actionMenuItems = this.callParent(arguments);
```

```
    // Adding a separator line for visual separation of the custom action from
    // the list of actions for the base section.
    actionMenuItems.addItem(this.getButtonMenuItem({
        Type: "Terrasoft.MenuSeparator",
        Caption: ""
    }));
    // Adding a menu item to the list of actions for the section.
    actionMenuItems.addItem(this.getButtonMenuItem({
        // Binding the menu item title to the localizable schema string.
        "Caption": {bindTo: "Resources.Strings.AccountsSectionAction"},
        // Binding the action handler method.
        "Click": {bindTo: "showOrdersInfo"},
        // Binding the availability property of the menu item to the value which
returns the isCustomActionEnabled method.
        "Enabled": {bindTo: "isCustomActionEnabled"}
    }));
    return actionMenuItems;
}
```

5. Save the created replacingschema

When the schema is saved and the system web-page is updated, a new action will appear in the [Orders] section. It will be active when selecting multiple records from the list (Fig. 4).

Fig. 4. – Demonstrating the case implementation result



See also:

- **Handling the selection of several records. Examples**

Handling the selection of several records. Examples

We recommend to preliminary study the case to add an action to the section in the mode of multiple records selection in the **How to add a section action: handling the selection of several records**.

Below are the examples of handling multiple records using a custom action.

Example 1

Case description

Implement the action for the [Activity] section, which sets the [Completed] status for several activities selected in the list.

Case implementation algorithm

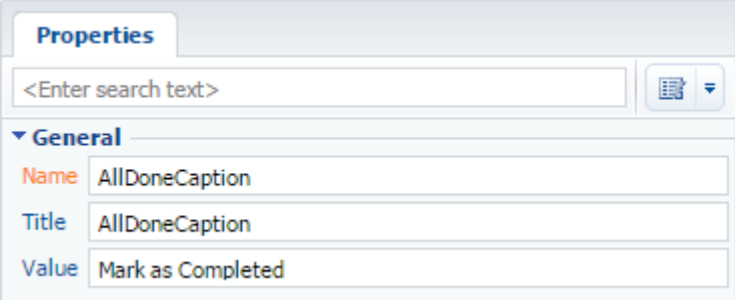
1. Create the [Activities] replacing section page in a custom package

The process of creating the replacing page is described in the **Creating a custom client module schema**.

2. Add a string with the [Actions] menu item to the collection of localizable strings in the replacingsection schema

Fill in properties for the created string as shown in Figure 1.

Fig. 1. — Properties of a custom localized string



3. Add the method implementation to the methods collection of the model view

- isCustomActionEnabled — the method that will determine the availability of the added menu item;
- setAllDone — the action handler method that will set the [Completed] status for several activities selected in the list;
- getSectionActions — the redefined virtual method wherein the handler method should be bound to an action.

Below is the code of the section replacing schema.

```
define("ActivitySectionV2", ["ConfigurationConstants"],
    function (ConfigurationConstants) {
        return {
            // Name of the section schema.
            entitySchemaName: "Activity",
            // Methods collection of the section view model.
            methods: {
                // Method ddefines whether the menu item will be available.
                isCustomActionEnabled: function () {
                    // At attempt to receive Id massive of the selected records.
                    var selectedRows = this.get("SelectedRows");
                    // If the massive contains elements (i.e. even if one record is
```

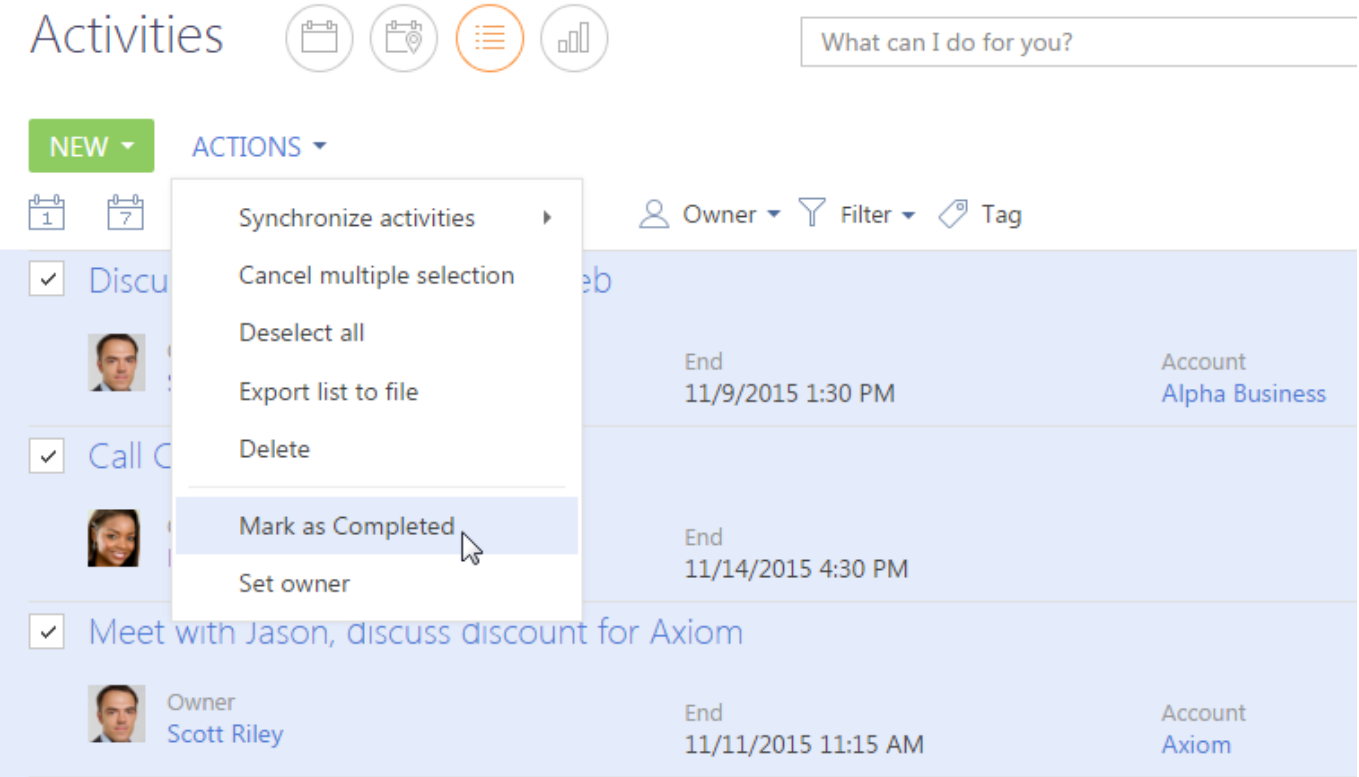
```
selected in the list),
                    // true is returned; otherwise false is present.
                    return selectedRows ? (selectedRows.length > 0) : false;
                },
                // Action handler method setting the "Completed" status for selected
records.
                setAllDone: function () {
                    // Receiving the massive of values in primary columns of selected
records.
                    var selectedRows = this.get("SelectedRows");
                    // Processing is started only if even one record is selected.
                    if (selectedRows.length > 0) {
                        // Creating an instance of the package request class.
                        var batchQuery = this.Ext.create("Terrasoft.BatchQuery");
                        // Updating each of the selected records.
                        selectedRows.forEach(function (selectedRowId) {
                            // Creating an instance of the UpdateQuery class with the
Activity core schema.
                            var update = this.Ext.create("Terrasoft.UpdateQuery", {
                                rootSchemaName: "Activity"
                            });
                            // Using the filter to determine a record to be updated.
                            update.enablePrimaryColumnFilter(selectedRowId);
                            // Set the "Completed" value for the Status column using
the
                            // ConfigurationConstants.Activity.Status.Done
configuration constant.
                            update.setParameterValue("Status",
                                ConfigurationConstants.Activity.Status.Done,
this.Terrasoft.DataValueType.GUID);
                            // Adding a request for a record update to a package
request.
                            batchQuery.add(update);
                        }, this);
                        // Fulfilling the package request to the server.
                        batchQuery.execute(function () {
                            // Updating the list.
                            this.reloadGridData();
                        }, this);
                    }
                },

                getSectionActions: function () {
                    // The parent implementation of the method is called to receive
                    // the initiated actions collections for the base section.
                    var actionMenuItems = this.callParent(arguments);
                    // Adding a separators line to visually separate the user action
from the list
                    // of actions in the base section.
                    actionMenuItems.addItem(this.getButtonMenuItem({
                        Type: "Terrasoft.MenuSeparator",
                        Caption: ""
                    }));
                    // Adding a menu item to the section actions list.
                    actionMenuItems.addItem(this.getButtonMenuItem({
                        // Binding the menu item title to the localizable schema
string.
                        "Caption": { bindTo: "Resources.Strings.AllDoneCaption" },
                        // Binding the action handler method.
                        "Click": { bindTo: "setAllDone" },
                        // Binding the availability property of the menu item to the
value which returns the isCustomActionEnabled method.
```

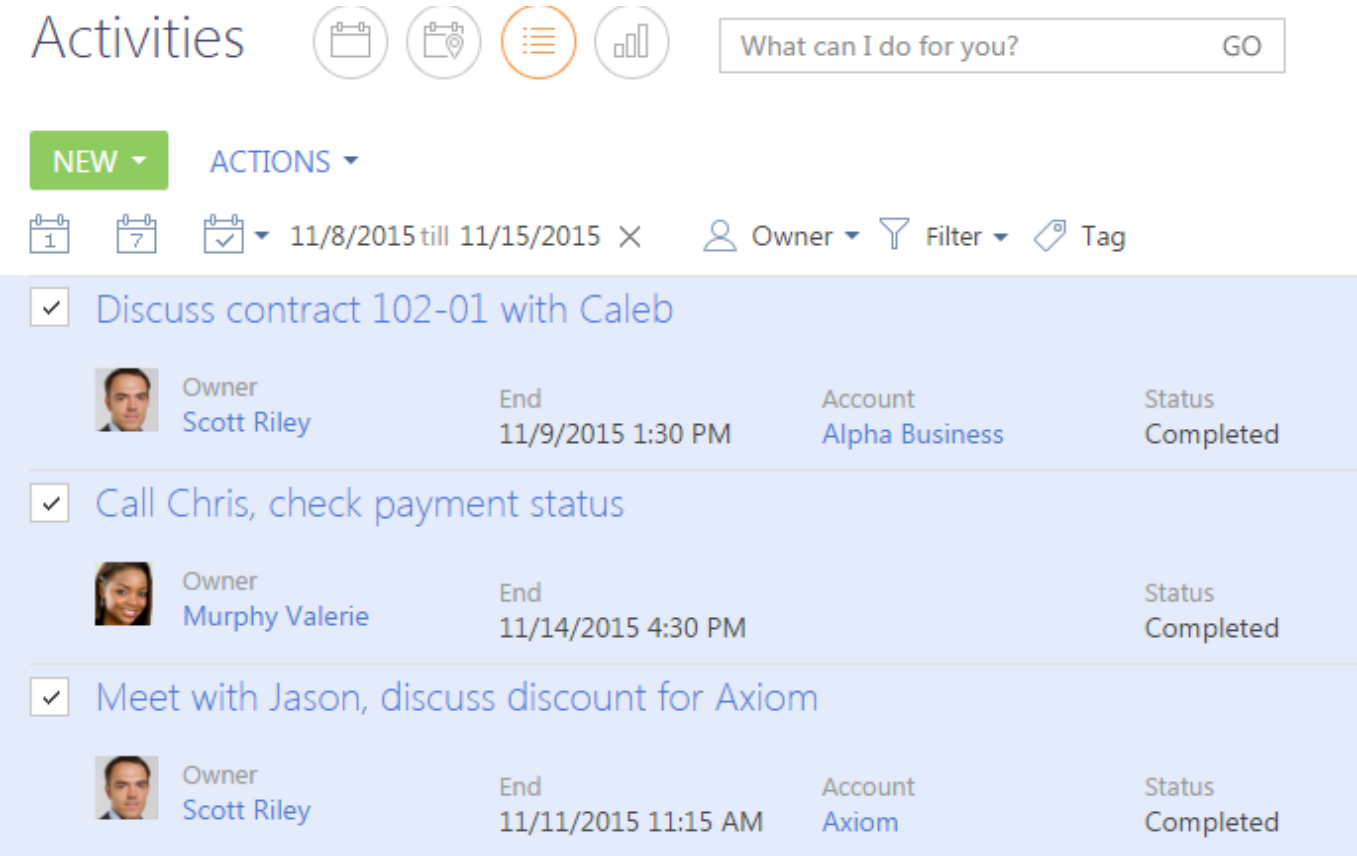
```
                        "Enabled": { bindTo: "isCustomActionEnabled" }
                    }));
                    return actionMenuItems;
                }
            }
        };
    });
```

4. Save the created replacing schema

When the schema is saved and the system web-page is updated, the [Completed] status may be used for several selected activities at once in the [Activities] section using the new [Mark as "Completed"] action (Fig. 2).

Fig. 2. – Result demonstration





Example 2

Case description

Implement the action for the [Activities] section, which calls the owner selection window and set the selected value for several activities selected in the list.

Case implementation algorithm

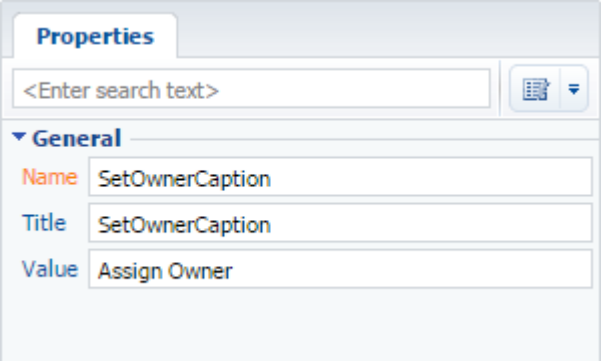
1. Create the [Activities] replacingsection page in a custom package

The process of creating a replacing page is described in the article **Creating a custom client module schema**.

2. Add a string with the menu item titled [Actions] to the collection of localizable strings in the replacingsection schema

Fill in properties for the created string as shown in Figure 3.

Fig. 3. — Properties of a custom localizable string



3. Add the method implementation to the methods collection of the view model

- isCustomActionEnabled — the method determining the accessibility of an added menu item;
- setOwner — the action handler method calling the opening of the [Contacts] lookup;
- lookupCallback — the callback method setting the contact selected in the lookup as the owner for records selected in the list;
- getSectionActions — the redefined virtual method wherein the handler method must be bound to an action.

Below is the code of the section replacing schema.

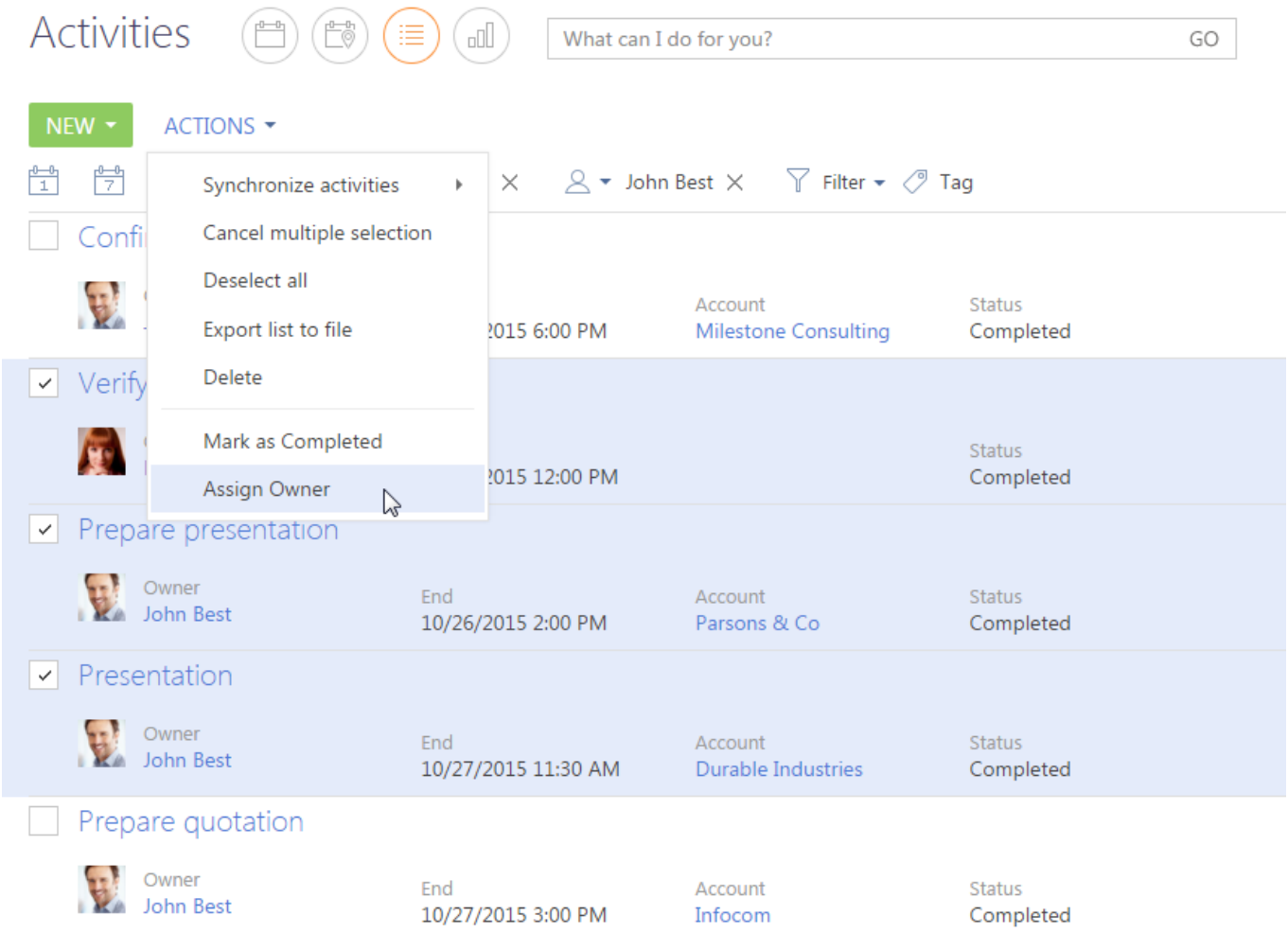
```
define("ActivitySectionV2", ["ConfigurationConstants"],
    function (ConfigurationConstants) {
        return {
            // Name of the section schema.
            entitySchemaName: "Activity",
            // Methods collection of the section view model.
            methods: {
                // The method determines whether the menu item will be available.
                isCustomActionEnabled: function () {
                    // At attempt to receive an array of selected record identifiers
is made.
                    var selectedRows = this.get("SelectedRows");
                    // If the array contains elements (i.e. even if one record is
selected in the list),
                    // true is returned; otherwise false is present.
                    return selectedRows ? (selectedRows.length > 0) : false;
                },
                // Action handler method calling the [Contacts] lookup opening.
                setOwner: function () {
                    // Determining the configuration of the opened lookup: [Contact]
schema; multiple
                    // selection is disabled; the displayed column is  [Name].
                    var config = {
                        entitySchemaName: "Contact",
                        multiSelect: false,
                        columns: ["Name"]
                    };
                    // Opening the lookup with a certain configuration and callback
function which will actuate
                    // when the [Select] button is clicked.
                    this.openLookup(config, this.lookupCallback, this);
                },
                // Method setting the contact selected in the lookup as the owner
                // for records selected in the list.
                lookupCallback: function (args) {
                    var activeRowId;
                    var lookupSelectedRows = args.selectedRows.getItems();
                    if (lookupSelectedRows && lookupSelectedRows.length > 0) {
                        // Receiving the Id of the record selected in the lookup.
                        activeRowId = lookupSelectedRows[0].Id;
                    }
                    // Receiving the array of values in primary columns of selected
records.
                    var selectedRows = this.get("SelectedRows");
                    // Processing starts only if even one record is selected in the
list and the owner is selected
                    // in the lookup.
                    if ((selectedRows.length > 0) && activeRowId) {
                        // Creating an instance of the package request class.
                        var batchQuery = this.Ext.create("Terrasoft.BatchQuery");
                        // Updating each of the selected records.
```

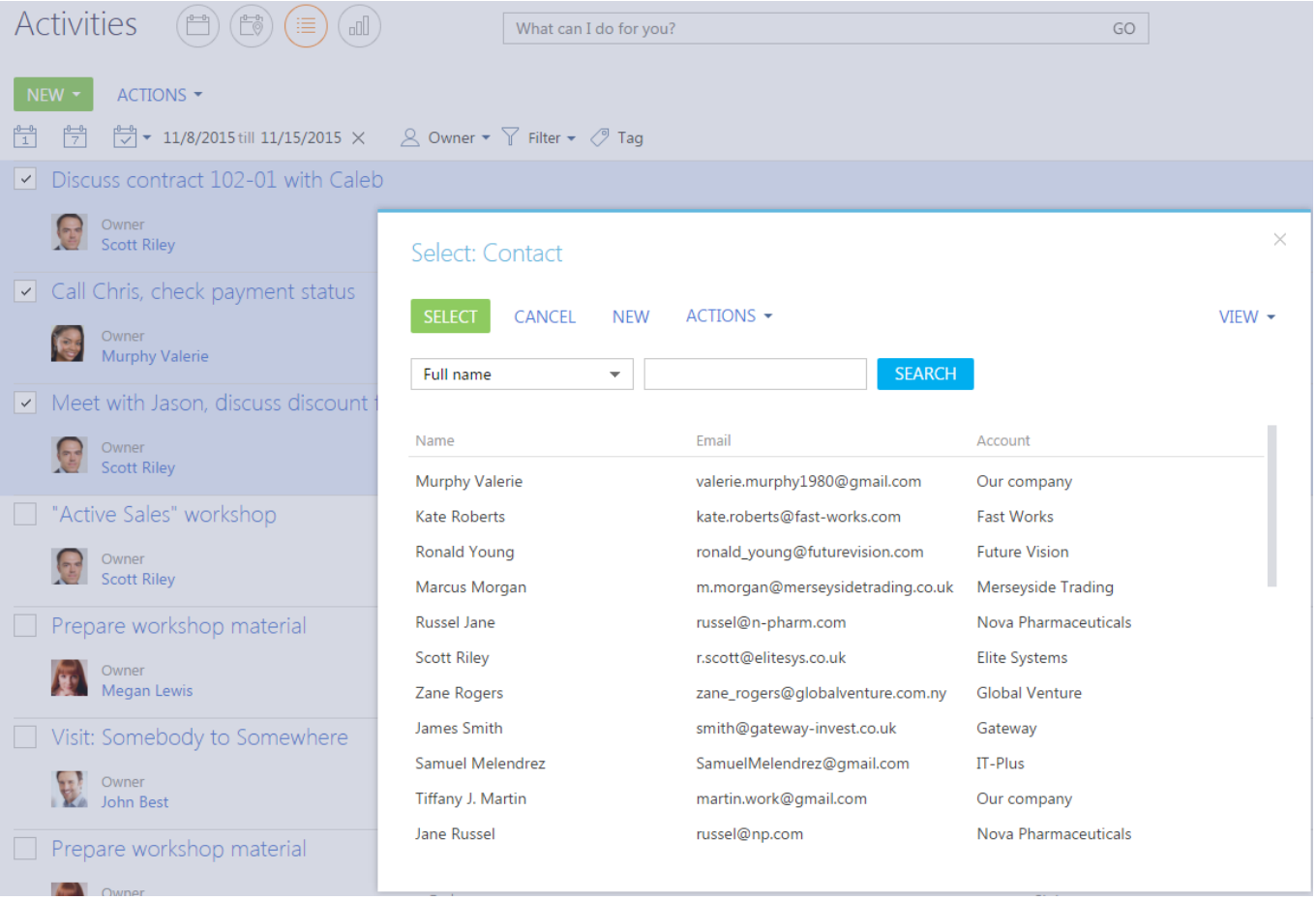
```
                        selectedRows.forEach(function(selectedRowId) {
                            // Creating an instance of the UpdateQuery class with the
Activity core schema.
                            var update = this.Ext.create("Terrasoft.UpdateQuery", {
                                rootSchemaName: "Activity"
                            });
                            // Using the filter to determine a record to be updated.
                            update.enablePrimaryColumnFilter(selectedRowId);
                            // Set the value for the [Owner] column, which is equal
to the contact selected in the lookup.
                            update.setParameterValue("Owner", activeRowId,
this.Terrasoft.DataValueType.GUID);
                            // Adding a request for a record update to a package
request.
                            batchQuery.add(update);
                        }, this);
                        // Fulfilling the package request to the server.
                        batchQuery.execute(function() {
                            // Updating the list.
                            this.reloadGridData();
                        }, this);
                    }
                },
                // Overriding  the base virtual method returning the section actions
collection.
                getSectionActions: function () {
                    // The parent implementation of the method is called to receive
                    // the initiated actions collections for the base section.
                    var actionMenuItems = this.callParent(arguments);
                    // Adding a separator line for visual separation of the custom
action
                    // from the list of actions for the base section.
                    actionMenuItems.addItem(this.getButtonMenuItem({
                        Type: "Terrasoft.MenuSeparator",
                        Caption: ""
                    }));
                    // Adding a menu item to the section actions list.
                    actionMenuItems.addItem(this.getButtonMenuItem({
                        // Binding the menu item title to the localizable schema
string.
                        "Caption": { bindTo: "Resources.Strings.SetOwnerCaption" },
                        // Binding the action handler method.
                        "Click": { bindTo: "setOwner" },
                        // Binding the availability property of the menu item to the
value which returns the isCustomActionEnabled method.
                        "Enabled": { bindTo: "isCustomActionEnabled" }
                    }));
                    return actionMenuItems;
                }
            }
        };
    });
```

4. Save the created replacing schema

When the schema is saved and the system web-page is updated, the "Owner" field of several selected activities may be modified in the [Activities] section using the new [Assign Owner] action (Fig. 4).

Fig. 4. – Result demonstration


Activities
What can I do for you?
GO
NEW
ACTIONS
Synchronize activities
Cancel multiple selection
Deselect all
Export list to file
Delete
Mark as Completed
Assign Owner
John Best
Filter
Tag
Account
Milestone Consulting
Status
Completed
Verify
Status
Completed
Prepare presentation
Owner
John Best
End
10/26/2015 2:00 PM
Account
Parsons & Co
Status
Completed
Presentation
Owner
John Best
End
10/27/2015 11:30 AM
Account
Durable Industries
Status
Completed
Prepare quotation
Owner
John Best
End
10/27/2015 3:00 PM
Account
Infocom
Status
Completed



How to add a button to a section

Example of implementing a button adding to a section

Case description

The button for opening the base contact edit page in the account selected in the list must be added to the [Accounts] page of the section.

> NOTE
>
> A selected record is addressed via the ***ActiveRow*** attribute of the section view model. The attribute returns the value in the primary column of the selected record. Then, this value may be used to receive values in other fields of the selected object, for example, from the list registry data collection stored in the ***GridData*** property of the list view model.

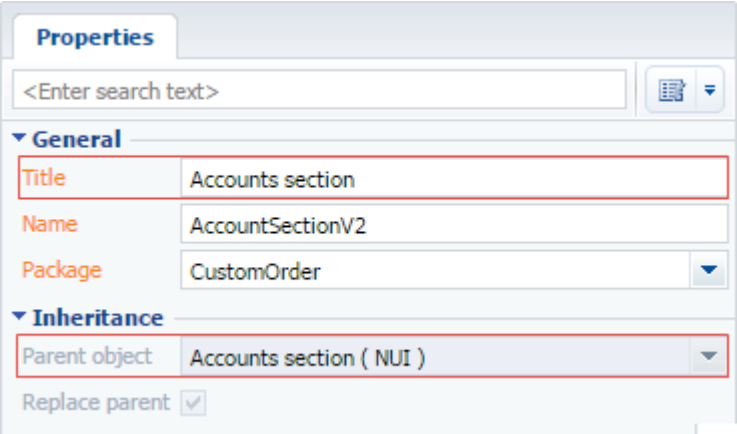
Case implementation algorithm

1. Create a replacingedit page for the [Accounts] section

A replacingclient 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**.

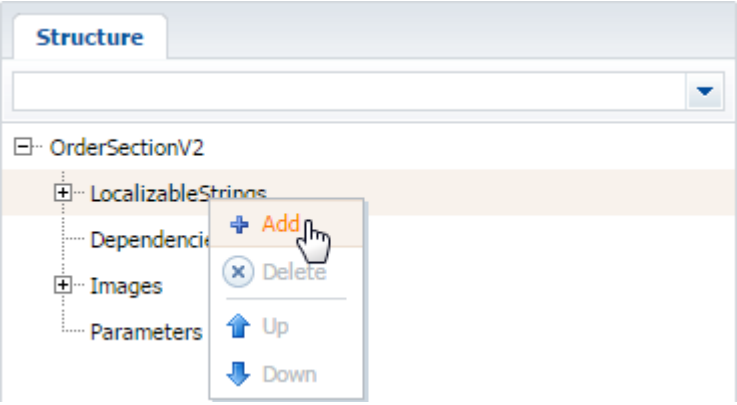
Fig. 1. — Properties of the replacing edit page



2. Add a string with the button title to the localizable strings collection of the replacingsection schema

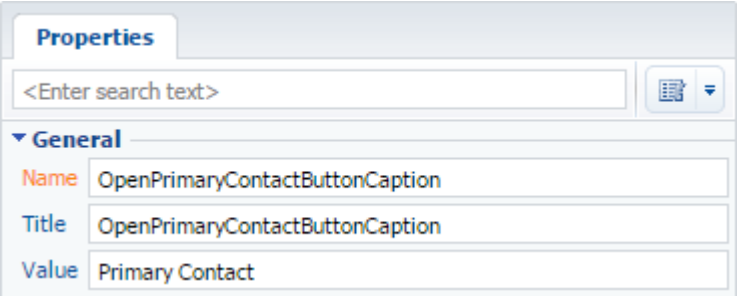
For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure node (Fig. 2).

Fig. 2. — Adding a localizable string to the schema



Fill in the properties for the created string 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 [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 in it to the methods collection of the view model.

4. Add a configuration object with settings of the button location on the page to the diff array

Below is the source code of the section replacing schema.

```
define("AccountSectionV2", [],
```

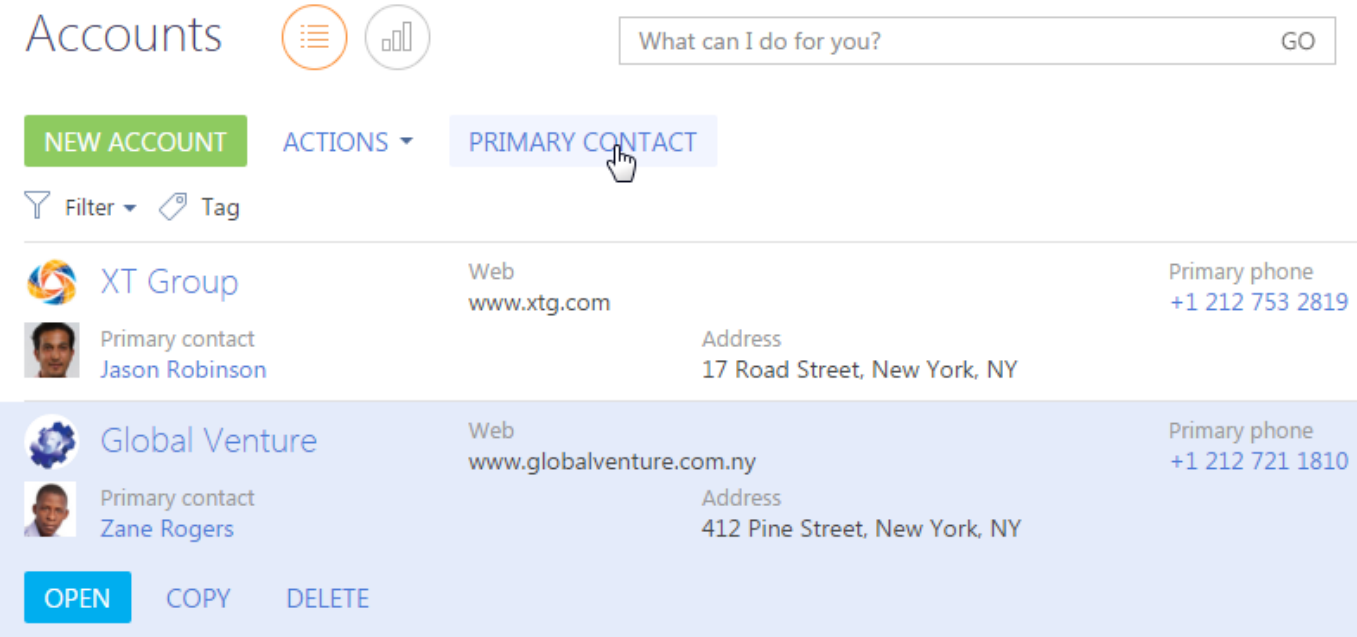
```
    function () {
        return {
            // Name of the section object schema.
            entitySchemaName: "Account",
            // Methods collection of the section view model.
            methods: {
                // Button press handler method.
                onOpenPrimaryContactClick: 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 of the selected
item is filled.
                isAccountPrimaryContactSet: function () {
                    var activeRow = this.get("ActiveRow");
                    if (activeRow) {
                        var pc =
this.get("GridData").get(activeRow).get("PrimaryContact");
                        return (pc || pc !== "") ? true : false;
                    }
                    return false;
                }
            },
            //Setting the button visualization in the section.
            diff: /**SCHEMA_DIFF*/[
                // Metadata for adding a custom button to the page.
                {
                    // Indicates that an operation of adding an item to the page is
being executed.
                    "operation": "insert",
                    // Metadata of the parent control item where the button is added.
                    "parentName": "ActionButtonsContainer",
                    // 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": "MainContactSectionButton",
                    // 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
                        }
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
    });
```

5. Save the created replacingsection schema

When the schema is saved and the system web-page is updated, the [Base Contact] button will appear in the [Accounts] section. The button will be activated after selecting the account with the specified Base Contact (Fig. 4).

Fig. 4. – Demonstrating the case implementation result



How to highlight a record in the list in color

Bpm'online has the possibility to set up the list display so that certain records are highlighted in it. This means that a list record can be highlighted in a certain color when certain conditions are met. This setting helps to highlight records requiring special attention.

The ***customStyle*** property of the list record is responsible for the list string display.

The customStyle property is an object whose properties form the style of displaying the list record. CSS-properties serve as properties, for example:

```
item.customStyle = {
    // Font color.
    "color": "darkgrey",
    // Background color.
    "background": "#D8FBC2"
};
```

To set up the display of separate list records, the following sequence of actions:

1. Overload the ***prepareResponseCollectionItem(item)*** base method that modifies the data string before uploading to the list..
2. Assign a certain value to the customStyle property in the ***prepareResponseCollectionItem(item)*** method for required list records.

Below is the example of highlighting the list records in color.

Case description

Highlight the list records at the [In progress] stage for the [Orders] section.

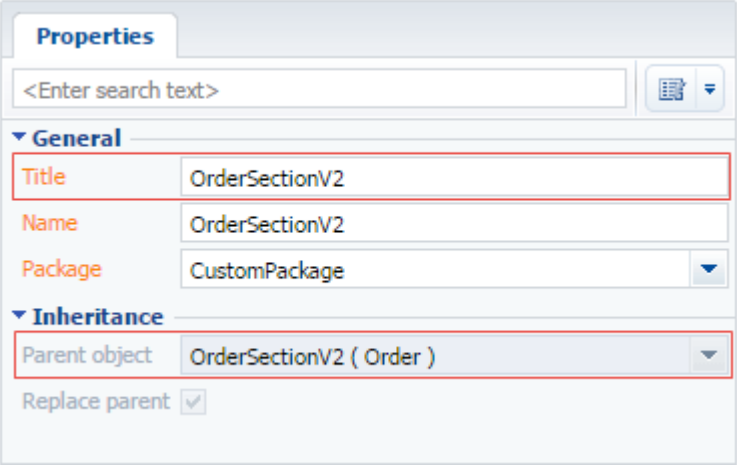
Case implementation algorithm

1. Create the [Orders] replacing section page in a custom package

A replacing client module must be created and [OrderSectionV2] must be specified as the parent object in it (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 [Orders] section replacing page



2. Add the isRunning method implementation into the methods collection of the section view model

- prepareResponseCollectionItem — the overloaded base method modifying the data string before uploading to the list; this will also add set styles to certain list strings.

For this purpose, add the program code of the section replacing module to the source code tab. Required methods are added in it to the methods collection of the view model.

Below is the source code of the section schema:

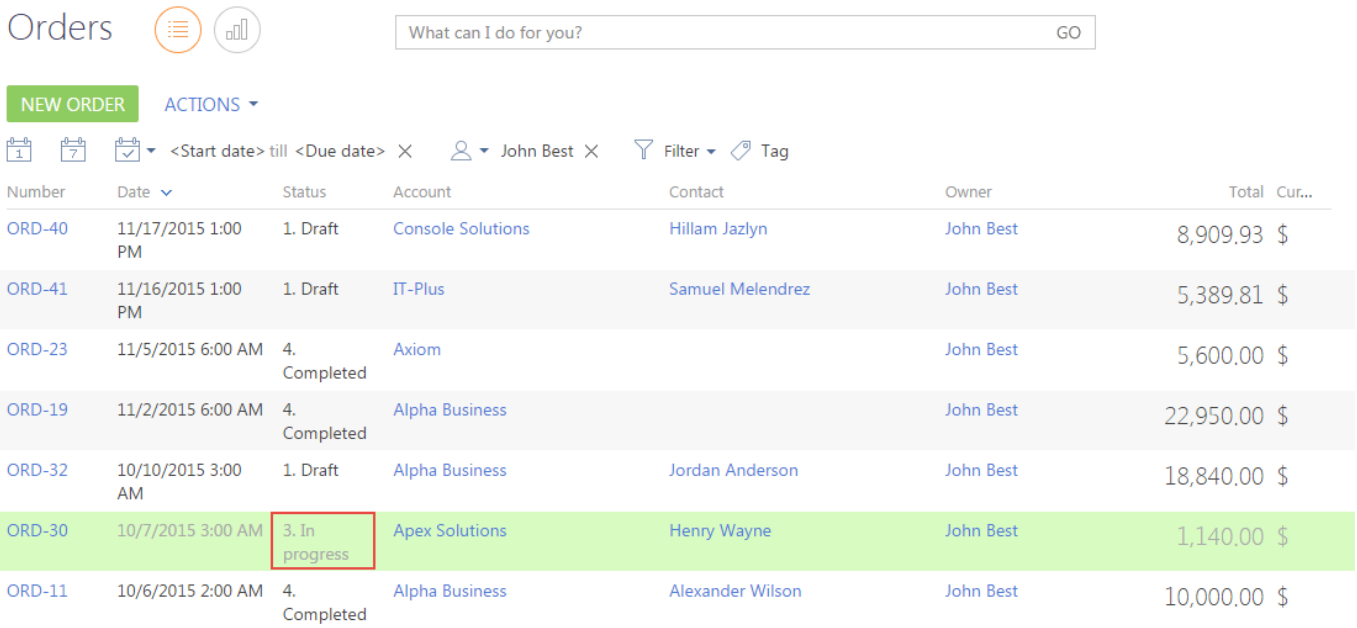
```
define("OrderSectionV2", ["OrderConfigurationConstants"],
    function (OrderConfigurationConstants) {
        return {
            // Name of the section schema.
            entitySchemaName: "Order",
            // Methods collection of the section view model.
            methods: {
                // Overloading the base method that modifies the data string before
uploading it to the list.
                prepareResponseCollectionItem: function (item) {
                    this.callParent(arguments);
                    item.customStyle = null;
                    var running = item.get("Status");
```

```
                    // If this condition is met, the record color changes to grey and
the background color changes to light green.
                    if (running.value ===
OrderConfigurationConstants.Order.OrderStatus.Running) {
                        item.customStyle = {
                            'color': "darkgrey",
                            'background': "#D8FBC2"
                        }
                    }
                }
            }
        };
    });
```

3. Save the created replacing schema

When the schema is saved and the system web page is updated, orders in the [Orders] section, which have the [In progress] status will be highlighted in bright green in the list (Fig. 2).

Fig. 2. – Demonstrating the case implementation result



How to add a fixed filters block to the section

Introduction

Filters are designed to search and filter records in sections. In the bpm'online provided quick, standard and advanced filters and folders. 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 following properties:

- *entitySchema* – object schema.
- *filters* – array of added filters.

Assign a reference to the created configuration object to the [fixedFiltersConfig] field of the view model:

```
this.set("FixedFilterConfig", fixedFilterConfig);
```

Case description

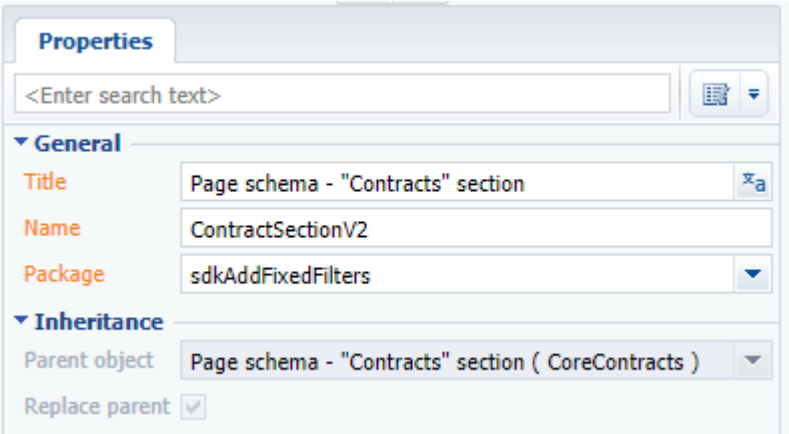
Add a block of quick filters to the [Contracts] section. Filter by the contract start date and owner.

Case implementation algorithm

1. Create a replacing schema of the [Contracts] section in the custom package.

Create a replacing client module and specify the *ContractSectionV2* schema as parent (Fig. 1). The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**" article.

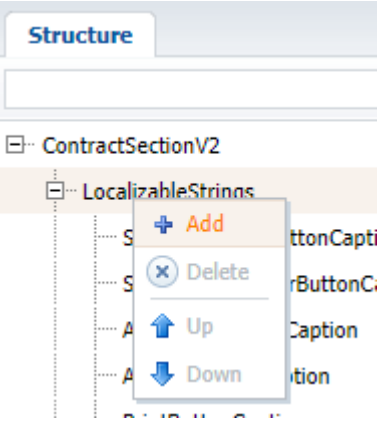
Fig. 1. Properties of the [Contracts] section replacing schema



2. Add localizable strings to the schema structure.

To do this, right-click the [LocalizableStrings] structure node and select [Add] from the context menu (Fig. 2).

Fig. 2. Adding localizable string to the schema



Create two localizable strings with following properties:

Name	Value
OwnerFilterCaption	Owner
PeriodFilterCaption	Period

3. Add the implementation of the initFixedFiltersConfig() method to the method collection of the section view model.

Create configuration object with the *PeriodFilter* and *OwnerFilter* filter arrays in the *initFixedFiltersConfig()* method, assign a reference to the created configuration object to the *fixedFiltersConfig* field of the view model.

The source code of the section view model schema:

```
define("ContractSectionV2", ["BaseFiltersGenerateModule"],
```

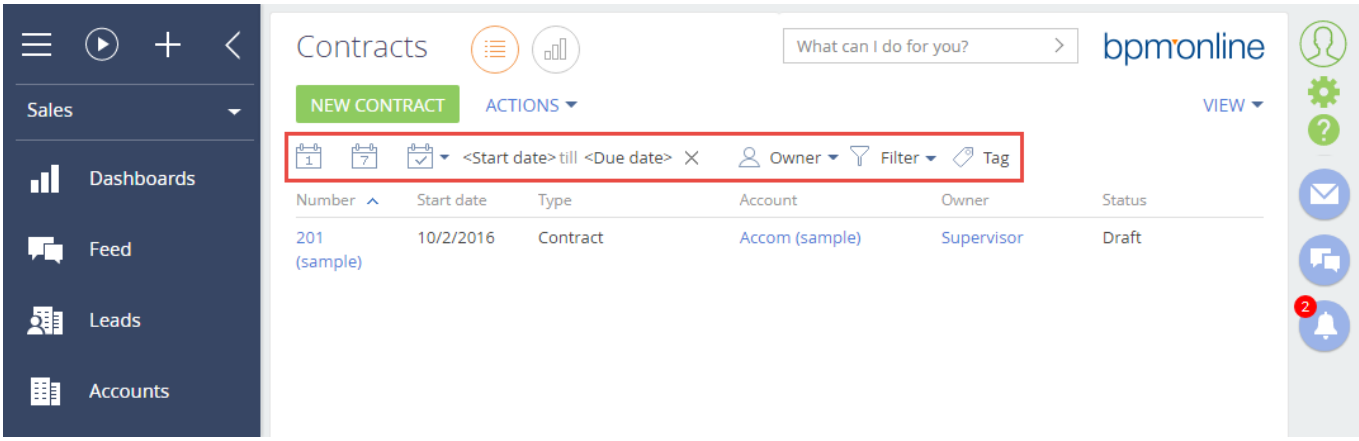
```
function(BaseFiltersGenerateModule) {
    return {
        // Name of the section schema
        entitySchemaName: "Contract",
        // Method collection of the section view model.
        methods: {
            // Initializes the fixed filters.
            initFixedFiltersConfig: function() {
                // 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);
            }
        }
    };
```

```
});
```

4. Save the created replacing schema

After saving the schema and restarting the system, a block of fixed filters will appear in the [Contracts] section. These filters will enable to filter contracts by start date and owner (Fig. 3).

Fig. 3. Case result



Page configuration

- **Creating a custom client module schema**
- **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 dashboard**
- **How to add a computed field**
- **How to set a default value for a field**
- **How to add the field validation**
- **Using filtration for lookup fields. Examples**
- **Displaying contact's time zone**

Setting the edit page fields using business rules

One of the tools for setting the page logic in bpm'online is business rules (or rules).

Rules are a standard mechanism of the system that enable the designer to add the typical functionality implementation by configuring the view model columns.

This means that there is no need to write additional user methods for solving typical tasks. It is enough to configure columns using the rules.

Rule functionality is realized in the BusinessRuleModel client module. To use this functionality, the BusinessRuleModule must be added to a list of user module dependencies.

Example 1. — Adding the BusinessRuleModule module to a list of dependencies

```
define("CustomPageModule", ["BusinessRuleModule"],
    function(BusinessRuleModule) {
        return {
```

```
            // Client module implementation.
        };
    });
```

General procedure for declaring rules:

- All rules are described in the rules property of the page view model.
- The rules are applied to the view model columns, but not to the management items.
- The rule has a name.
- The rule parameters are set in its configuration object.

Example 2. — Procedure for declaring rules

```
// List of all rules of the view model.
rules : {
    // Column name, for which the rule is added.
    «FirstColumnName» : {
        // The list of rules for the FirstColumnName column.
        // Rule name.
        FirstRuleName : {
            // Configuration item of the FirstRuleName rule.
            ruleType: <a value from the BusinessRuleModule.enums.RuleType>
            // Other configuration properties of the rule.
        },
        SecondRuleName: {
            // Configuration item of the SecondRuleName rule.
            ruleType: <a value from the BusinessRuleModule.enums.RuleType>
            // Other configuration properties of the rule.
        }
    },
    «SecondColumnName» : {
        // The list of rules for the SecondColumnName column.
    }
}
```

The types of bpm'online rules are listed in RuleType of the BusinessRuleModue module.

There are 3 types of rules defined:

1. BINDPARAMETER

The BINDPARAMETER is used to bind the properties one column to values of various parameters. For example, to set the column visibility or accessibility depending on the other column value.

Configuring the BINDPARAMETER rule

Property	Value
ruleType	Type of the rule. The ruleType is set using a value from the BusinessRuleModule.enums.RuleType list. The BINDPARAMETER type is used here.
property	Property of a management item. The property is set using a value from the BusinessRuleModule.enums.Property list: • VISIBLE —column visibility; • ENABLED —column accessibility; • REQUIRED — filling required; • READONLY — the read only column.
conditions	Array of conditions for a rule application.

Property	Value
	Each condition is a configuration object described below.
logical	Logical operator for combining conditions from the "conditions" property. The logical property is set using a value from the Terrasoft.LogicalOperatorType list.

Configuring a condition of the BINDPARAMETER rule

Property	Value

leftExpression	Expression of the left part of a condition. The leftExpression is a configuration object with the following properties:
	• type — a type of expression. The type is set using a value from the BusinessRuleModule.enums.ValueType list:
	  • CONSTANT — the constant;
	  • ATTRIBUTE — a value in the view model column;
	  • SYSSETTING — a system setting;
	  • SYSVALUE — a system value. Item of the system value list Terrasoft.core.enums.SystemValueType.
	• attribute – the name of the model column;
	• attributePath – the meta-path to the lookup schema column;
	• value – a value for comparison.

comparisonType	Type of comparison. The comparisonType is set using a value from the Terrasoft.core.enums.ComparisonType list.
rightExpression	Expression of the right part of the condition (like as leftExpression).

2. FILTRATION

Filtration of the view model column values is used to find a specific lookup type column depending on a value of the current page status, for example.

Configuring the FILTRATION rule

Property	Value
ruleType	Type of the rule. The ruleType is set using a value from the BusinessRuleModule.enums.RuleType list. The FILTRATION type is used here.
autocomplete	Inverse filtration checkbox. They checkbox may have the true or false values.
baseAttributePatch	Meta-path to the lookup schema column, which will be used for filtration. The feedback principle is used for building the path to a column (as in EntitySchemaQuery). The path is formed in relation to the schema referenced in the model column.
comparisonType	Type of a comparison operation. It is set using a value from the Terrasoft.ComparisonType list.
type	Type of the value to be compared with baseAttributePatch. It is set using a value from the BusinessRuleModule.enums.ValueType list.
attribute	Name of the view model column.

Property	Value
	This property is described if the ATTRIBUTE value type (type) is specified.
attributePath	Meta-path to an object schema column. The feedback principle is used for building the path to a column (as in EntitySchemaQuery). The path is formed in relation to the schema referenced in the model column.
value	Value to be filtered. This property is described if any value type (type), except ATTRIBUTE is specified.

Examples of adding rules to the edit page

- **Adding the [FILTRATION] rule to the edit page**
- **How to block the edit page field by a specific condition**
- **How to hide the edit page field by a specific condition**
- **How to make the field required by a specific condition**
- **Business rules created in wizards**

Adding the [FILTRATION] rule to the edit page

Example of adding the [FILTRATION] rule to the edit page

Case description

Add the [State/province] and [City] field to the contact page. When selecting a specific region, a list of cities must be filtered by the selected region. When selecting a specific city, the region field must be filled with the corresponding region.

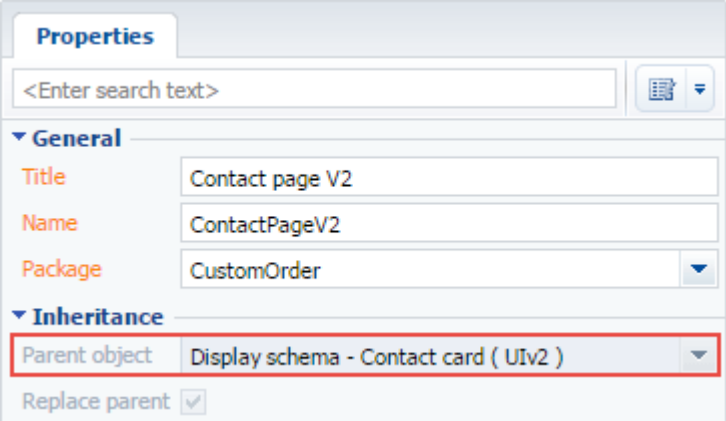
Case implementation algorithm

1. Create a replacement contact edit page

A replacement client module must be created; [Display schema – Contact card] (Fig. 1) must be specified as the parent object.

The process of creating a replacement page is described in the article **Creating a custom client module schema**.

Fig. 1. — Properties of the replacement edit page



2. Add a configuration object with settings of the page field location to the diff array

Locate the [City] and [State/province] fields on the contact edit page.

For this purpose, add the program code of the replacement module of the page to the source code tab.

```
// Add the BusinessRuleModule module to the list of module rules.
define("ContactPageV2", ["BusinessRuleModule", "ConfigurationConstants"],
    function(BusinessRuleModule, ConfigurationConstants) {
        return {
            // Schema name of the edit page object.
            entitySchemaName: "Contact",
            // Visualization setup for the [City] and [State/provicne] fields on the
edit page.
            diff: [
                // Metadata for adding the [City] field.
                {
                    "operation": "insert",
                    "parentName": "ContactGeneralInfoBlock",
                    "propertyName": "items",
                    "name": "City",
                    "values": {
                        "layout": {"column": 13, "row": 2, "colSpan": 12}
                    }
                },
                // Metadata for adding the [State/province] field.
                {
                    "operation": "insert",
                    "parentName": "ContactGeneralInfoBlock",
                    "propertyName": "items",
                    "name": "Region",
                    "values": {
                        "layout": {"column": 0, "row": 2, "colSpan": 12}
                    }
                }
            ]
        };
    });
```

3. Add the rule with the FILTRATION type for the [City] column to the rules property of the page view model

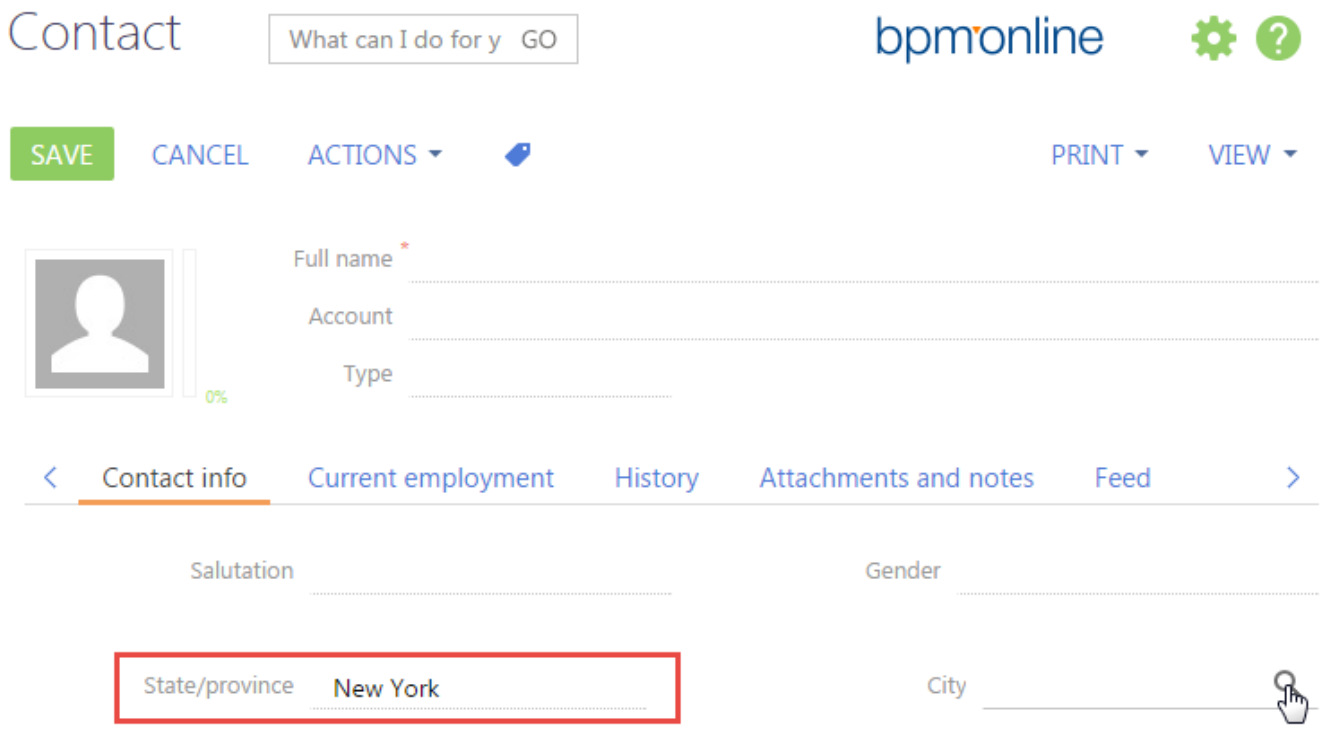
```
define("ContactPageV2", ["BusinessRuleModule", "ConfigurationConstants"],
    function(BusinessRuleModule, ConfigurationConstants) {
        return {
            entitySchemaName: "Contact",
                // Visualization setup for the [City] and [State/provicne] fields on
the edit page.
                diff: [
                // Metadata for adding the [City] field.
                // Metadata for adding the [State/provicne] field.
            ],
            // Rule object for the edit page view model.
            rules: {
                // The set of rules for the [City] column of the view model.
                "City": {
                    // Rules for filtering the [City] column by the [Region] column.
                    FiltrationCityByRegion: {
                        /// FILTRATION rule type.
                        ruleType: BusinessRuleModule.enums.RuleType.FILTRATION,
                        // Reverse filtering will be performed.
                        autocomplete: true,
                        autoClean: true,
                        // Path to the column in the [City] reference shema,
```

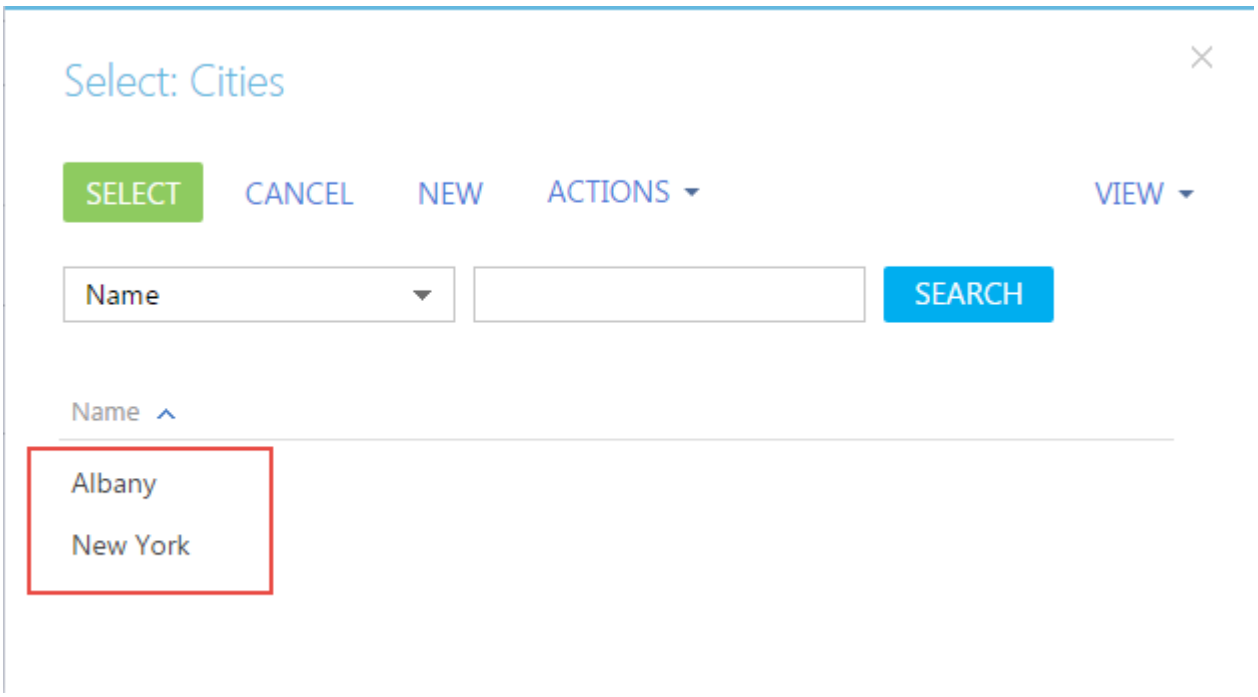
```
                        // which the [City] column of the page view model is
referencing.
                        baseAttributePatch: "Region",
                        // Type of the comparison operation in filter.
                        comparisonType: Terrasoft.ComparisonType.EQUAL,
                        // The view model column (attribute) is used as the value
                        // during the comparison.
                        type: BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                        // Name of the page view model column,
                        // by which the filtering will be performed.
                        attribute: "Region"
                    }
                }
            }
        };
    });
```

4. Save the created replacement page schema

When the schema is saved and the web-page of the system is updated on the contact edit page, two new fields [State/province] and [City] will appear. The content will be filtered in each of them depending on the value selected in another field (Fig. 2).

Fig. 2. – Demonstrating the case implementation result





How to hide the edit page field by a specific condition

Case description

Add a new [Meeting Place] field to the activity edit page. The field will be displayed only for [Visit] type activities.

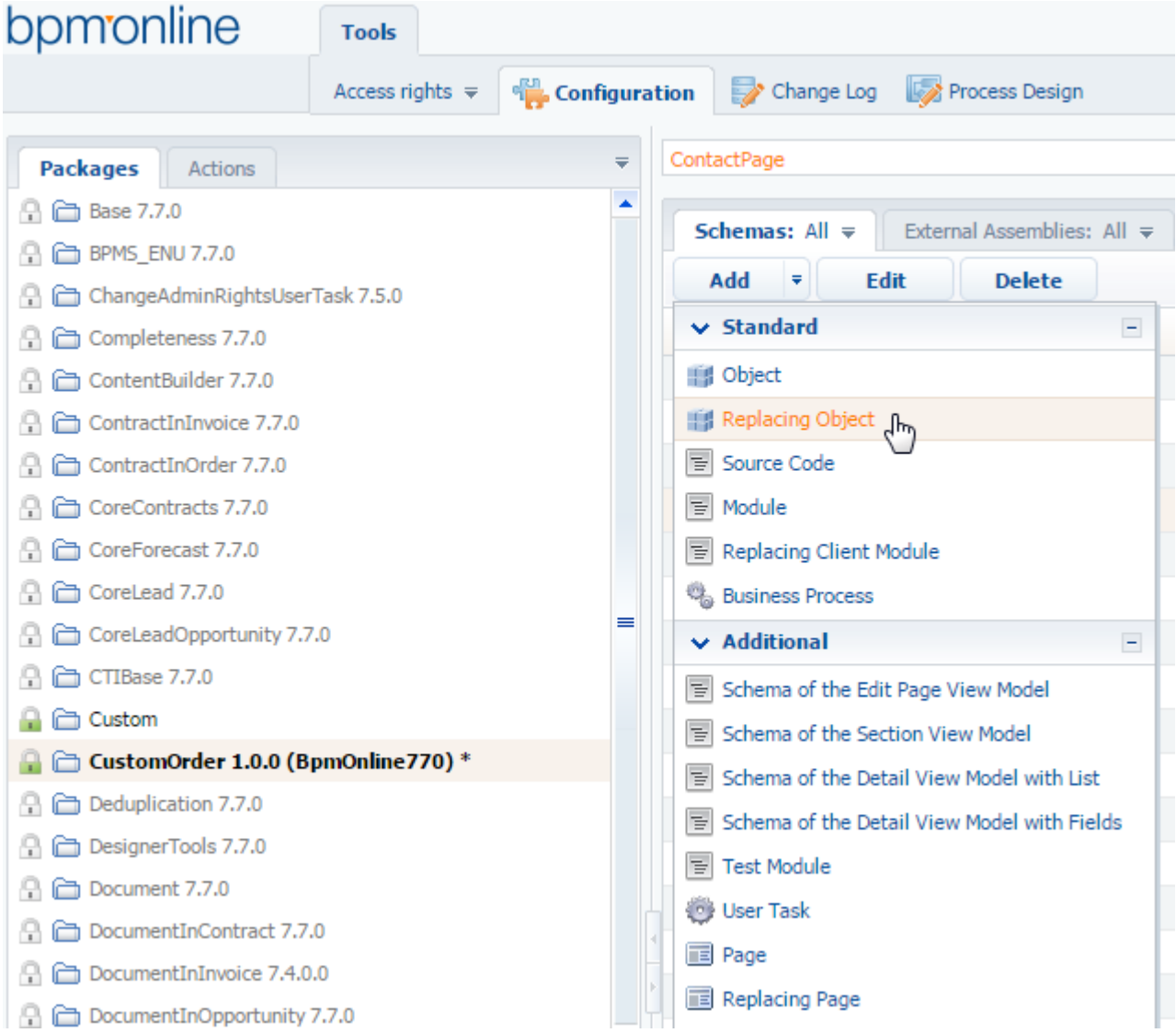
> NOTE
>
> A field to the edit page may be added by two ways – using the Section Wizard and manually.
>
> The process of adding a field to the edit page is described in the article **Adding a new field to the edit page**.

Case implementation algorithm

1. Create a [Activity] replacement object and add a new [Meeting Place] column

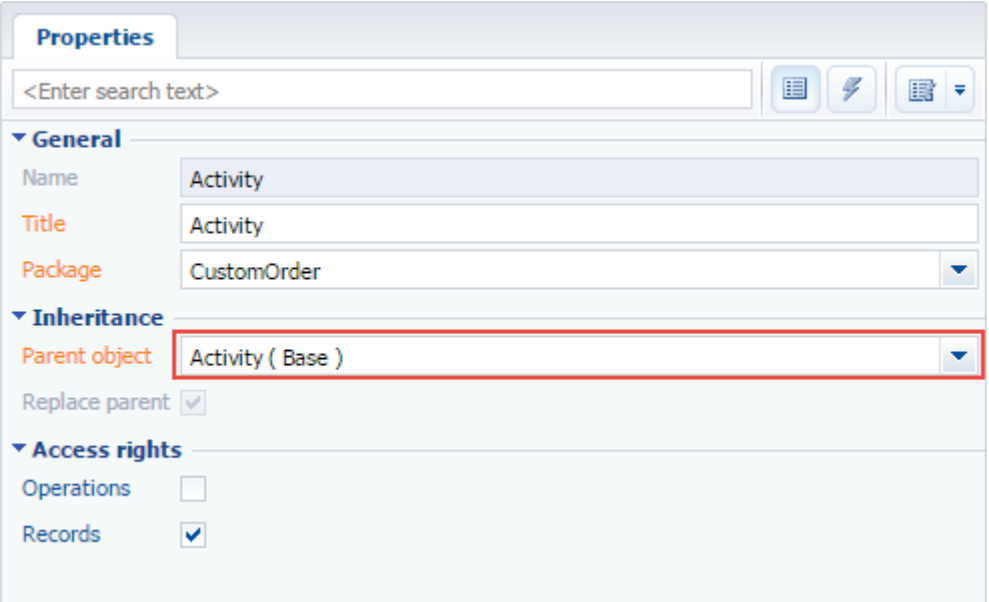
For this purpose, select a user package and run the [Add] > [Replacing Object] menu item (Fig. 1) on the [Schemas] tab.

Fig. 1. – Creating the replacement schema of the object



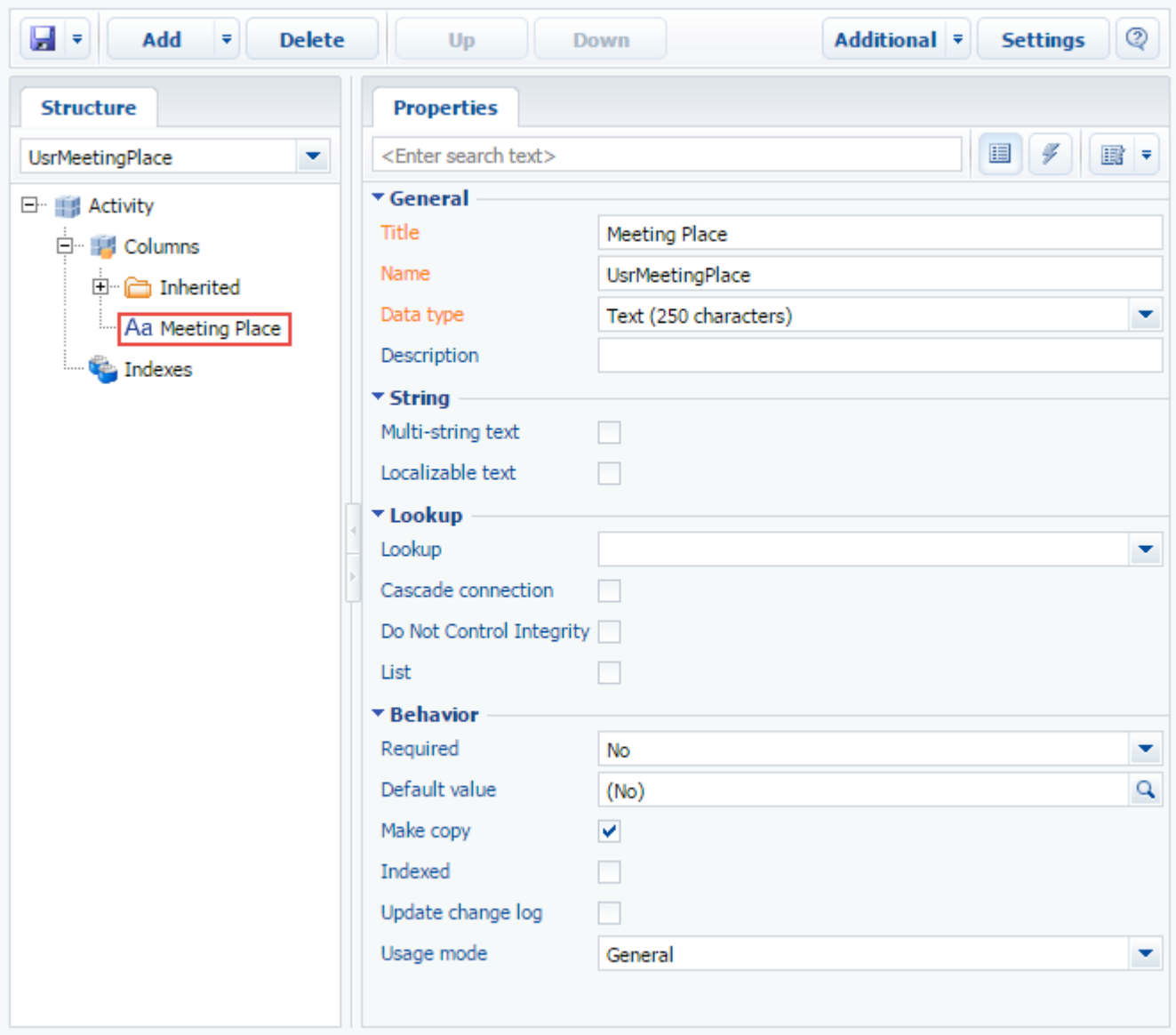
Fill the new object properties by specifying [Activity] as the parent object (Fig. 2).

Fig. 2. — Properties of the [Activity] replacement object



Add the new text column [Meeting Place] to the replacement object (Fig. 3.).

Fig. 3. – Adding the user column to the replacement object



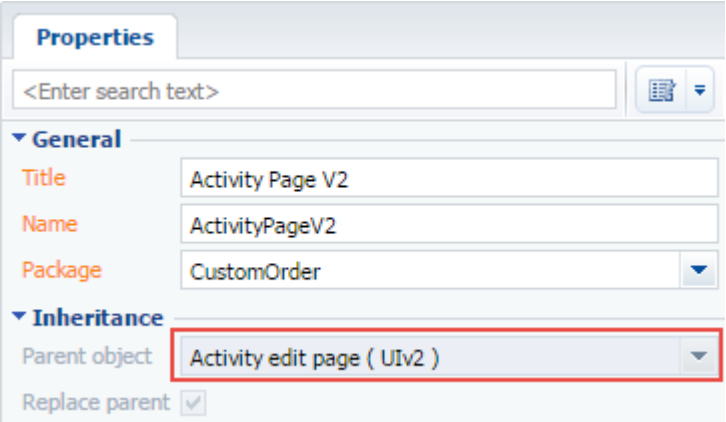
Publish the object.

2. Create a replacement client module of the activity edit page

A replacement client module must be created and [Activity edit page] must be specified as the parent object in it (Fig. 4).

The process of creating the replacement page is described in the article **Creating a custom client module schema**.

Fig. 4. — Properties of the replacement edit page



3. Add the [Meeting Place] field to the activity edit page

Add to the diff array a configuration object with the settings of the [Meeting Place] field location on the page.

4. Add a rule with the BINDPARAMETER type for the [UsrMeetingPlace] column to the rules property of the page view model

5. Initialize the property property of the rule for the [UsrMeetingPlace] column using the BusinessRuleModule.enums.Property.VISIBLE value

6. Add a condition for the rule implementation to the conditions array — the [Type] field value of the model must be equal to the identifier of the [Visit] type specified in the activities type lookup)

Below is the full code of the replacement contact page.

```
// Add module BusinessRuleModule to the list of the module dependencies.
define("ActivityPageV2", ["BusinessRuleModule", "ConfigurationConstants"],
    function (BusinessRuleModule, ConfigurationConstants) {
        return {
            // Schema name of the edit page object.
            entitySchemaName: "Activity",
            // Setup of additional field visualization on the edit page.
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            diff: /**SCHEMA_DIFF*/[
                {
                    // Metadata for adding the [Meeting place] field.
                    "operation": "insert",
                    "parentName": "Header",
                    "propertyName": "items",
                    "name": "UsrMeetingPlace",
                    "values": {
                        "layout": { "column": 0, "row": 5, "colSpan": 12 }
                    }
                }
            ]/**SCHEMA_DIFF*/,
            // Rule object of the edit page view model.
            rules: {
                // Rule set for the [UsrMeetingPlace] column of the view model.
                "UsrMeetingPlace": {
                    // Setup of the visibility rule for the [UsrMeetingPlace] field
by the
                    // [Type] field.
                    BindParametrVisibilePlaceByType: {
                        // BINDPARAMETER rule type.
                        ruleType: BusinessRuleModule.enums.RuleType.BINDPARAMETER,
                        // The rule regulates the VISIBLE property of the field.
```

```
                        property: BusinessRuleModule.enums.Property.VISIBLE,
                        // Array of conditions that trigger the rule.
                        // In this case the array contains single condition for
[Type]
                        // comparing the [Type] column with the [Visit] activity type
Id.
                        conditions: [{
                            // Expression of the left part of the condition.
                            leftExpression: {
                                // The ATTRIBUTE expression type points to the fact
that
                                // a view model attribute (column) is used as
expression.
                                type: BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                                // Name of the view model column, whose value is
compared in
                                // the expression.
                                attribute: "Type"
                            },
                            // Type of the comparison operation.
                            comparisonType: Terrasoft.ComparisonType.EQUAL,
                            // Expression of the right part of the condition.
                            rightExpression: {
                                // The CONSTANT expression type points to the fact
that a
                                // constant expression is used (the [Visit] activity
type Id).
                                type: BusinessRuleModule.enums.ValueType.CONSTANT,
                                // The value whith which the expression in the left
part.
                                // is compared. In this case the [Activity] type Id
is
                                // specified as a string.
                                value: "e3831dec-cfc0-df11-b00f-001d60e938c6"
                            }
                        }]
                    }
                }
            }
        };
    });
```

7. Save the created replacement page schema

When the schema is saved and the web-page of the system is updated on the activity edit page, the [Meeting Place] field will be displayed only if the activity type is Visit (Fig. 5, 6).

Fig. 5. — Demonstrating the case implementation result (the activity type is Call; no [Meeting Place] field is available).

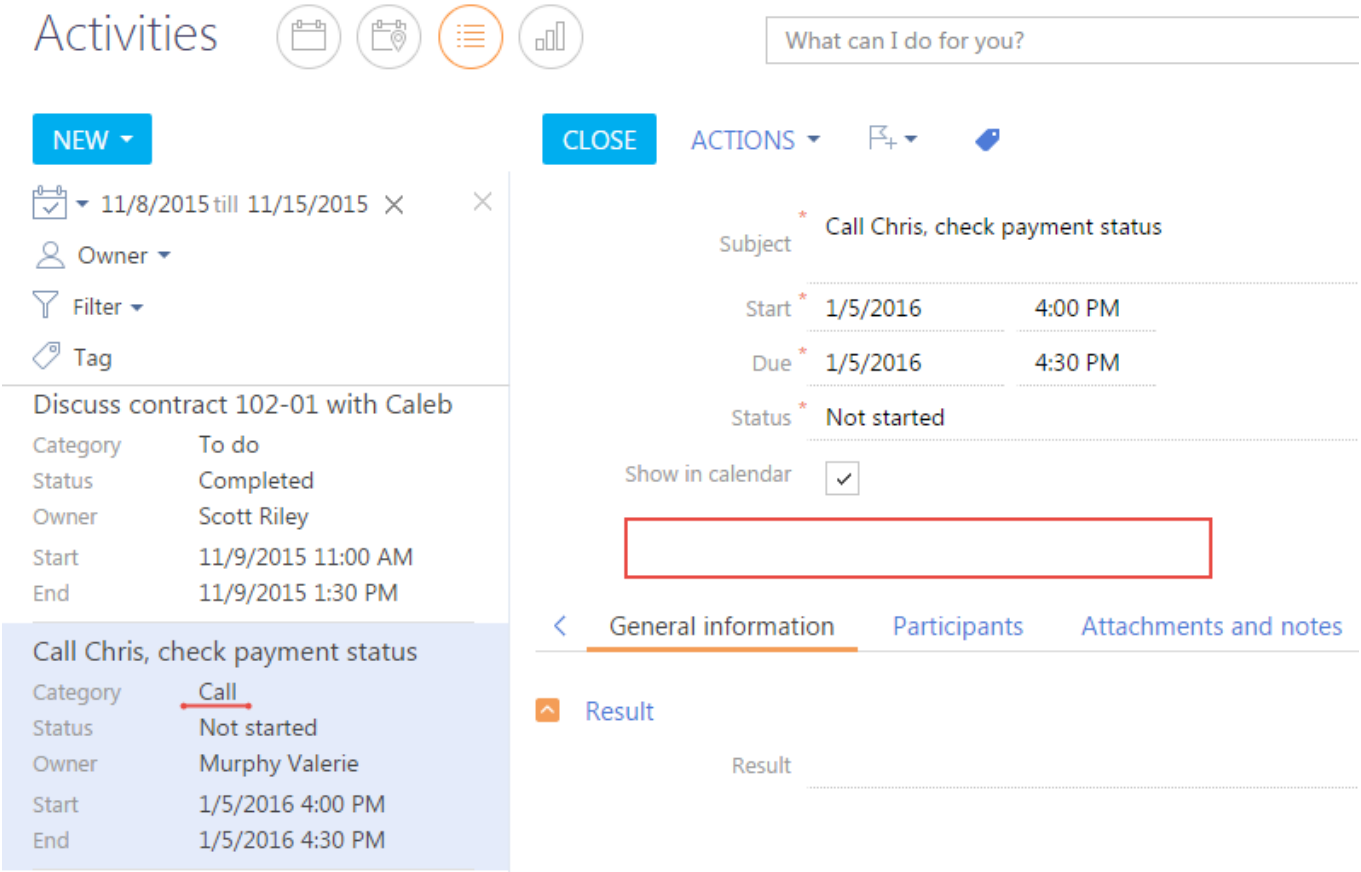
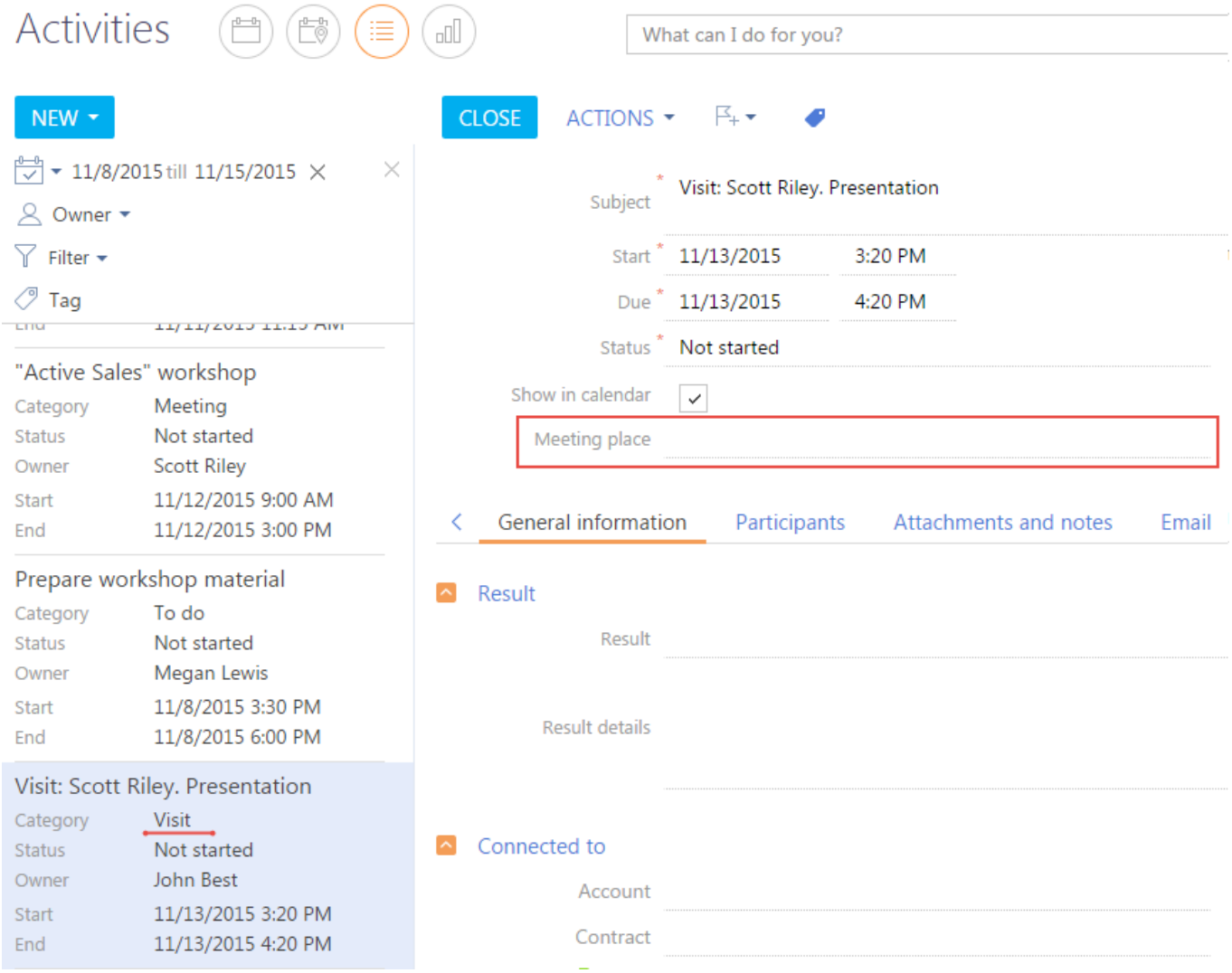


Fig. 6. — Demonstrating the case implementation result (the activity type is Visit; the [Meeting Place] field is displayed).



How to block the edit page field by a specific condition

Case description

Set fields on the contact edit page so that the [Account] field is accessible only when the [Name] field is filled.

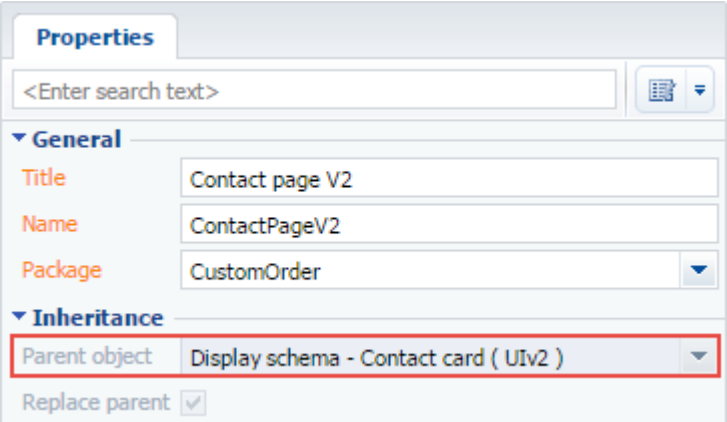
Case implementation algorithm

1. Create a replacement client module of the contact edit page

A replacement client module must be created and [Display schema – Contact card] (Fig. 1) must be specified as the parent object.

The process of creating the replacement page is described in the article **Creating a custom client module schema**

Fig. 1. — Properties of the replacement edit page



2. Add a rule with the BINDPARAMETER type for the [Account] column to the rules property of the page view model

3. Initialize the property property of the rule for the [Account] column using the BusinessRuleModule.enums.Property.ENABLED value

4. Add a condition for the rule implementation to the conditions array – the [Name] column value of the model cannot be undefined

```
// Add BusinessRuleModule to the list of module dependencies.
define("ContactPageV2", ["BusinessRuleModule", "ConfigurationConstants"],
    function (BusinessRuleModule, ConfigurationConstants) {
        return {
            // Object schema name of the edit page.
            entitySchemaName: "Contact",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/,
            // Rule object for the edit page view model.
            rules: {
                // Rule set for the [Account] column of the view model.
                "Account": {
                    // Rule that determines dependency of the [Account] field
availability
                    // from the value in the [Name] field.
                    BindParameterEnabledAccountByName: {
                        // BINDPARAMETER rule type.
                        ruleType: BusinessRuleModule.enums.RuleType.BINDPARAMETER,
                        // The rule regulates the ENABLED property of the field.
                        property: BusinessRuleModule.enums.Property.ENABLED,
                        // Array of conditions required for the rule to trigger.
                        // In this case the array contains single condition that
                        // determines whether the [Name] column has value.
                        conditions: [{
                            // Expression of the left part of the condition.
                            leftExpression: {
                                // The ATTRIBUTE expression type points to the fact
that
                                // a view model attribute (column) is used as
expression.
                                type: BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                                // Name of the view model column, whose value is
compared in
                                // the expression.
                                attribute: "Name"
                            },
                            // Type of the comparison operation.
                            comparisonType: Terrasoft.ComparisonType.NOT_EQUAL,
```

```
                                // Expression of the right part of the condition.
                                rightExpression: {
                                    // The CONSTANT expression type points to the fact
that
                                    // a constant expression is used.
                                    type: BusinessRuleModule.enums.ValueType.CONSTANT,
                                    // The value whith which the expression in the left
part.
                                    value: undefined
                                }
                            }]
                        }
                    }
                }
            };
        });
```

5. Save the created replacement page schema

When the schema is saved and the web-page of the system is updated on the contact edit page, the [Account] field will be uneditable until data are set into the [Full Name] field (Fig. 2, 3).

Fig. 2. — Demonstrating the case implementation result (the [Full Name] is not filled; the [Account] field is blocked).

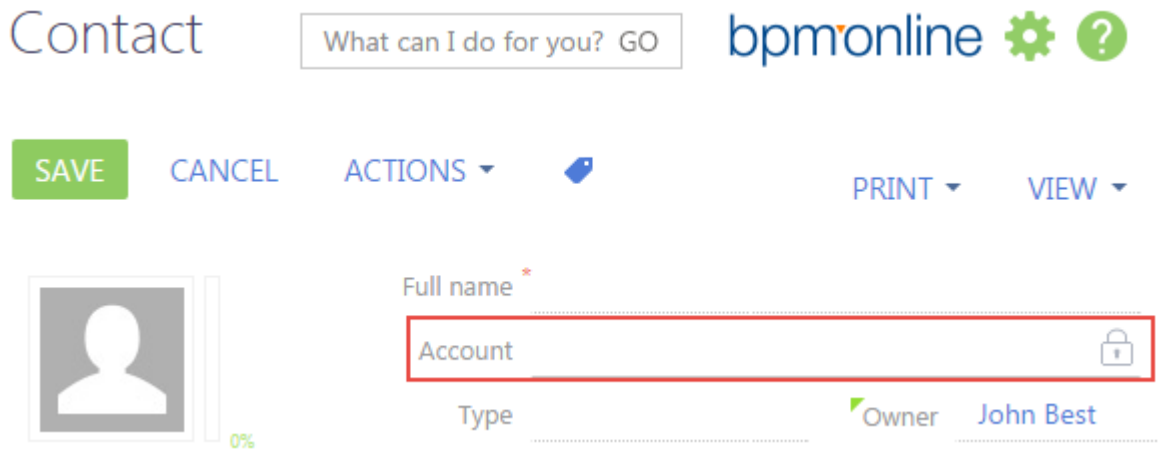
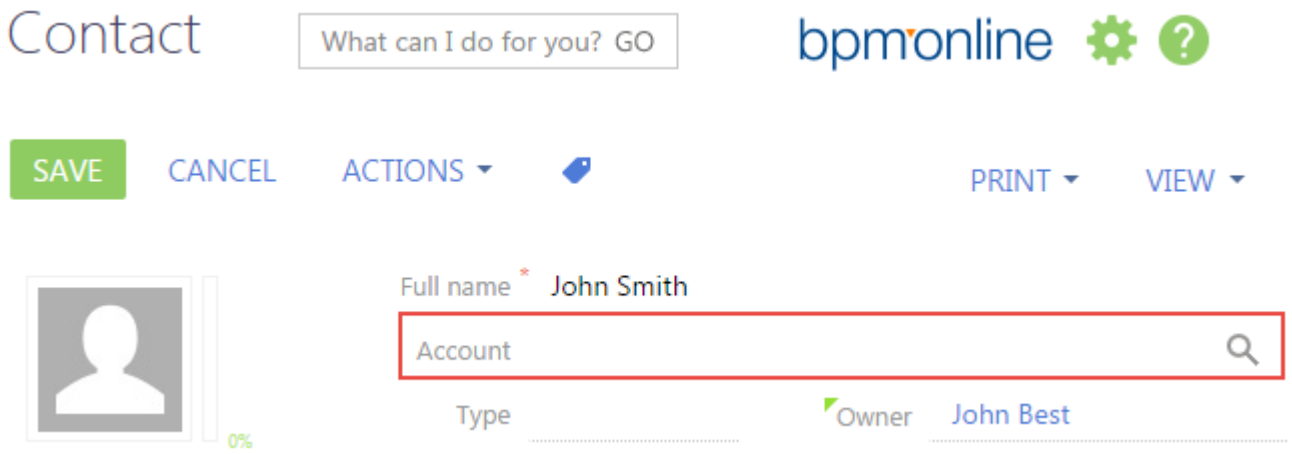


Fig. 3. — Demonstrating the case implementation result (the [Full Name] is filled; the [Account] field is available).



How to make the field required by a specific condition

Case description

Set fields on the contact edit page so that the [Account] field is required only when the field [Contact Type] = Customer.

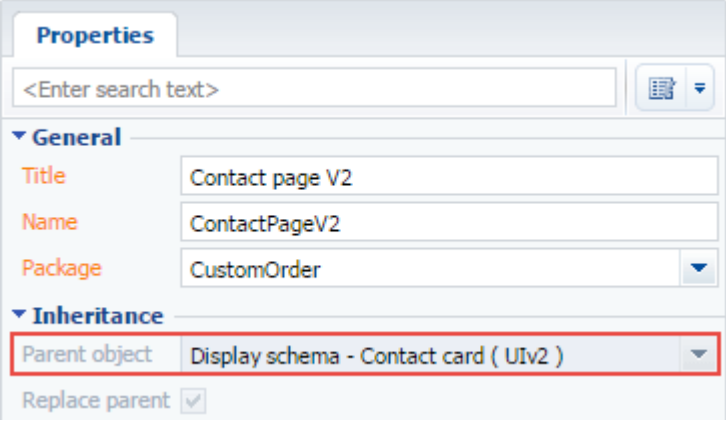
Case implementation algorithm

1. Create a replacement client module of the contact edit page

A replacement client module must be created and [Display schema – Contact card] (Fig. 1) must be specified as the parent object.

The process of creating the replacement page is described in the article **Creating a custom client module schema**

Fig. 1. — Properties of the replacement edit page



2. Add a rule to the rules property of the page view model

Add a rule with the BINDPARAMETER type for the [Account] column.

3. Set the property property of the rule for the [Account] column

Initialize the property property using the BusinessRuleModule.enums.Property.REQUIRED value.

4. Add a condition for the rule implementation to the conditions array

The rule is implemented if the [Type] column value of the view model is equal to the value of the [Customer] contact type identifier.

```
// Add BusinessRuleModule and ConfigurationConstants to the list of module
dependencies.
define("ContactPageV2", ["BusinessRuleModule", "ConfigurationConstants"],
    function (BusinessRuleModule, ConfigurationConstants) {
        return {
            // Object schema name of the edit page.
            entitySchemaName: "Contact",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/,
            // Rule object for the edit page view model.
            rules: {
                // Rule set for the [Account] column of the view model.
                "Account": {
                    // Rule that determines whether the [Account] field is required
                    // depending on the value in the [Type] field.
```

```
                    BindParameterRequiredAccountByType: {
                        // BINDPARAMETER rule type.
                        ruleType: BusinessRuleModule.enums.RuleType.BINDPARAMETER,
                        // The rule regulates the REQUIRED property of the field.
                        property: BusinessRuleModule.enums.Property.REQUIRED,
                        // Array of conditions required for the rule to trigger.
                        // In this case the array contains single condition that
                        // determines whether the value in the [Type] column is
"Customer".
                        conditions: [{
                            // Expression of the left part of the condition.
                            leftExpression: {
                                // The ATTRIBUTE expression type points to the fact
that
                                // a view model attribute (column) is used as
expression.
                                type: BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                                // Name of the view model column, whose value is
compared in the expression.
                                // In this case a lookup column is specified,
therefore
                                // the attributePath property must be additionally
specified.
                                attribute: "Type",
                                // Name of the lookup column specified in the
attribute property.
                                attributePath: "Id"
                            },
                            // Type of the comparison operation.
                            comparisonType: Terrasoft.ComparisonType.EQUAL,
                            // Expression of the right part of the condition.
                            rightExpression: {
                                // The CONSTANT expression type points to the fact
that
                                // a constant expression is used.
                                type: BusinessRuleModule.enums.ValueType.CONSTANT,
                                // The value whith which the expression in the left
part
                                // is compared. In this case the expression is the
ContactType
                                // enumeration from the ConfigurationConstants
module.
                                value: ConfigurationConstants.ContactType.Client
                            }
                        }]
                    }
                }
            }
        };
    });
```

5. Save the created replacement page schema

When the schema is saved and the web-page of the system is updated on the contact edit page, the [Account] field is required if the contact type is Customer (Fig. 2, 3, 4).

Fig. 2. — Demonstrating the case implementation result (the type is not filled; the [Account] field is optional)

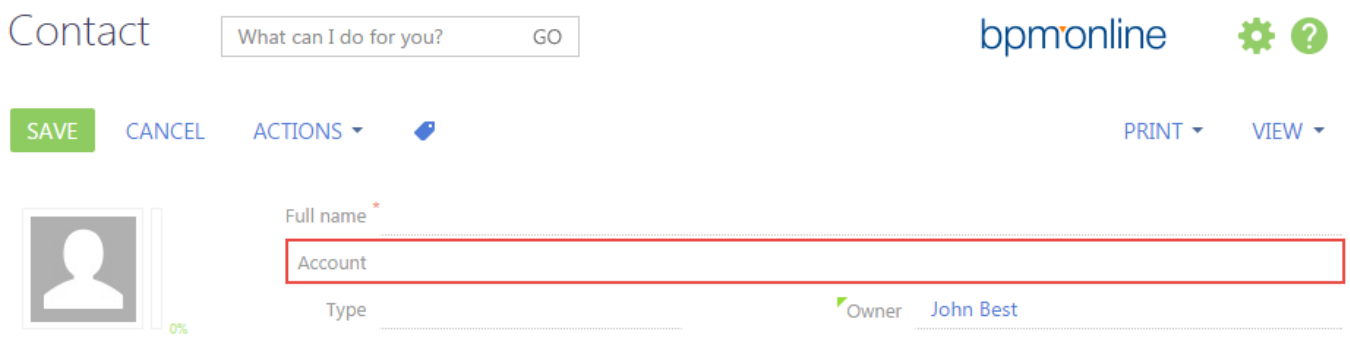


Fig. 3. — Demonstrating the case implementation result (the type is Customer; the [Account] field is mandatory)

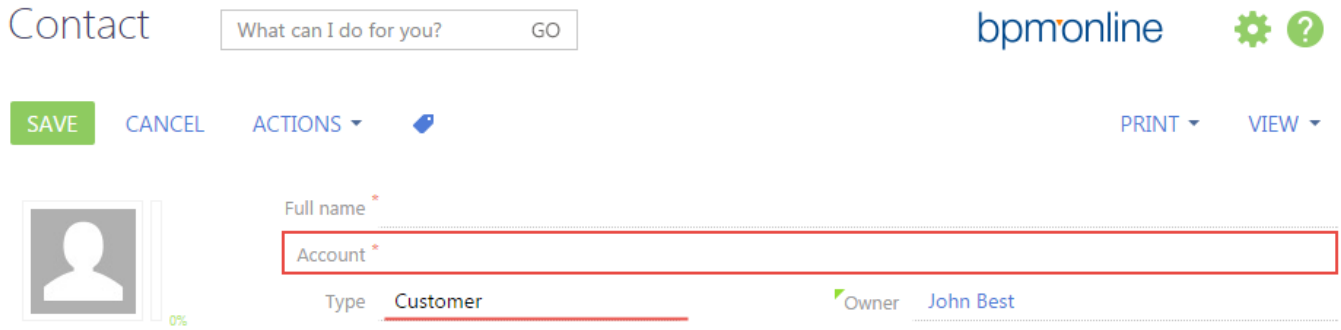


Fig. 4. — Demonstrating the case implementation result (the type is Contact person; the [Account] field is optional)

Business rules created in wizards

Introduction

In bpm'online version 7.10.0 and higher, in addition to **custom business rules created by developers**, there are business rules created in wizards. Objects for configuring these business rules are specified in the *businessRules* property of the page view model schema (see "**Business rules. The businessRules property**"). Using the enumerations defined in the *BusinesRuleModule* in these objects is forbidden.

Additional properties

Additional properties of business rules defined in the of the businessRules property configuration objects of the edit page view model schema are shown in Table 1.

Table 1. Additional properties

Property	Description
uId	The unique rule identifier. Assigned a GUID-type value.
enabled	A checkbox indicating whether the rule is active. Available values – *true* or *false*.
removed	A checkbox indicating whether the rule is deleted. Available values – *true* or *false*.
invalid	A checkbox indicating whether the rule is valid. Available values – *true* or *false*.

Examples

An example of a business rule created in the wizard, connected to the field property (visible, active, mandatory):

```
define("SomePage", [], function() {
  return {
     // ...
     businessRules: /**SCHEMA_BUSINESS_RULES*/{
        // A set of rules for the [Type] column of the view model.
        "Type": {
           // Rule code generated by the wizard.
           "ca246daa-6634-4416-ae8b-2c24ea61d1f0": {
              // Unique rule identifier.
              "uId": "ca246daa-6634-4416-ae8b-2c24ea61d1f0",
              // A checkbox indicating whether the rule is active.
              "enabled": true,
              // A checkbox indicating whether the rule is deleted.
              "removed": false,
              // A checkbox indicating whether the rule is valid.
              "invalid": false,
              // Rule type.
              "ruleType": 0,
              // The code of a property that governs the rule.
              "property": 0,
              // Logical connection between several rules.
              "logical": 0,
              // A conditions array in which the rule is triggered.
              // In this case, the array contains one condition for comparing the
value
              // with the [Type] column value through a direct path
([Account.PrimaryContact.Type]).
              "conditions": [
                 {
                    // Type of comparison.
                    "comparisonType": 3,
                    // Expression of the left side of the condition.
                    "leftExpression": {
                       // Expression type.
                       "type": 1,
                       // Name of the view model column.
                       "attribute": "Account",
                       // Diract path, the value of which if compared in the
expression.
                       "attributePath": "PrimaryContact.Type"
                    },
                    // Expression of the right side of the condition.
                    "rightExpression": {
                       // Condition type.
```

```
                        "type": 1,
                        //  Name of the view model column.
                        "attribute": "Type"
                    }
                }
            ]
        }
    }
  }/**SCHEMA_BUSINESS_RULES*/
  // ..
};
});
```

A field filtering business rule created in a wizard:

```
define("SomePage", [], function() {
  return {
    // ...
    businessRules: /**SCHEMA_BUSINESS_RULES*/{
      // Набор правил для колонки [Type] модели представления.
      "Account": {
        // The rule code generated by the wizard.
        "a78b898c-c999-437f-9102-34c85779340d": {
          // Unique rule identifier.
          "uId": "a78b898c-c999-437f-9102-34c85779340d",
          // A checkbox indicating whether the rule is active.
          "enabled": true,
          // A checkbox indicating whether the rule is deleted.
          "removed": false,
          // A checkbox indicating whether the rule is valid.
          "invalid": false,
          // Rule type.
          "ruleType": 1,
          // The path to the column for filtering
          // in the [Account] lookup schema, which is
          // referenced by the [Type] column of the
          // edit page view model.
          "baseAttributePatch": "PrimaryContact.Type",
          // Type of comparison in the filter.
          "comparisonType": 3,
          // The column (attribute) of the page model acts as a value in the
comparison process.
          "type": 1,
          // The column name of the edit page view model, the value of which will
be filtered.
          "attribute": "Type"
        }
      }
    }/**SCHEMA_BUSINESS_RULES*/
    // ..
  };
});
```

Adding an action to the edit page

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.

```
/**
* Returns the collection of 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 transmitted configuration object is transmitted as callback.
    actionMenuItems.addItem(this.getButtonMenuItem({
        // Configuration object for action setting.
    }));
    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 horizonal 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	
Visible	a logical property controlling the menu item visibility	

Procedure for adding a custom action

1. Override the getActions 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 getActions method of the replacing module, the parent implementation of this method must be called first to initialize actions of the parent section.

Example of implementing the action adding to the edit page

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.

> 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 obejct 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.

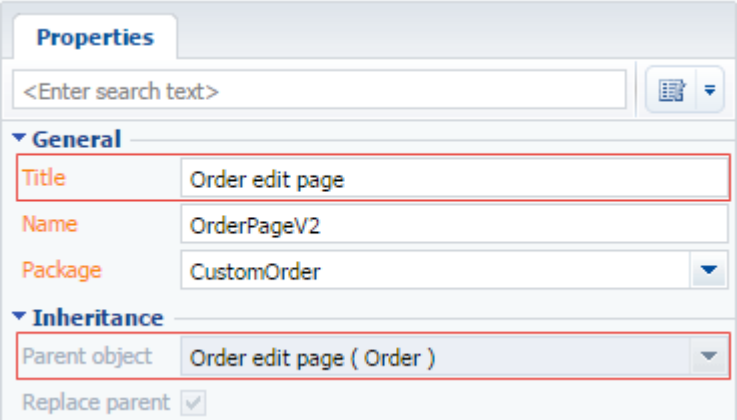
Case implementation algorithm

1. Create a replacingedit page for an order in a custom package

A replacing client module must be created and [Order edit page] must be specified as the parent object in it (Fig. 1).

The procedure for creating the replacingpage is described in the article **Creating a custom client module schema**.

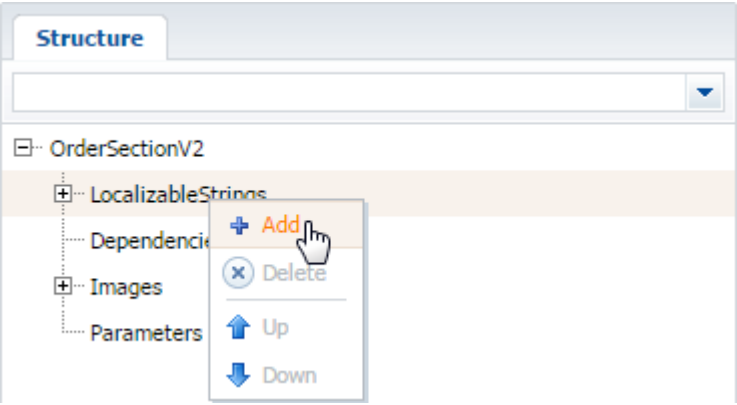
Fig. 1. — Properties of the replacing edit page



2. Add a string with the [Actions] menu item to the localizable strings collection of the page replacing schema

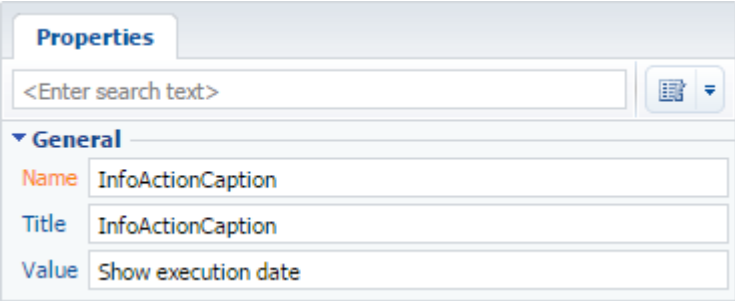
For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure node (Fig. 2).

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 page view model

- isRunning - checks whether the order is at the In-progress stage;
- ShowOrderInfo action handler method - displays the scheduled end date of the order in the data window.

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("OrderPageV2", ["OrderConfigurationConstants"],
    function(OrderConfigurationConstants) {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Order",
            // Methods collection of the edit page view model.
            methods: {
                // Method which checks the stage of the order execution.
                isRunning: function() {
                    // The status value of the selected order is compared to the "In-
progress" status value and the method
                    // returns true or false depending on the comparison result.
                    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);
                }
            }
        };
});
```

4. Override the getActions virtual method where the handler method must be bound to an action

For this purpose, add the code of the redefined method to the methods collection.

```
// 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 for visual separation of the custom action from the
```

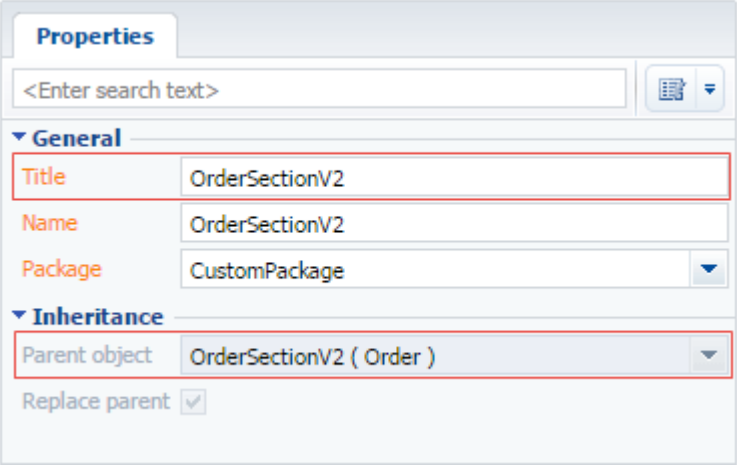
```
actions list
    // of the base page.
    actionMenuItems.addItem(this.getButtonMenuItem({
        Type: "Terrasoft.MenuSeparator",
        Caption: ""
    }));
    // Adding a menu item to the actions list of the edit page.
    actionMenuItems.addItem(this.getButtonMenuItem({
        // Binding 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.
        "Visible": {bindTo: "isRunning"}
    }));
    return actionMenuItems;
}
```

5. Publish the created replacingschema

6. Create a replacingschema for the [Orders] section in a custom package

Specify OrderSectionV2 as the parent object (Fig. 4).

Fig. 4. — Properties of the [Orders] section replacing page



7. Add a string with the [Actions] menu item to the collection of localizable strings in the replacing section schema

For this purpose, repeat the actions in par. 2 by filling in the string properties as shown in Fig. 3.

8. Add the isRunning method implementation into the methods collection of the section view model. This will check whether the selected order is at the In-progress stage

For this purpose, add the program code of the section replacingreplacing module to the source code tab. A required method is added in it to the methods collection of the view model.

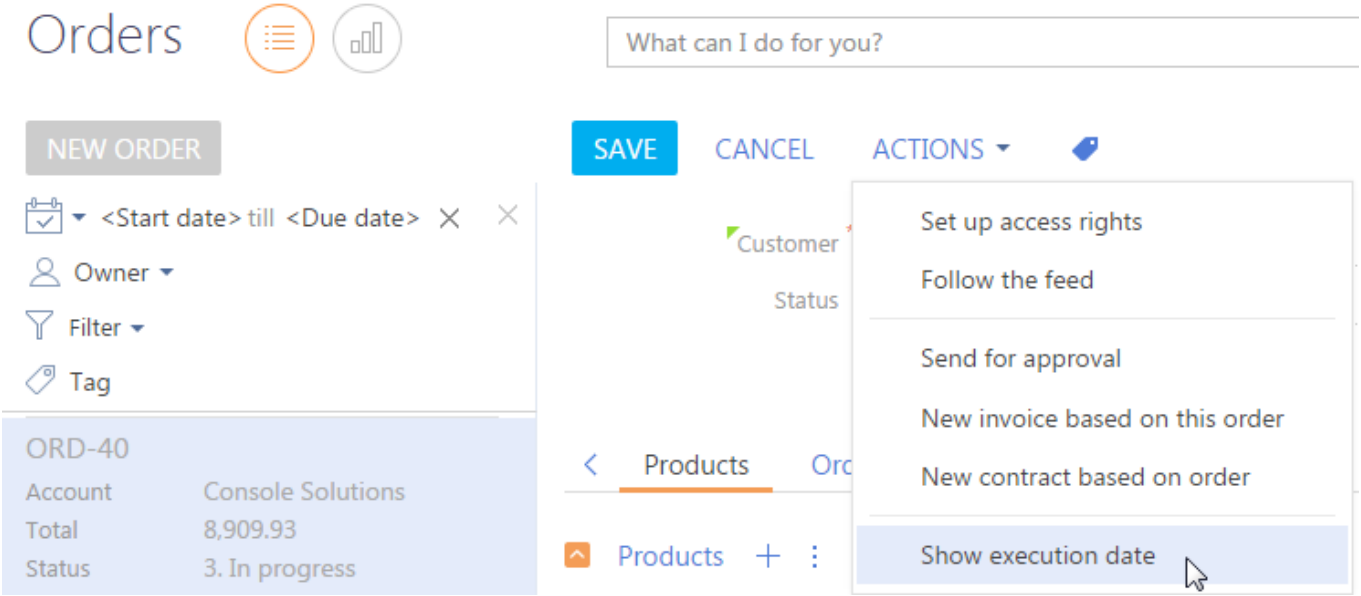
```
define("OrderSectionV2", ["OrderConfigurationConstants"],
    function(OrderConfigurationConstants) {
        return {
            // Name of the section schema.
            entitySchemaName: "Order",
            // Methods collection of the section view model.
            methods: {
```

```
                // Method checking the stage of the selected order to determine the
menu item visibility.
               isRunning: function(activeRowId) {
                   var activeRowId = this.get("ActiveRow");
                   // Receiving the data collection of the list view of the [Order]
section list.
                   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;
               }
           }
       };
   });
```

9. Save the created section replacing schema

When the schema is saved and the system web-page is updated, a new action will appear on the order edit page when selecting an order at the "In progress" stage (Fig. 5).

Fig. 5. – Demonstrating the case implementation result



Control elements

- **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**

Adding a new field to the edit page

A field may be added to the edit page by two ways:

1. With the help of the Section Wizard (see articles Section Wizard, Setting up the page fields).

Using the Section Wizard in the custom package will result in creating a base object replacingschema (for example, [Activity] object and replacing schema for the base edit page (for example, [Activity edit page]).

A new column will be added to the object replacing schema.

A configuration object with settings of the new field location on the page will be added to the diff array in the replacing edit page schema.

Additional business logic of the edit page or development of new items for the client interface must be implemented in the created replacingschema of the edit page.

2. By creating a replacingbase object and replacing base page.

Below is the example of this method.

Case description

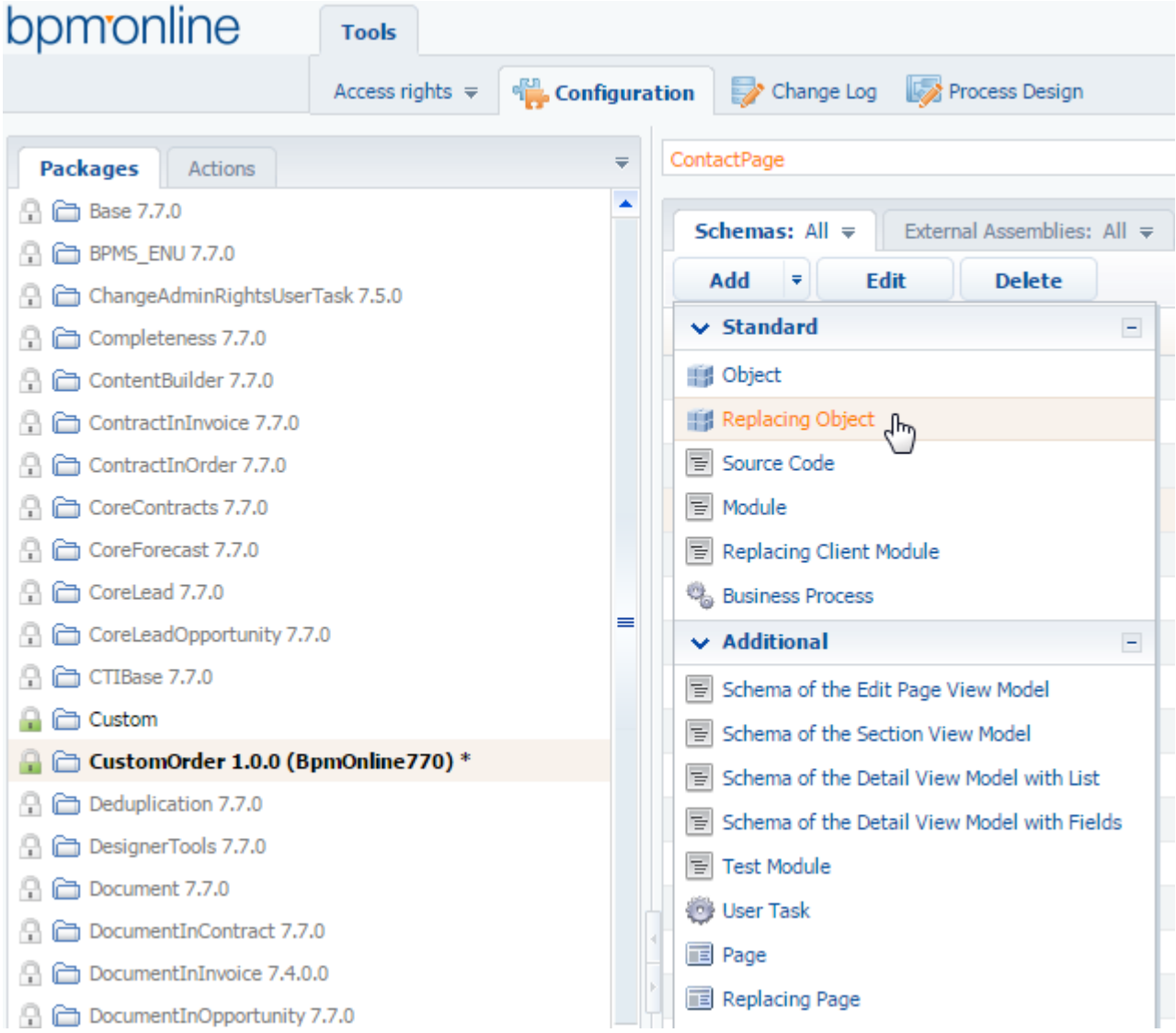
Add the new [Meeting Place] field to the activity edit page.

Case implementation algorithm

1. Create the [Activity] replacing object and add the new [Meeting Place] column to it

For this purpose, select a custom package and run the [Add] > [ReplacingObject] menu item on the [Schemas] tab (Fig. 1).

Fig. 1. – Creating a replacing object

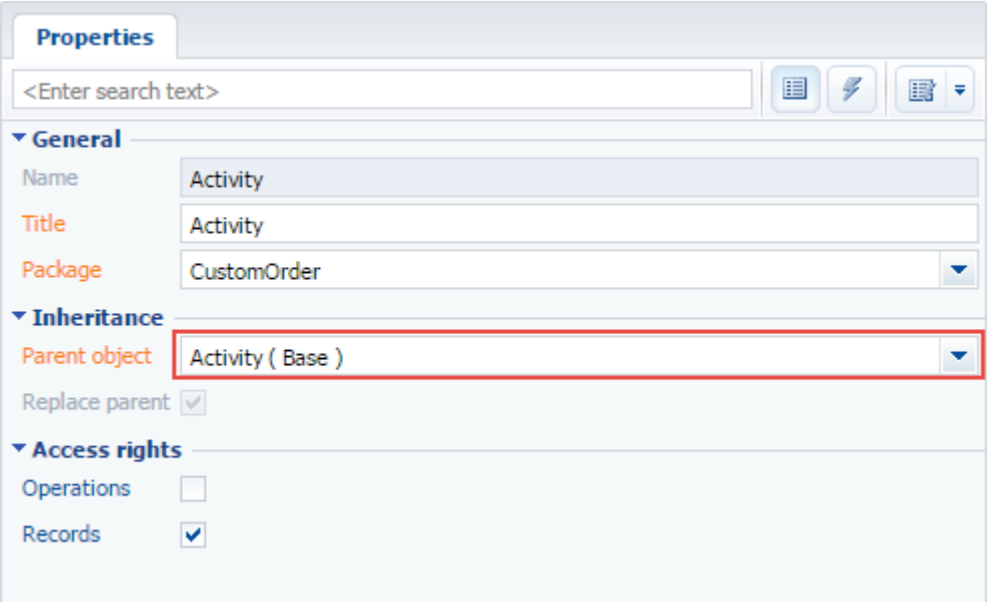


For the module to replace a specific object, the name of the base object to replace must be specified in the [Parent Object] field.

This means that to create the [Activity] replacingobject, it must be specified as the parent object.

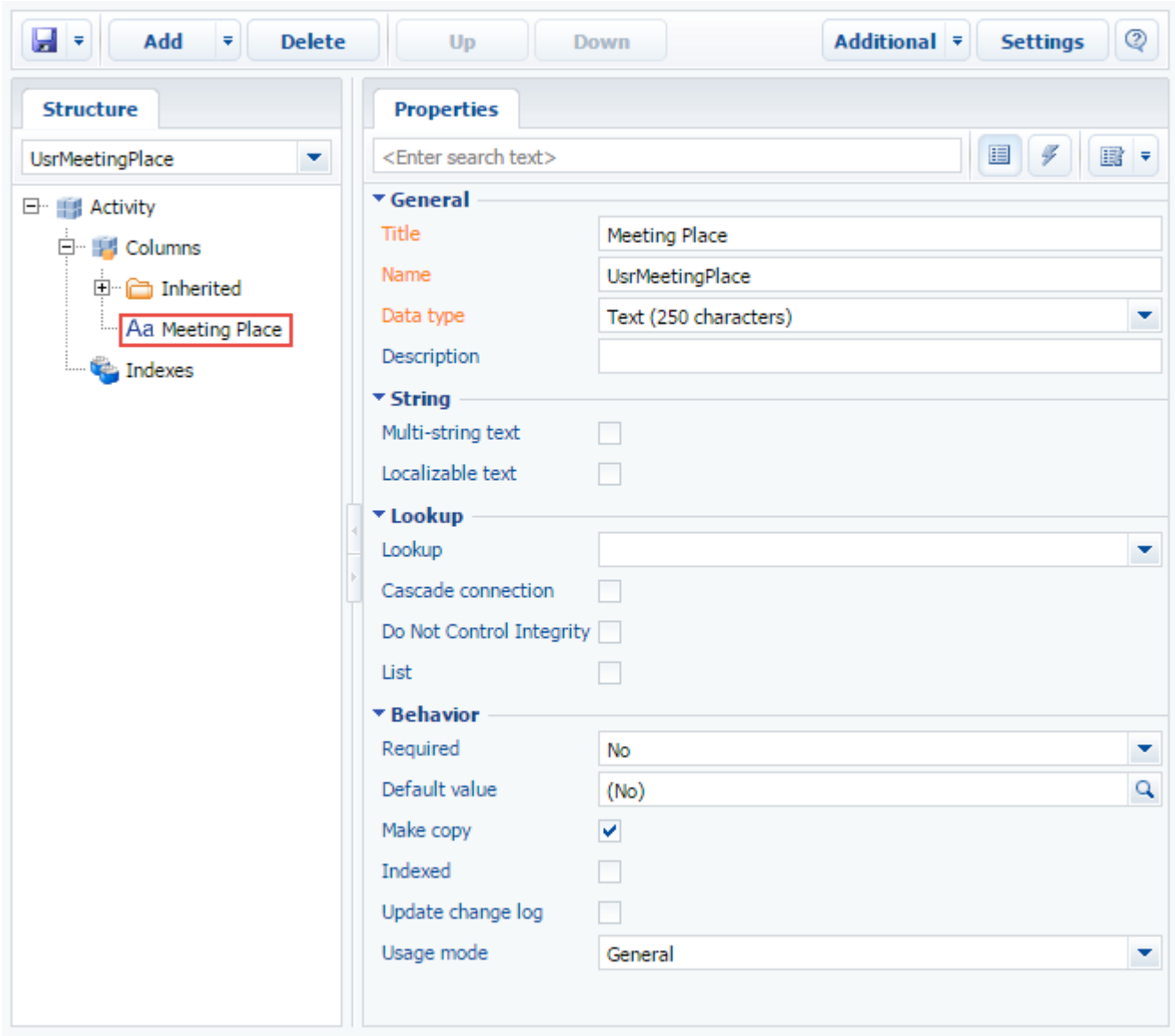
When the selected parent object is selected, the remaining properties fields are filled in automatically (Fig. 2).

Fig. 2. — Properties of the [Activity] replacing object



Add the new [Meeting Place] column to the replacing object (Fig. 3.).

Fig. 3. – Adding the user column to the replacing object



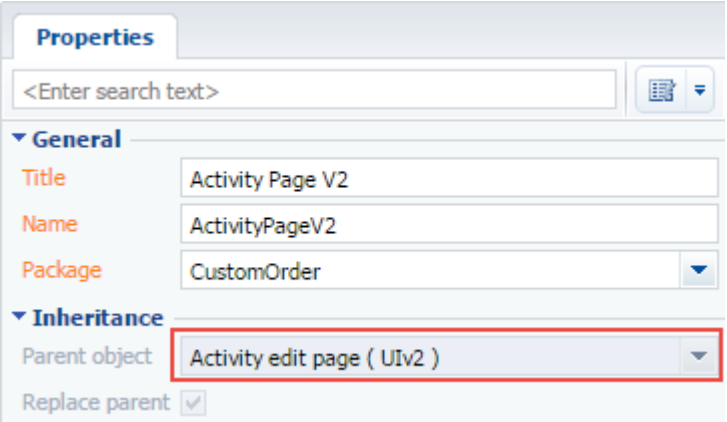
Publish the object.

2. Create a replacing client module of the activity edit page

A replacing client module must be created and [Activity edit page] must be specified as the parent object (Fig. 4).

The procedure for creating the replacing page is described in the article **Creating a custom client module schema**.

Fig. 4. — Properties of the replacing edit page



3. Add the [Meeting Place] field to the activity edit page

For this purpose, add a configuration object with settings of the [UsrMeetingPlace] field location on the page to the diff array.

```
define("ActivityPageV2", [],
    function() {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Activity",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            // Setting the visualization of the additional field on the edit page.
            diff: /**SCHEMA_DIFF*/[
               {
                   // Metadata for adding the [Meeting Place] field.
                   "operation": "insert",
                   "parentName": "Header",
                   "propertyName": "items",
                   "name": "UsrMeetingPlace",
                   "values": {
                       "layout": {"column": 0, "row": 5, "colSpan": 12}
                   }
                }
            ]/**SCHEMA_DIFF*/
        };
    });
```

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:

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:

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).	Properties not required for the initialization, but which allow setting the dynamic behavior and supplementary properties for a control item.
visible	Controls the button visibility.	
style	Setup of custom styles.	

Example of implementing a button adding to the edit page

- **How to add a button to an existing record edit page**
- **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

Example of adding 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.

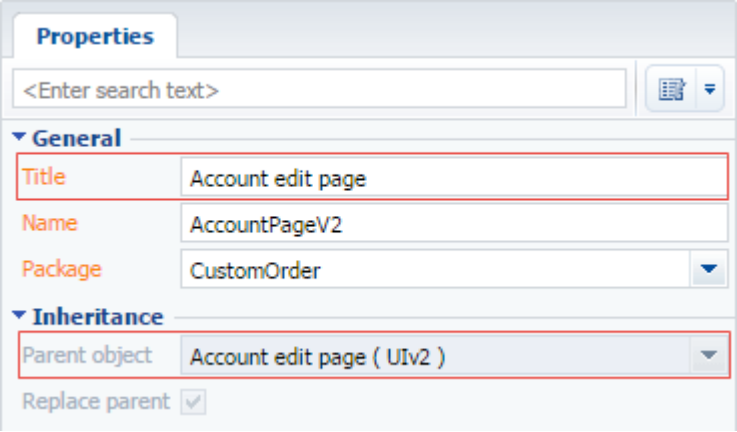
Case implementation algorithm

1. Create a replacingaccount edit page

A replacingclient module must be created and [AccountPageV2] must be specified as the parent object in it (Fig. 1).

The procedure for creating the replacingpage is described in the article **Creating a custom client module schema**.

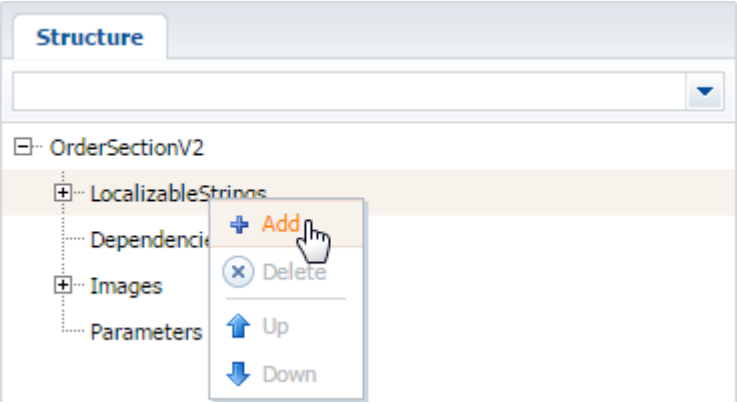
Fig. 1. — Properties of the replacing edit page



2. Add a string with the button name to the localizable strings collection of the page replacing schema

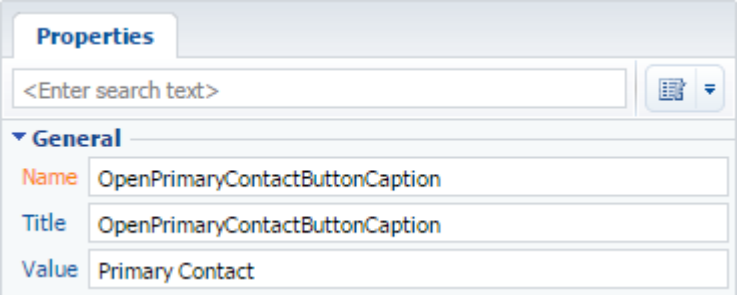
For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure node (Fig. 2).

Fig. 2. — Adding a localizable string to the schema



Fill in properties for the created string as shown in Fig. 3.

Fig. 3. — Properties of a custom localized string



3. Add the method implementation to the methods collection of the page 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 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 [Primary contact] field is
completed.
                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
                        });
                    }
```

```
                }
            }
        };
    });
```

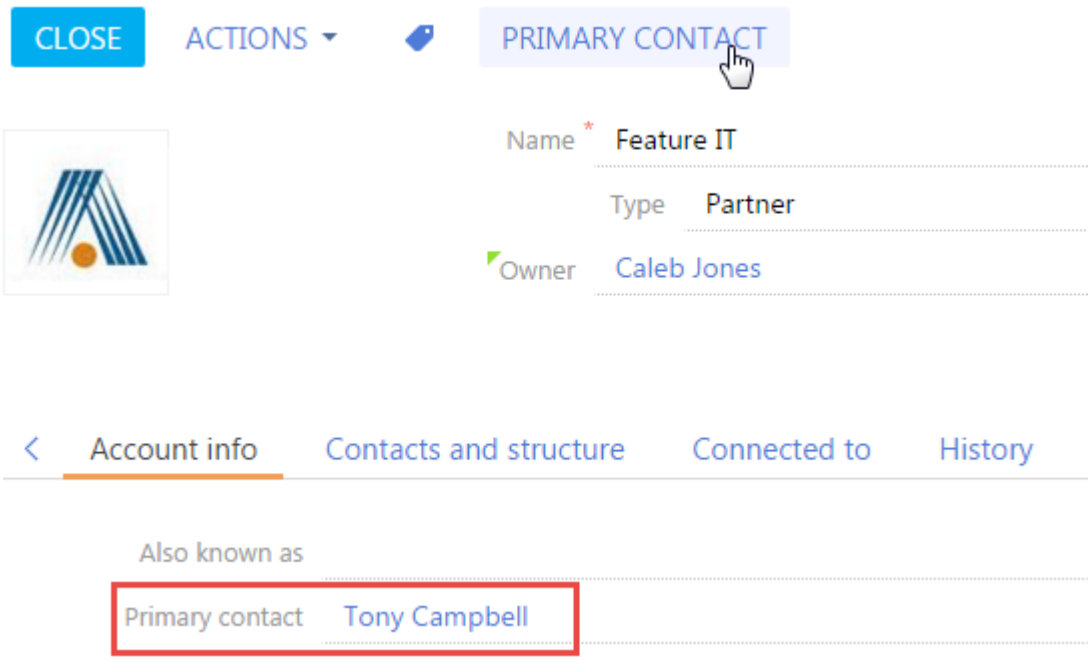
4. Add a configuration object to the diff array with button location settings on a page

```
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",
                    // Metadata 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": "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 page replacing schema

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



This button will be displayed on the page in the new record add mode. To display the button on the page, the case **How to add a button to an existing record edit page** must be implemented in the edit mode.

How to add a button to an existing record edit page

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.

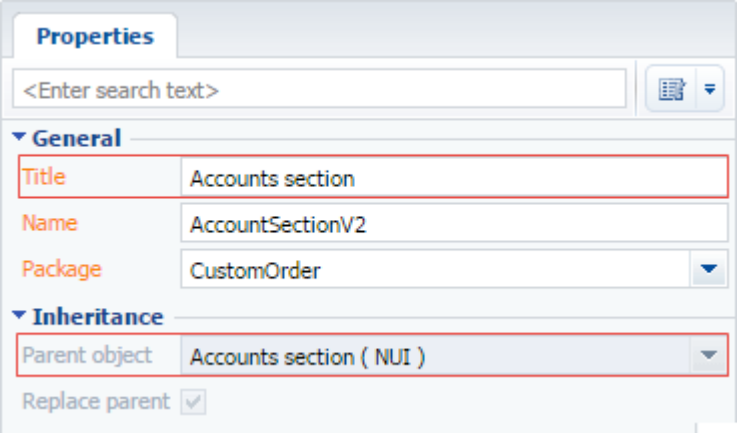
Case implementation algorithm

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**.

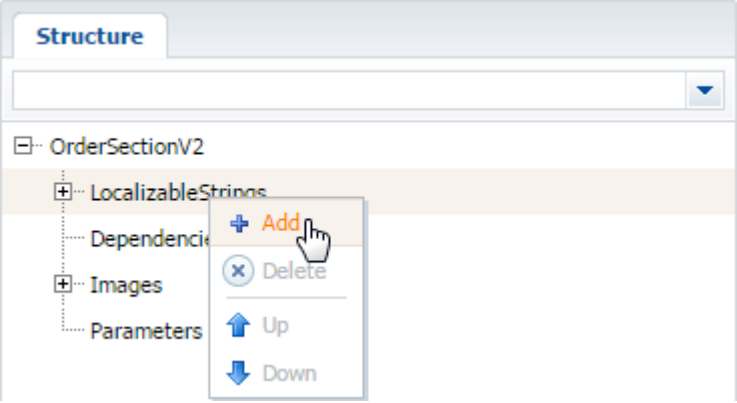
Fig. 1. — Properties of the section replacing page



2. Add a string with the button name to the localizable strings collection of the section replacing schema

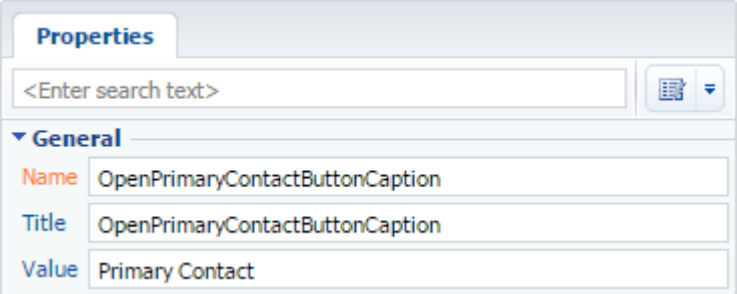
For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure node (Fig. 2).

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
                        }
                    }
                }
            ]
        };
    });
```

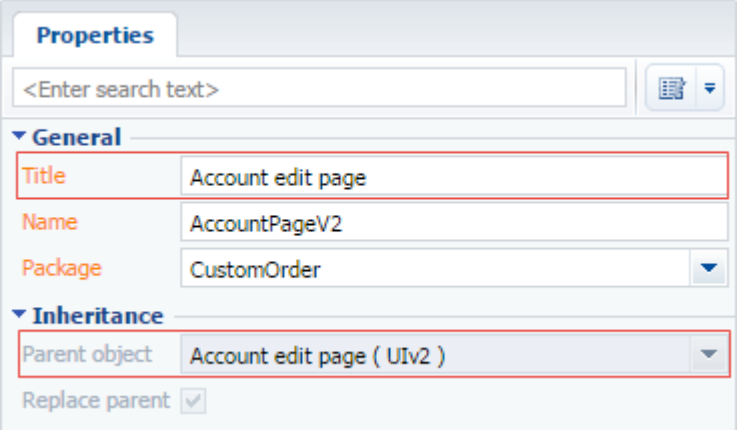
5. Save the created replacingpage schema

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**.

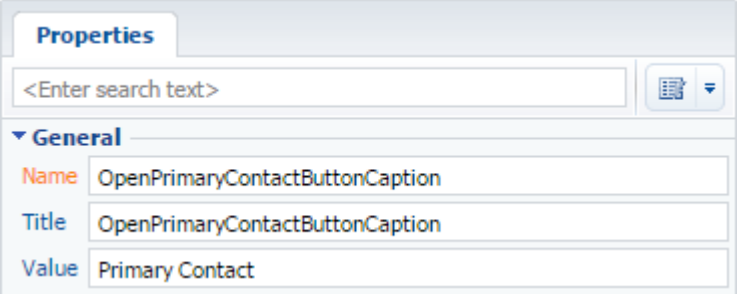
Fig. 4. — Properties of the replacing edit page



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

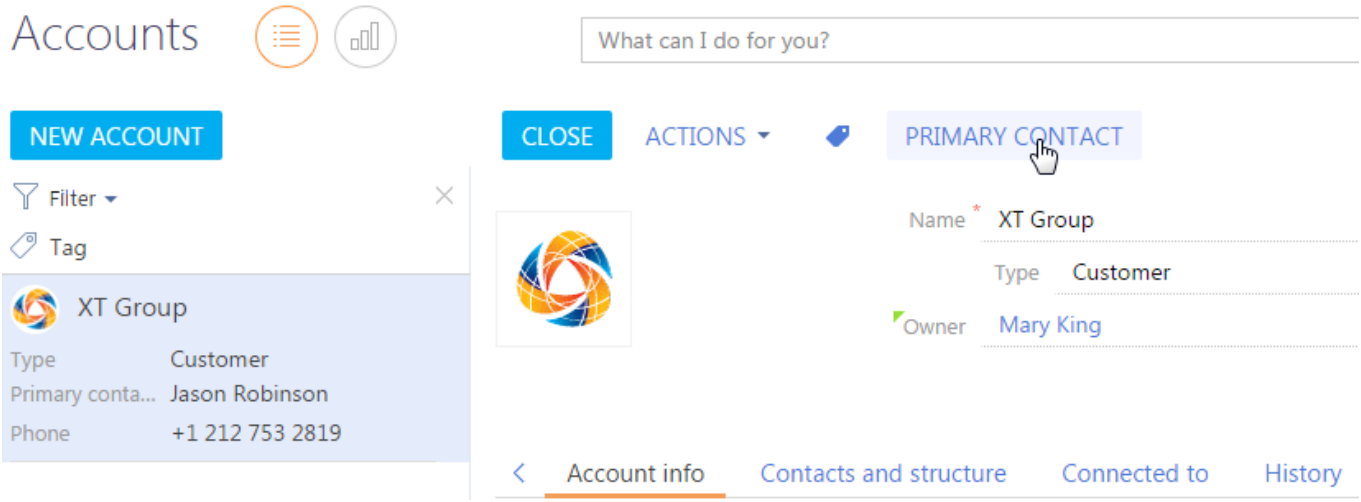
```
define("AccountPageV2", [],
    function() {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Account",
            // Methods collection of the edit page view model.
            methods: {
                // onOpenPrimaryContactClick, isAccountPrimaryContactSet 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

Adding a field with an image to the edit page (contact image, product image, account agent, etc.) has the following specifics:

1. The object column to be used for a field with an image must be of the "Image Link" type.
2. A default image must be added to the images collection of the edit page schema.
3. The field with the image must be added to the diff array of the edit page using an auxiliary wrapper container with the ["image-edit-container"] class.
4. The configuration object with the settings of the field with the image must contain the following properties in the values property:
    - getSrcMethod — the name of the method receiving an image by link;
    - onPhotoChange — the name of the method called when an image is modified;

- readonly — the property determining a possibility to edit (modify, delete) the image;
- defaultImage — an image to be displayed by default;
- generator — — the control item generator, "ImageCustomGeneratorV2.generateCustomImageControl", must be specified for the field with the image.

5. The following methods must be added to the methods collection of the edit page schema:
   - method for receiving an image by a link;
   - method called when an image is modified;
   - method for storing a link to a modified image in the object column.

Examples of adding a field with an image to the edit page are provided below.

Case description

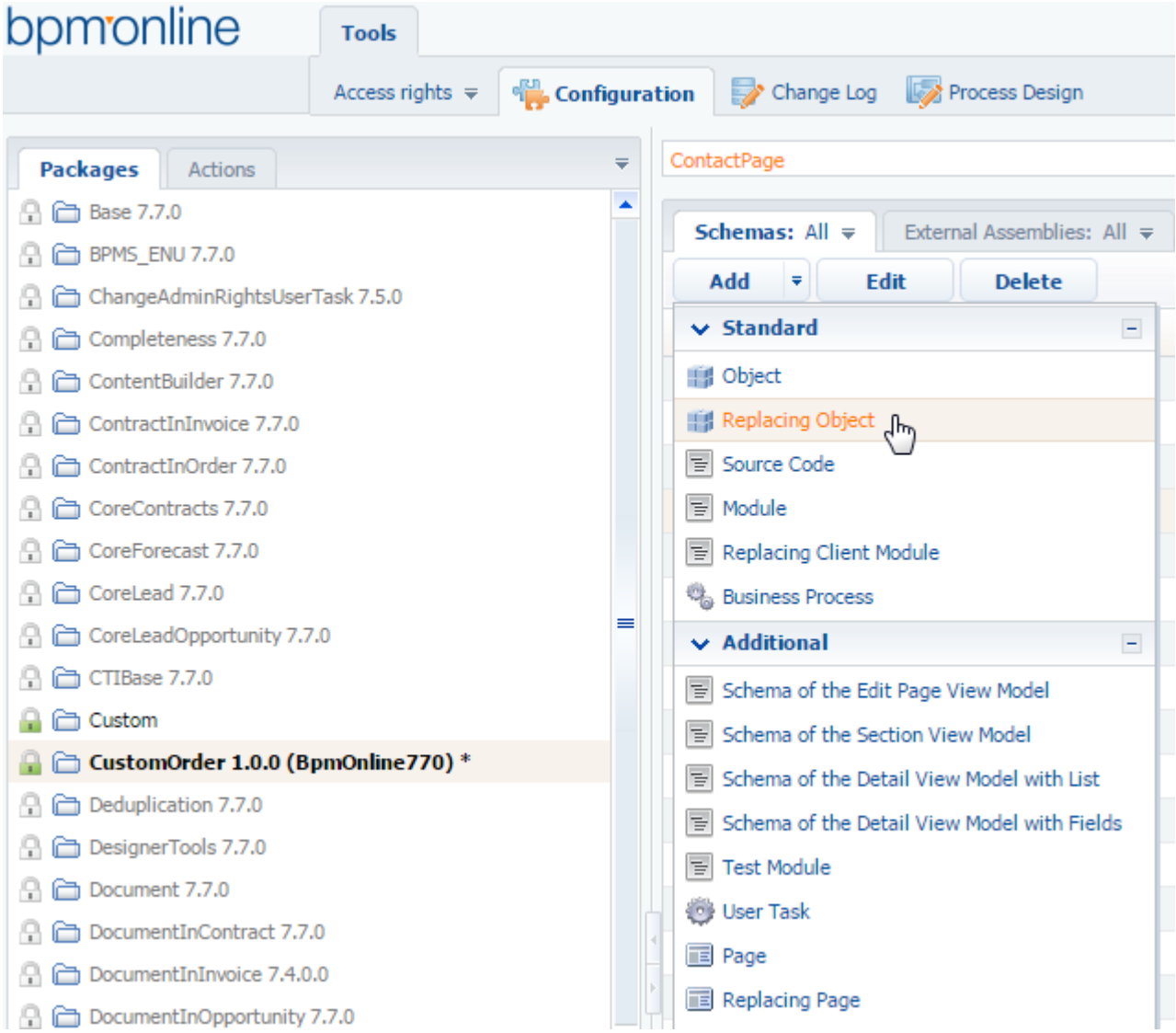
Add a field with a logo to the account edit page.

Case implementation algorithm

1. Create the [Account] replacing object

For this purpose, select a custom package and run the [Add] > [Replacing Object] (Fig. 1).

Fig. 1. – Creating the object replacing chema

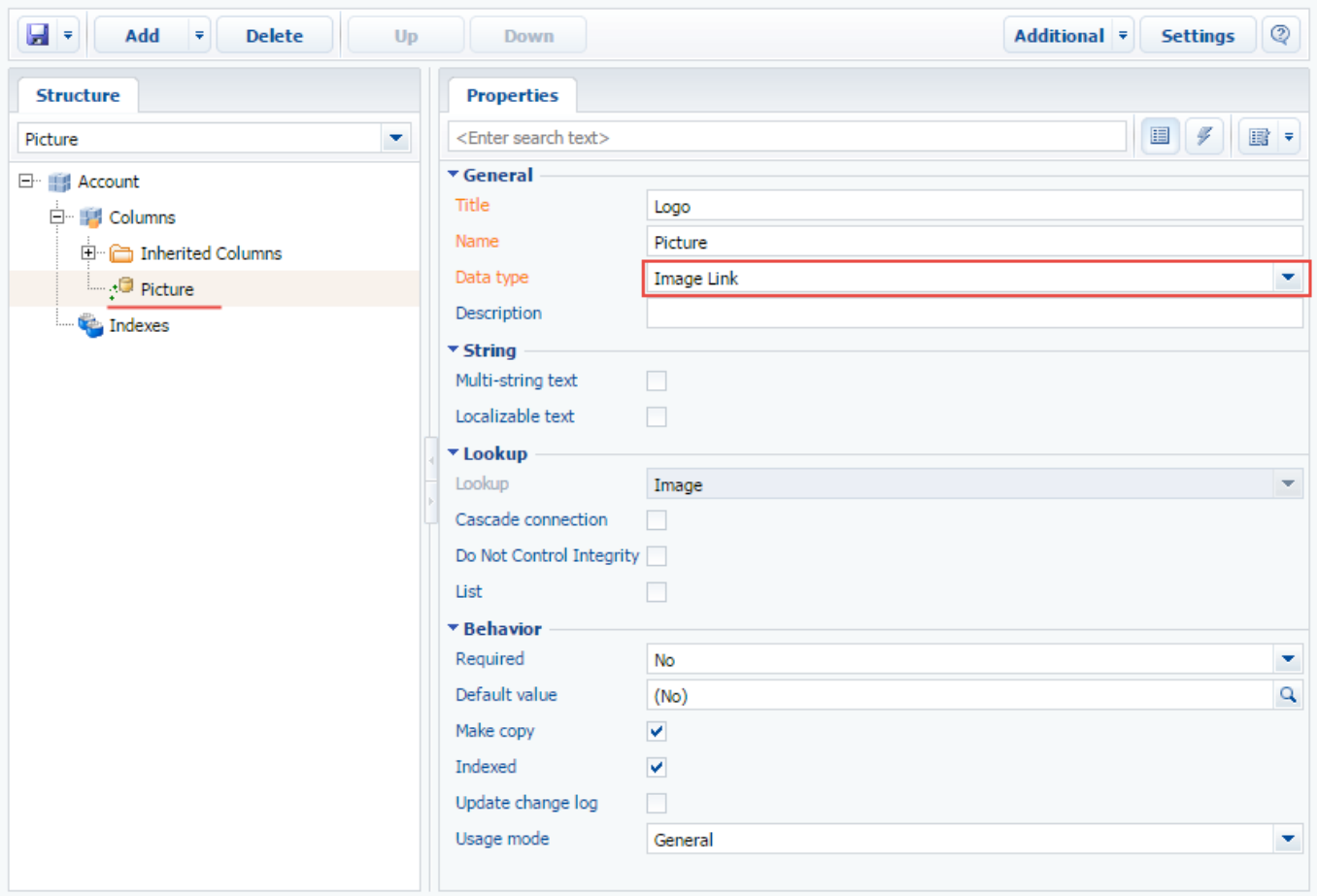


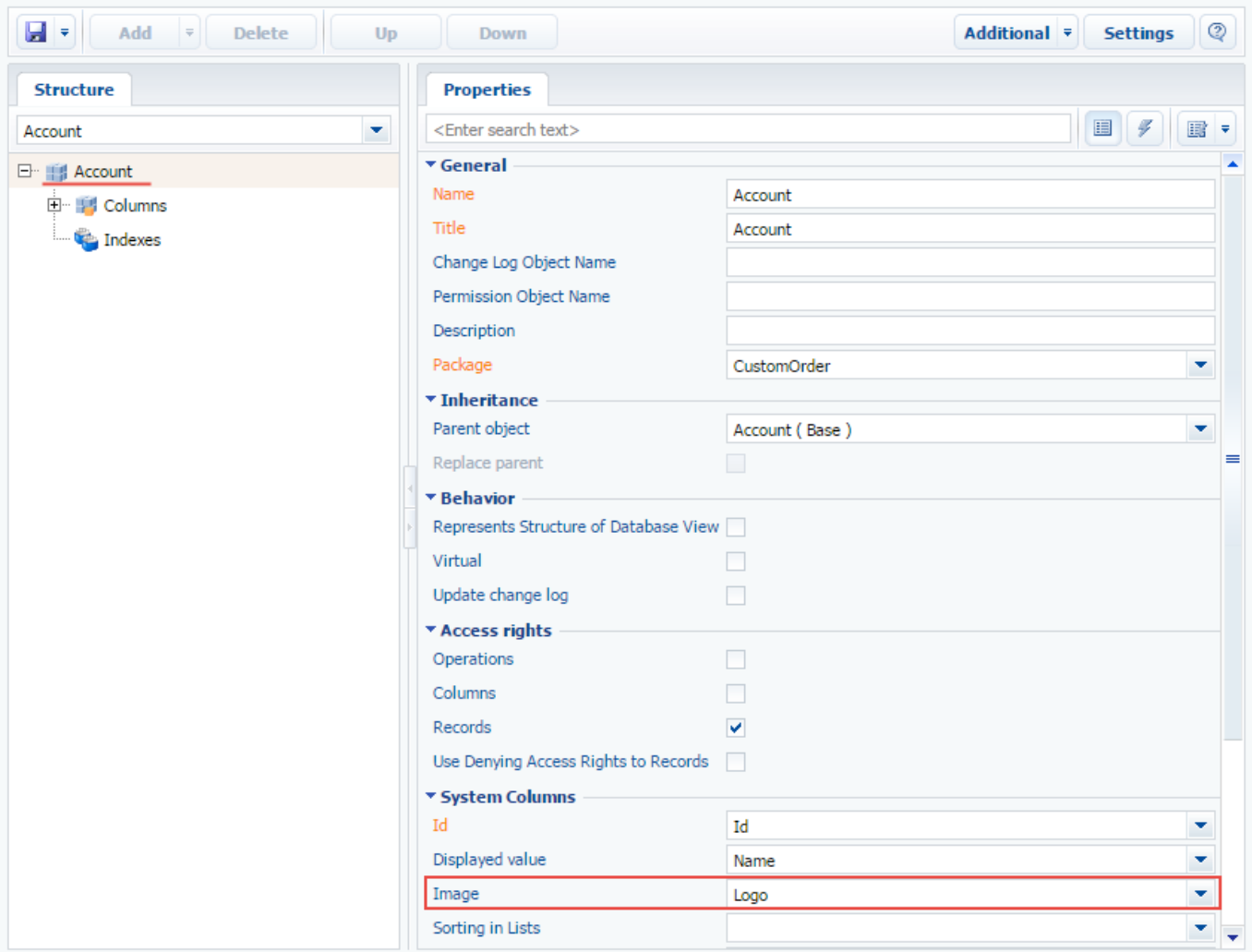
Fill in properties of the new object by specifying [Account] as the parent object.

2. Add a new [Logo] column to the replacing object

To create a column, the type of data, specifically, "Image Link", must be specified (Fig. 2).

Fig. 2. – Adding the user column to the replacing object





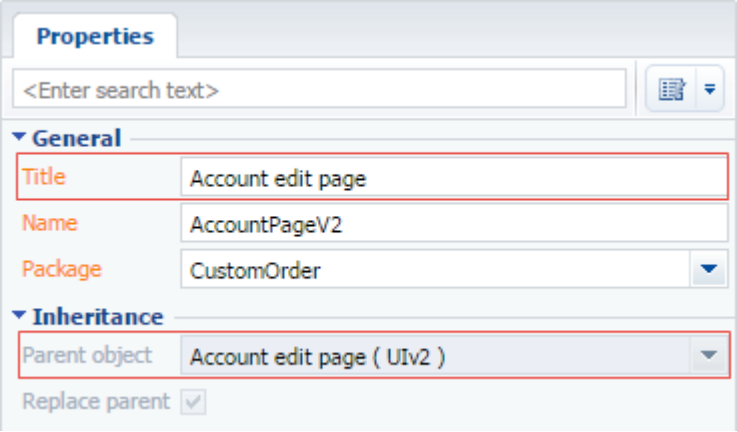
3. Publish the created replacingobject

4. Create a replacing client module of the account edit page

A replacingclient module must be created and [Account edit page] must be specified as the parent object (Fig. 3).

The procedure for creating the replacing page is described in the article **Creating a custom client module schema**.

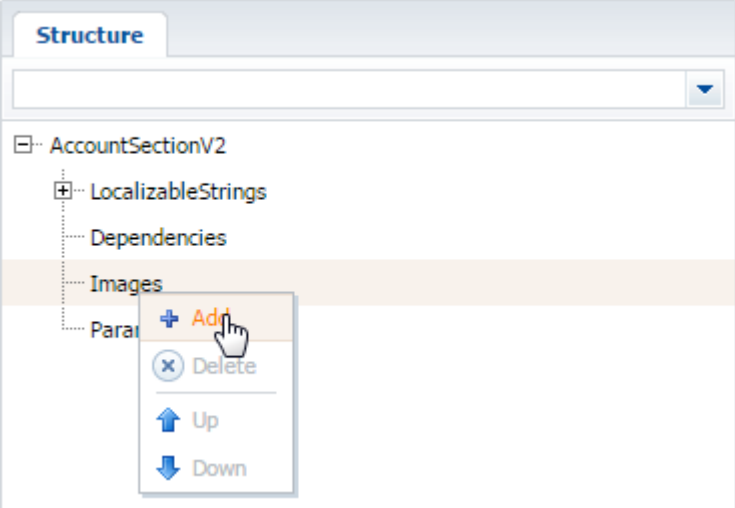
Fig. 3. — Properties of the replacing edit page



5. Add a default value to the [Images] resources of the edit page schema

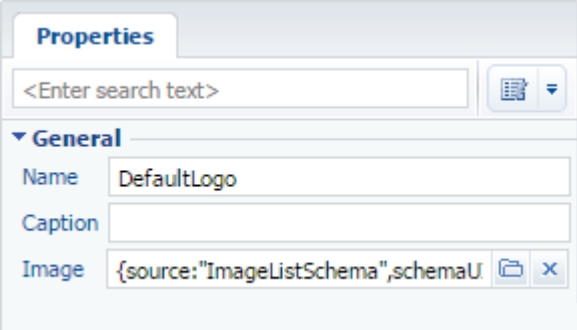
For this purpose, select [Add] by right-clicking the [Images] structure node (Fig. 4).

Fig. 4. — Adding a default image to the schema resources



Fill in properties for the created image as shown in Figure 5. Select a file with the default image from the file system in the [Image] field.

Fig. 5. — Properties of the schema resource



6. Set up the visual display of the field with the logo on the edit page

The field with the logo must be located in the upper part of the account edit page. Fields in the base implementation of this schema are located so that a logo may interfere with the page interface. In addition to a new field location, the location of existing fields must be modified.

For this purpose, describe the configuration object of the field with the logo with required parameters in the diff property of the view model and describe modifications for the fields present in the upper part of the page: [Name], [Owner], [Type].

The field with the image is added to the page using an auxiliary wrapper container with the ["image-edit-container"] class.

7. Add the implementation of the following methods to the methods collection of the page view model:

- getAccountImage — the method for receiving an image by a link;
- getAccountDefaultImage — the method for setting a default value;
- onPhotoChange — the method called when an image is modified;
- onPhotoUploaded — the method for storing a link to a modified image in the object column.

Below is the source code for the replacingschema of the edit page:

```
define("AccountPageV2", ["AccountPageV2Resources", "ConfigurationConstants"],
    function (resources, ConfigurationConstants) {
        return {
```

```
            // Name of the edit page object schema.
            entitySchemaName: "Account",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            // Methods collection of the edit page view model.
            methods: {
                // Returns url of the account login by the link saved in the object.
                getAccountImage: function () {
                    // A link to the image from the object column is received.
                    var imageColumnValue = this.get("Picture");
                    // if the link is set, url of the image file is returned.
                    // If not, the method that returns the default image is called.
                    if (imageColumnValue) {
                        return this.getSchemaImageUrl(imageColumnValue);
                    }
                    return this.getAccountDefaultImage();
                },
                // Returns url of the default account logo.
                getAccountDefaultImage: function () {
                    return
this.Terrasoft.ImageUrlBuilder.getUrl(this.get("Resources.Images.DefaultLogo"));
                },
                // Process the account logo modification. A file with an image is
accepted as the parameter.
                onPhotoChange: function (photo) {
                    if (!photo) {
                        this.set("Picture", null);
                        return;
                    }
                    // A file is uploaded to the database. Upon the uploading
completion, the method
                    // storing the link to the modified image in the [Logo] column
                    //of the [Account] object is called.
                    this.Terrasoft.ImageApi.upload({
                        file: photo,
                        onComplete: this.onPhotoUploaded,
                        onError: this.Terrasoft.emptyFn,
                        scope: this
                    });
                },
                // Method storing a link to the modified image. The Id of the saved
file
                // is returned from the database as a parameter.
                onPhotoUploaded: function (imageId) {
                    var imageData = {
                        value: imageId,
                        displayValue: "Image"
                    };
                    // A link to the image is assigned to the field with the image.
                    this.set("Picture", imageData);
                }
            },
            //
            diff: /**SCHEMA_DIFF*/[
                // Wrapper container where the account logo will be located.
                {
                    "operation": "insert",
                    "parentName": "Header",
                    "propertyName": "items",
                    "name": "PhotoContainer",
                    "values": {
                        // The item type is "container".
                        "itemType": Terrasoft.ViewItemType.CONTAINER,
```

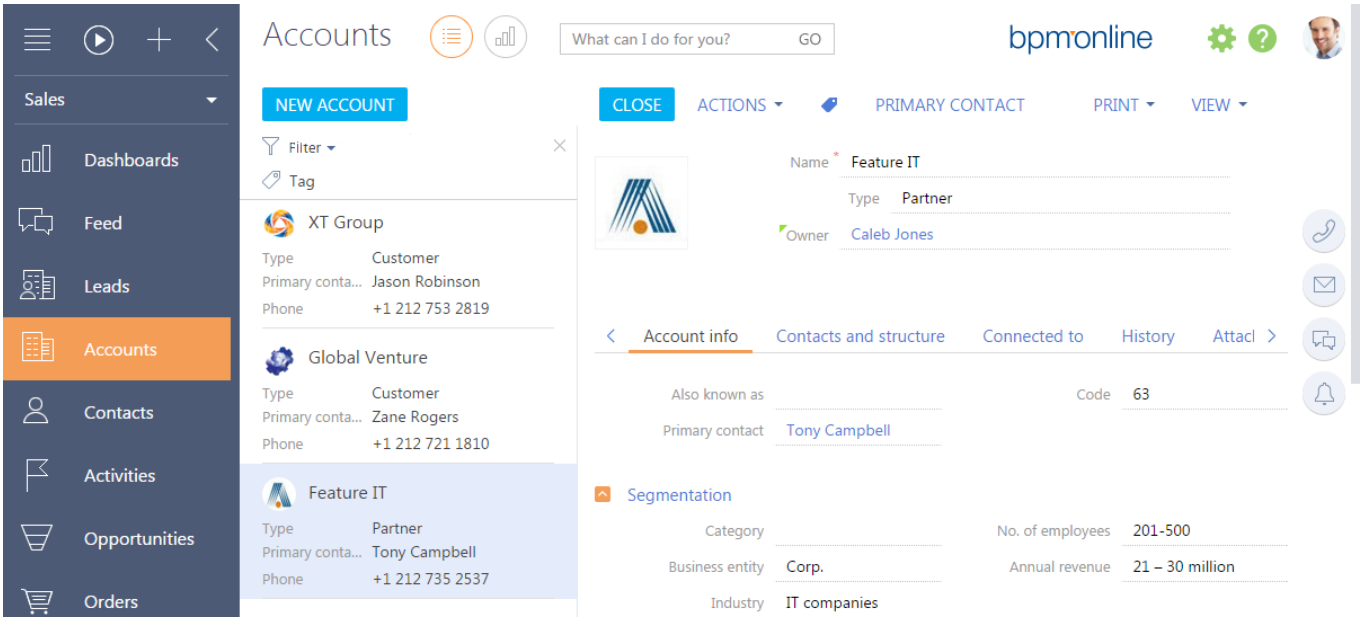
```
                        "wrapClass": ["image-edit-container"],
                        "layout": { "column": 0, "row": 0, "rowSpan": 4, "colSpan": 4
},
                        "items": []
                    }
                },
                // The [Picture] field is the field with the account logo.
                {
                    "operation": "insert",
                    "parentName": "PhotoContainer",
                    "propertyName": "items",
                    "name": "Picture",
                    "values": {
                        // Name of the method receiving the image by a link.
                        "getSrcMethod": "getAccountImage",
                        // Handler method called if the image is modified.
                        "onPhotoChange": "onPhotoChange",
                        // Property determing a possibility to edit (modify, delete)
an image.
                        "readonly": false,
                        // Image to be displayed by default.
                        "defaultImage":
Terrasoft.ImageUrlBuilder.getUrl(resources.localizableImages.DefaultLogo),
                        // Control item generator.
                        "generator":
"ImageCustomGeneratorV2.generateCustomImageControl"
                    }
                },
                // Modification of the [Name] field location.
                {
                    // To modify the existing field location in the replacing schema,
                    // the merge operation is used.
                    "operation": "merge",
                    "name": "Name",
                    "parentName": "Header",
                    "propertyName": "items",
                    "values": {
                        "bindTo": "Name",
                        "layout": {
                            "column": 4,
                            "row": 0,
                            "colSpan": 18
                        }
                    }
                },
                // Modification of the [Owner] field location.
                {
                    // To modify the existing field location in the replacing schema,
                    // the merge operation is used.
                    "operation": "merge",
                    "name": "Owner",
                    "parentName": "Header",
                    "propertyName": "items",
                    "values": {
                        "bindTo": "Owner",
                        "layout": {
                            "column": 4,
                            "row": 2,
                            "colSpan": 18
                        }
                    }
                },
```

```
                // Modification of the [Type] field location.
                {
                    // To modify the existing field location in the replacing schema,
                    // the merge operation is used.
                    "operation": "merge",
                    "name": "Type",
                    "parentName": "Header",
                    "propertyName": "items",
                    "values": {
                        "bindTo": "Type",
                        "layout": {
                            "column": 4,
                            "row": 1,
                            "colSpan": 18
                        },
                        "contentType": Terrasoft.ContentType.ENUM
                    }
                },
            ]/**SCHEMA_DIFF*/
        };
    });
```

8. Save the page replacing schema

When the schema is saved and the system web-page is updated, the logo icon will appear on the account edit page (Fig. 6).

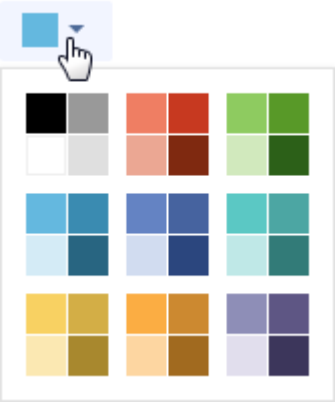
Fig. 6. – Demonstrating the case implementation result



How to add the color select button to the edit page

Adding one of the control items that are unique to bpm'online (the color select button) is covered below (Fig. 1).

Fig. 1. – Color select button



Algorithm of adding the color select button to the object edit page:

1. Add a column with the data type to the object [String (50 symbols)] which will store data on the selected color.
2. Add the description of the item with the [COLOR_BUTTON] type to the diff array. Set the binding to the column added in the previous step for the value property of this item.
3. Add the caption for the button, if required, using the LABEL control item.

Below is the example of adding the color select button to the product page.

Case implementation algorithm

1. Create the "Product" replacingobject and add the [Color] column to it

The creation of the replacingobject and column addition are detailed in the article **Adding a new field to the edit page**.

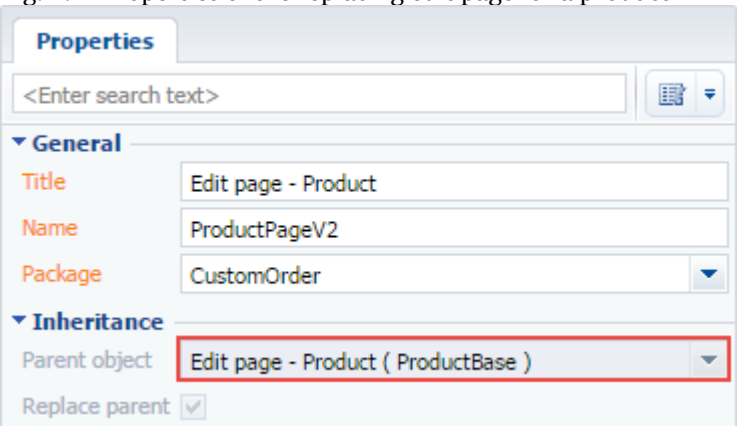
The type of data [String (50 symbols)] must be selected for the added column.

2. Create a replacing edit page for a product in a custom package

A replacing client module must be created and [Edit page - Product] must be specified as the parent object in it (Fig. 2).

The procedure for creating the replacingpage is described in the article **Creating a custom client module schema**.

Fig. 2. – Properties of the replacing edit page for a product

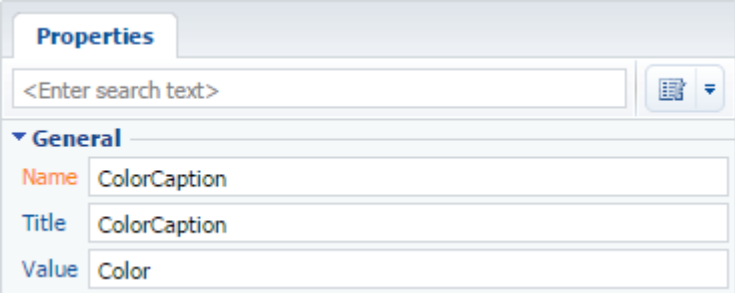


3. Add a string with the field title to the localizable strings collection of the edit replacing page schema

For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure node.

Fill in properties for the created string as shown in Figure 3.

Fig. 3. — Properties of a custom localized string



4. Add the following configuration objects to the diff array:

- Container where the color select button will be located. Specify the type of the control item – Terrasoft.ViewItemType.GRID_LAYOUT.
- Caption for the color select button. Specify the type of the control item – Terrasoft.ViewItemType.LABEL and the ColorCaption localizable string for the caption property.
- Directly the color select button. Specify the type of the control item — Terrasoft.ViewItemType.COLOR_BUTTON and binding of the control item value to the [Color] column of the view model.

Below is the source code for the edit page schema.

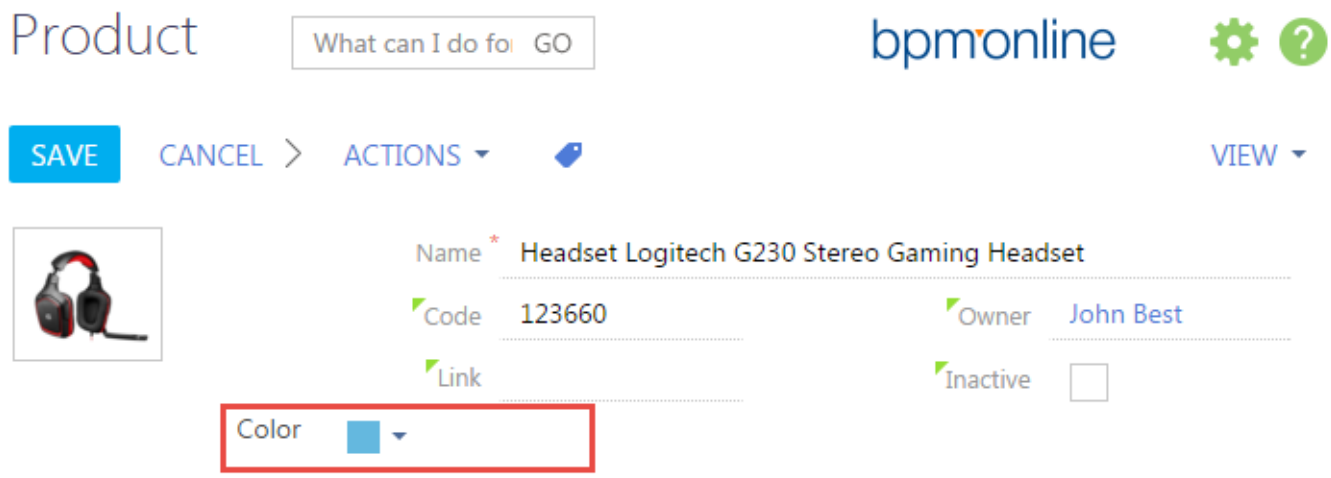
```
define("ProductPageV2", [],
    function () {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Product",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            diff: /**SCHEMA_DIFF*/[
                // Container for locating the color select button and its caption.
                {
                    "operation": "insert",
                    // Name of the parent container.
                    "parentName": "ProductGeneralInfoBlock",
                    "propertyName": "items",
                    // Name of the control item.
                    "name": "ColorBlock",
                    // Property of the control item.
                    "values": {
                        "itemType": this.Terrasoft.ViewItemType.GRID_LAYOUT,
                        "layout": { "column": 1, "row": 2, "colSpan": 24 },
                        "items": []
                    }
                },
                // Caption for the color select button.
                {
                    "operation": "insert",
                    // Name of the parent container.
                    "parentName": "ColorBlock",
                    // Name of the control item.
                    "name": "ColorLabel", "propertyName": "items",
                    // Property of the control item.
                    "values": {
                        "itemType": this.Terrasoft.ViewItemType.LABEL,
                        // Link to the localizable string with the caption text.
                        "caption": { "bindTo": "Reources.Strings.ColorCaption" },
                        "layout": { "column": 0, "row": 0, "colSpan": 2 }
                    }
```

```
                },
                // Color select button.
                {
                    "operation": "insert",
                    // Name of the parent container.
                    "parentName": "ColorBlock",
                    "propertyName": "items",
                    // Name of the control item.
                    "name": "ColorButton",
                    // Property of the control item.
                    "values": {
                        "itemType": this.Terrasoft.ViewItemType.COLOR_BUTTON,
                        // Binding of the control item value to the view model
column.
                        "value": { "bindTo": "Color" },
                        "layout": { "column": 2, "row": 0, "colSpan": 20 }
                    }
                }
            ],/**SCHEMA_DIFF*/
            methods: {}
        };
    });
```

5. Save the page replacing schema

When the schema is saved and the system web-page is updated, the color select button with the [Color] caption will appear on the product edit page (Fig. 4).

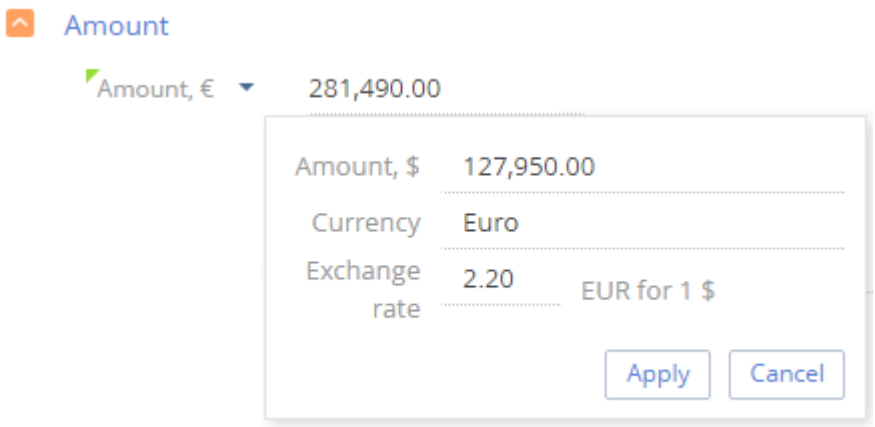
Fig. 4. – Result demonstration



How to add multi-currency field

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



How to add a multi-currency field on an edit page

1. Add 4 fields to the view model:
   - [Currency]
   - [Exchange rate]
   - [Amount]
   - [Amount in base currency]

   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.
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 **How to add a computed field** article.

Below is a multi-currency field implementation example.

Case description

Add a multi-currency [Amount] field to the project edit page.

Case implementation

1. Create replacing Project object and add [Currency], [Amount] and [PrimaryAmount] columns

For more information on creating replacing objects and adding columns, please refer to the "**Adding a new field to the edit page**" article.

Column properties are available on Fig. 2, 3 and 4,

Fig. 2. [Currency] column properties

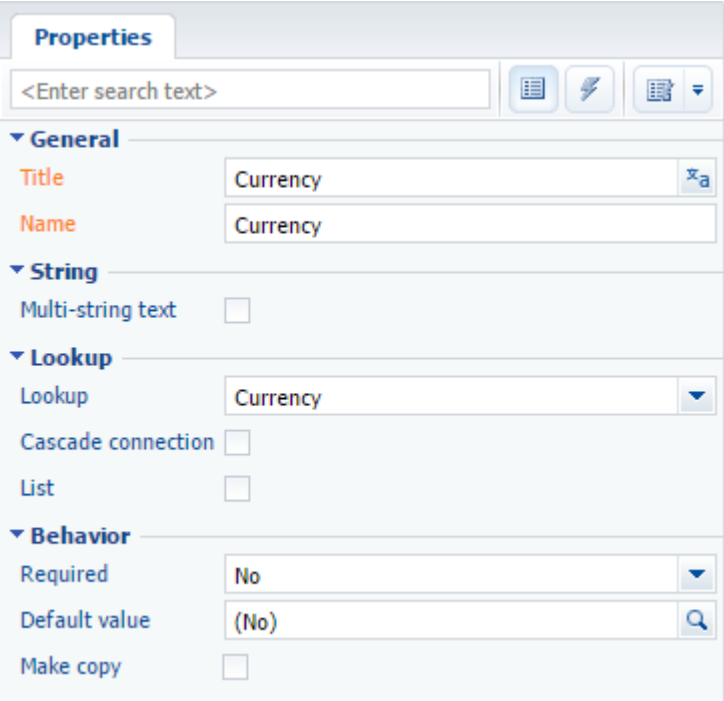


Fig. 3. [Amount] column properties

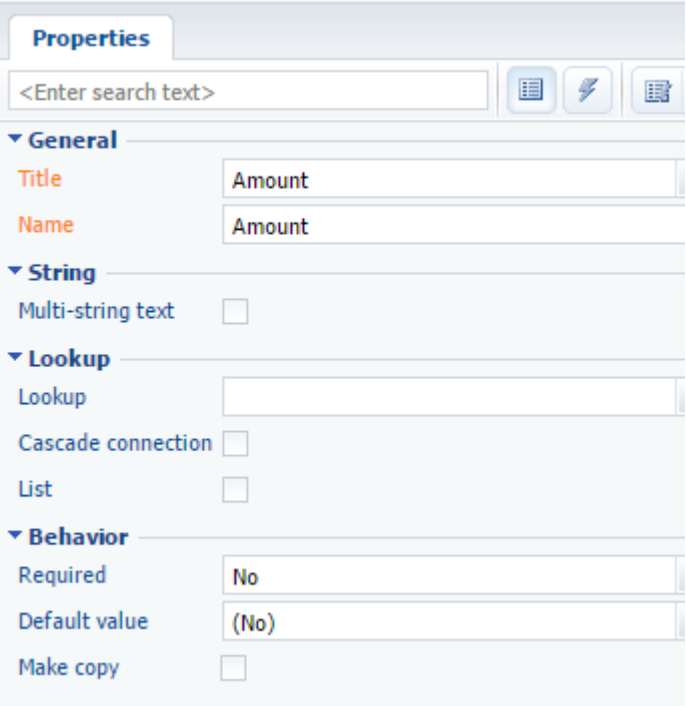
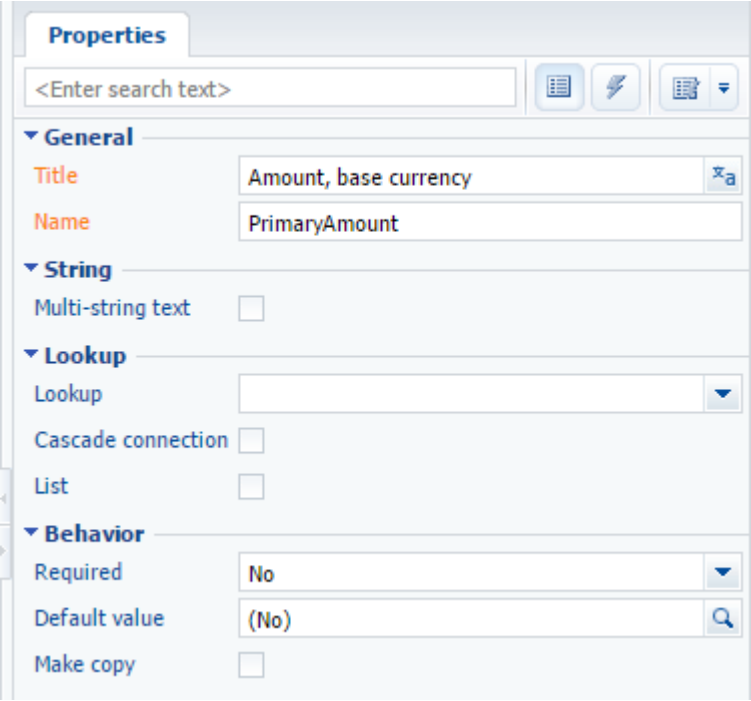


Fig. 4. [PrimaryAmount] column properties

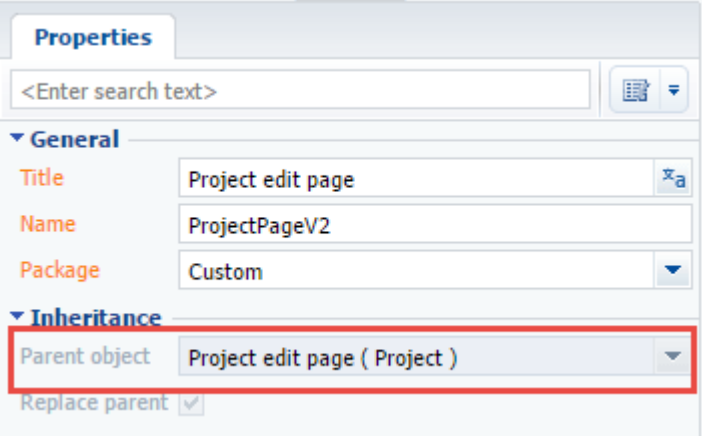


2. Create a replacing edit page for the [Projects] section in the custom package

Create a replacing client module and specify [ProjectPageV2] as the parent object (Fig. 5).

For more information on creating replacing pages, please refer to the "**Creating a custom client module schema**" article.

Fig. 5. Properties of the replacing project edit page



3. The [Currency rate] field must be declared directly in the attributes value of the edit page schema

```
"CurrencyRate": {
    dataValueType: Terrasoft.DataValueType.FLOAT,
}
```

4. Specify the following modules as dependencies when declaring view model class:

- MoneyModule,
- MultiCurrencyEdit,
- MultiCurrencyEditUtilities.

```
define("ProjectPageV2", ["MoneyModule", "MultiCurrencyEdit",
"MultiCurrencyEditUtilities"],
```

```
    function(MoneyModule, MultiCurrencyEdit, MultiCurrencyEditUtilities) {
```

5. Connect the Terrasoft.MultiCurrencyEditUtilities mixin to the view model

```
mixins: {
    /**
    * Multi-currency management mixin in the record edit page.
    */
    MultiCurrencyEditUtilities: "Terrasoft.MultiCurrencyEditUtilities"
}
```

Additionally, initialize the mixin in the overridden init method:

```
// Overriding the Terrasoft.BasePageV2.init base method
init: function() {
    // Initializing the multi-currency management mixin.
    this.mixins.MultiCurrencyEditUtilities.init.call(this);
    // Calling parent implementation of the init method.
    this.callParent(arguments);
},
```

6. Add a configuration object with multi-currency field settings to the diff array

```
diff: /**SCHEMA_DIFF*/[
    {
        "operation": "insert",
        // Parent container name.
        "parentName": "Header",
        "propertyName": "items",
        // Control element name.
        "name": "Amount",
        "values": {
            // View model column name in which the binding is done.
            "bindTo": "Amount",
            // Item location in container.
            "layout": {"column": 0, "row": 2, "colSpan": 12},
            // Name of the column, which contains amount in base currency.
            "primaryAmount": "PrimaryAmount",
            // Name of the column, which contains the currency of the amount.
            "currency": "Currency",
            // Name of the column, which contains the currency rate.
            "rate": "CurrencyRate",
            // The property that determines if the base currency amount field is
editable.
            "primaryAmountEnabled": false,
            // Control element view generator.
            "generator": "MultiCurrencyEditViewGenerator.generate"
        }
    }
],/**SCHEMA_DIFF*/
```

7. Implement the amount recalculation logic

In the attributes property, add dependencies for the [CurrencyRate] and [Amount] attributes. Add handler methods for these dependencies to the methods collection.

Below is the complete source code of the edit page.

```
// 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: {
                // View model column name.
                "CurrencyRate": {
                    // Data type of the view model column.
                    dataValueType: Terrasoft.DataValueType.FLOAT,
                    // Array of configuration objects that determine the dependencies
of the [CurrencyRate] column.
                    dependencies: [
                        {
                            // The [CurrencyRate] column value depends on the
[Currency] column value.
                            columns: ["Currency"],
                            // Handler method that is called when the value in the
[Currency] column changes.
                            methodName: "setCurrencyRate"
                        }
                    ]
                },
                // View model column name.
                "Amount": {
                    // View model column data type.
                    dataValueType: Terrasoft.DataValueType.FLOAT,
                    // Array of configuration objects that determine dependencies of
the [Amount] column.
                    dependencies: [
                        {
                            // Value in the [Amount] column.
                            columns: ["Amount"],
                            // Handler method, which is called when the value in the
[Amount] column changes.
                            methodName: "recalculatePrimaryAmount"
                        }
                    ]
                }
            },
            // Collection of view model mixins.
            mixins: {
                // Multi-currency management mixin on the edit page.
                MultiCurrencyEditUtilities: "Terrasoft.MultiCurrencyEditUtilities"
            },
            // Collection of page view model methods.
            methods: {
                // Overriding base method Terrasoft.BasePageV2.init.
                init: function () {
                    // Initialize multi-currency management mixin.
                    this.mixins.MultiCurrencyEditUtilities.init.call(this);
                    // Calling parent implementation of the init method.
                    this.callParent(arguments);
                },
                // Handler method, which is called on changing the currency.
                setCurrencyRate: function () {
                    // Calling the LoadCurrencyRate function of the MoneyModule,
which calculates
                    // currency exchange rate. [Currency] and [CurrencyRate] are
passed as parameters.
                    MoneyModule.LoadCurrencyRate.call(this, "Currency",
"CurrencyRate", this.get("StartDate"),
                        function () {
```

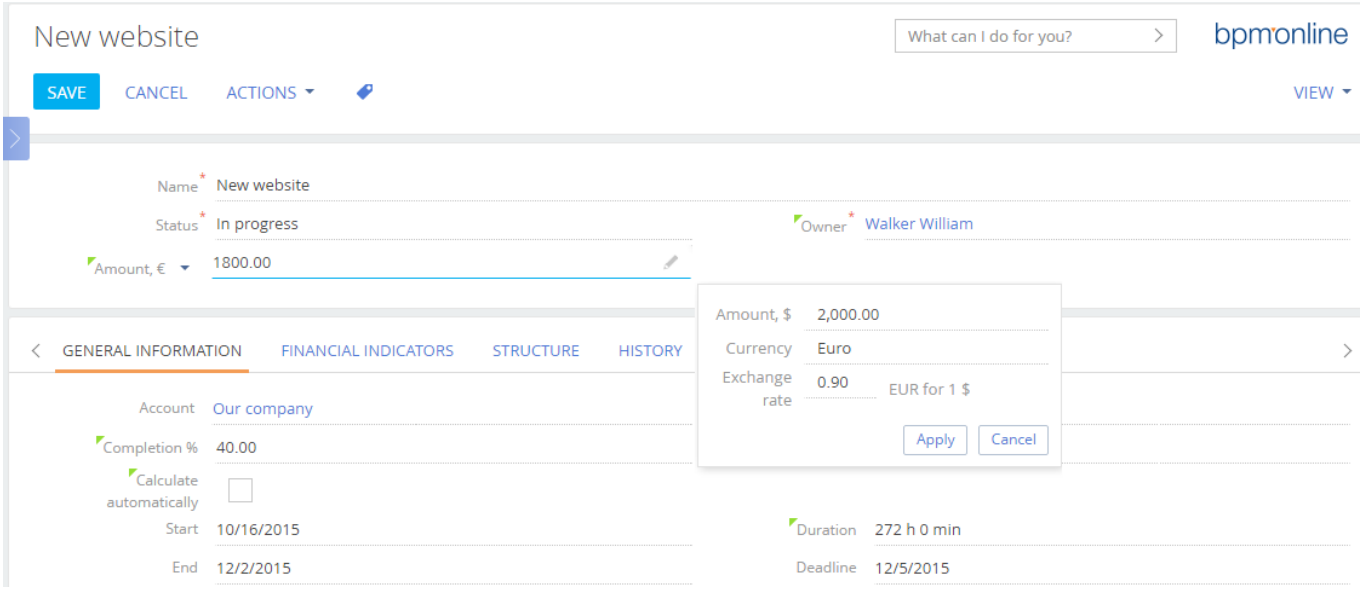
```
                            var division = this.getCurrencyDivision();
                            MoneyModule.RecalcCurrencyValue.call(this,
"CurrencyRate", "Amount", "PrimaryAmount",
                                 division);
                        }
                    );
                },
                // Auxiliary method, which returns delimiting coefficient for a
currency.
                getCurrencyDivision: function () {
                    var currency = this.get("Currency");
                    return currency && currency.Division;
                },
                // Handler method, which is called if the amount is modified and
conducts recalculation of amount
                // in base currency.
                recalculatePrimaryAmount: function () {
                    // Determining the delimiting coefficient for the current
currency.
                    var division = this.getCurrencyDivision();
                    // Calling the RecalcBaseValue function of the MoneyModule, which
calculates
                    // amount for the [Amount in base currency] field. Column names
for
                    // [CurrencyRate], [Amount], [PrimaryAmount], as well as
delimiting coefficient are passed as parameters.
                    MoneyModule.RecalcBaseValue.call(this, "CurrencyRate", "Amount",
"PrimaryAmount",
                        division);
                }
            },
            // Multi-currency field visualization setup.
            diff: /**SCHEMA_DIFF*/[
                // Metadata to add the [Amount] field.
                {
                    "operation": "insert",
                    // Parent container name.
                    "parentName": "Header",
                    "propertyName": "items",
                    // Control element name.
                    "name": "Amount",
                    "values": {
                        // View model column name, to which the binding is made.
                        "bindTo": "Amount",
                        // Element position within container.
                        "layout": { "column": 0, "row": 2, "colSpan": 12 },
                        // Name of the column that contains amount in base currency.
                        "primaryAmount": "PrimaryAmount",
                        // Name of the column that contains the currency.
                        "currency": "Currency",
                        // Name of the column that contains exchange rate.
                        "rate": "CurrencyRate",
                        // Property that determines availability of the base currency
amount field.
                        "primaryAmountEnabled": false,
                        // Control element view generator.
                        "generator": "MultiCurrencyEditViewGenerator.generate"
                    }
                }
            ],/**SCHEMA_DIFF*/
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/
        };
```

```
});
```

8. Save page replacing schema

After saving the schema and updating the web page, a new multi-currency edit field [Amount] will appear on the page (Fig. 6).

Fig. 6. Case results



How to add a computed field

The term *computed field* means a page management item whose value is formed depending on the status or values of other items of this page.

The possibility to set computed fields in bpm'online is implemented at a level of the page view model columns. Computed fields operate on the basis of the primary client mechanism in bpm'online, namely, subscribing to events of changes to the page view model properties. Bpm'online contains the implemented mechanism for setting dependencies of the view model column, which enables implicit subscribing to the model change events.

A configuration object may be set for any column of the view model. The object must contain names of columns which change results in the change of the column value and the name of the event handler method.

General algorithm of adding a computed field

1. Add a column for storing the computed field value to the page object.
2. Set the column dependency in the page view model by specifying names of columns it depends on and set the handler name.
3. Add the handler method implementation to the collection of the model methods.
4. Set the visualization of the calculated field in the diff property of the view model.

Setting dependencies of a computed field

The name of the column, for which a dependency is set, must be added to the attributes property of the view model.

The dependencies property representing an array of configuration objects each of which contains the properties specified below must be declared for this column:

- columns — the array of column names, on which the current column value depends;
- methodName — the handler method name.

If the value of just one of the above columns changes in the view model, the handler method of the event whose name is specified in the methodName property will be called.

The handler method implementation must be added to the collection of the view model methods.

Examples of adding a computed field to the edit page

Case description

Add the [Payment balance] to the order edit page. The difference between the order amount from the [Total] field and payment amount will be displayed here.

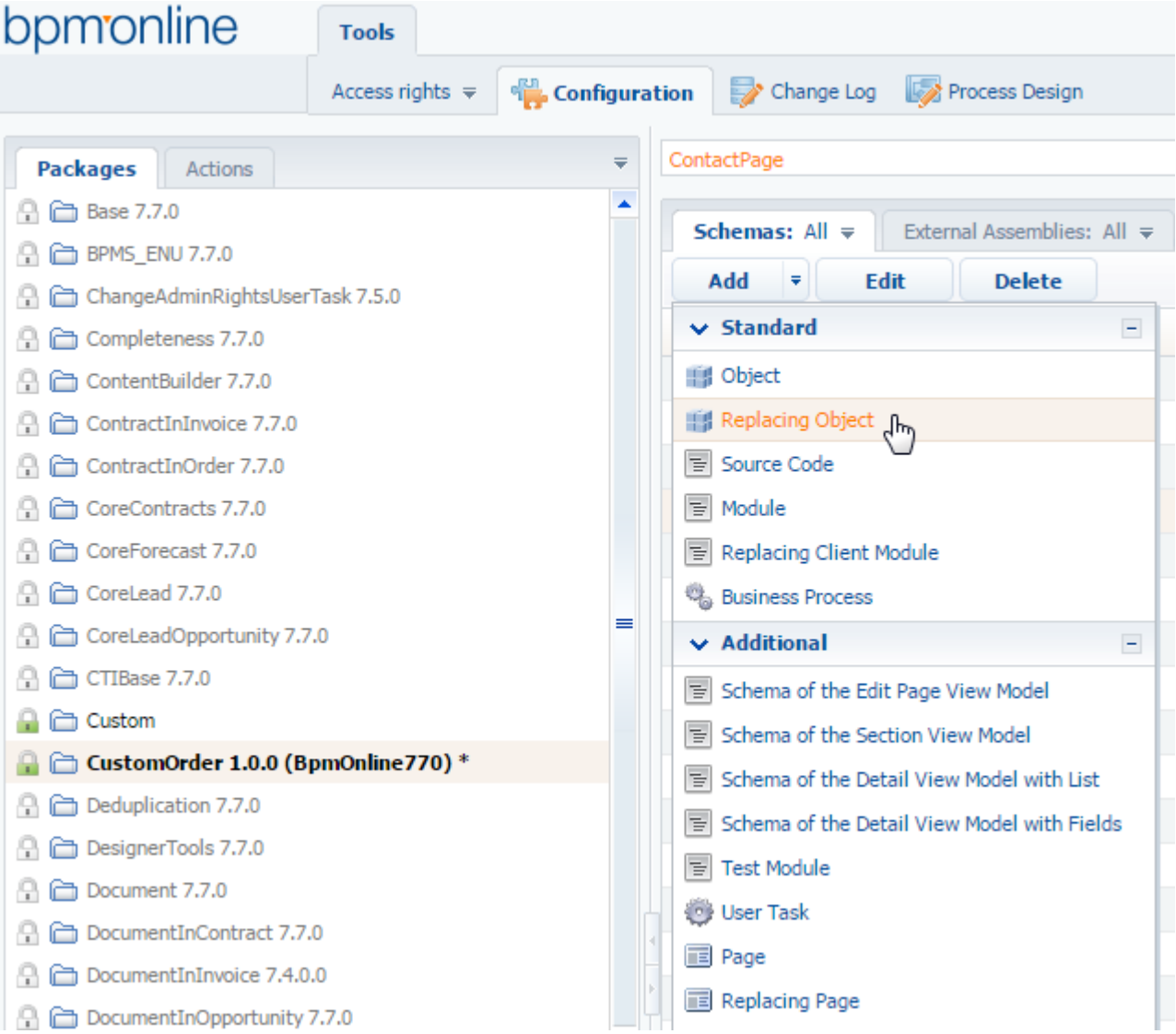
> NOTE
>
> Adding a field to the edit page may be implemented by two ways – using the Section Wizard and manually.
>
> The process of adding a field to the edit page is described in the article **Adding a new field to the edit page**.

Case implementation algorithm

1. Create the [Order] replacement object and add the new [Payment balance] column to it

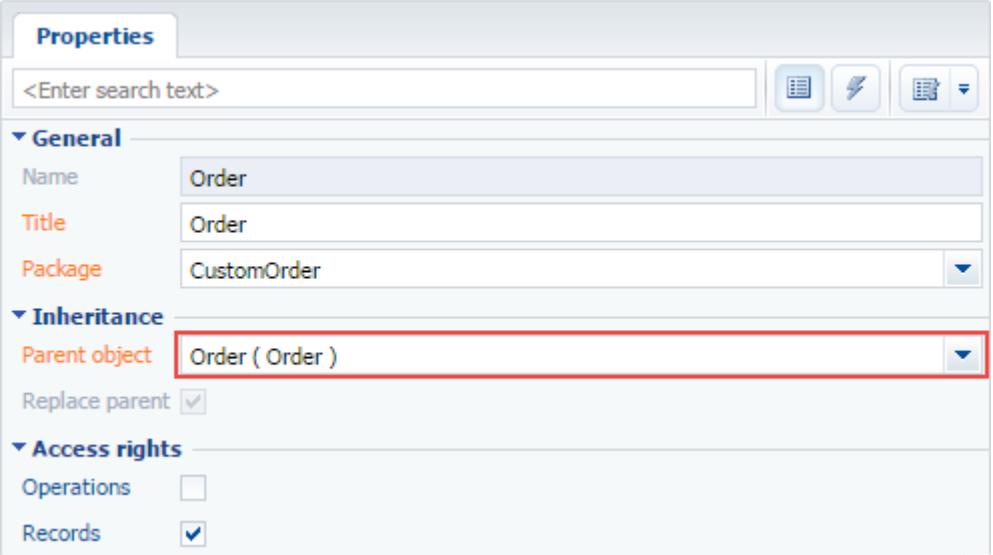
For this purpose, select a user package and run the [Add] > [Replacing Object] menu item (Fig. 1) on the [Schemas] tab.

Fig. 1. – Creating the replacement schema of the object



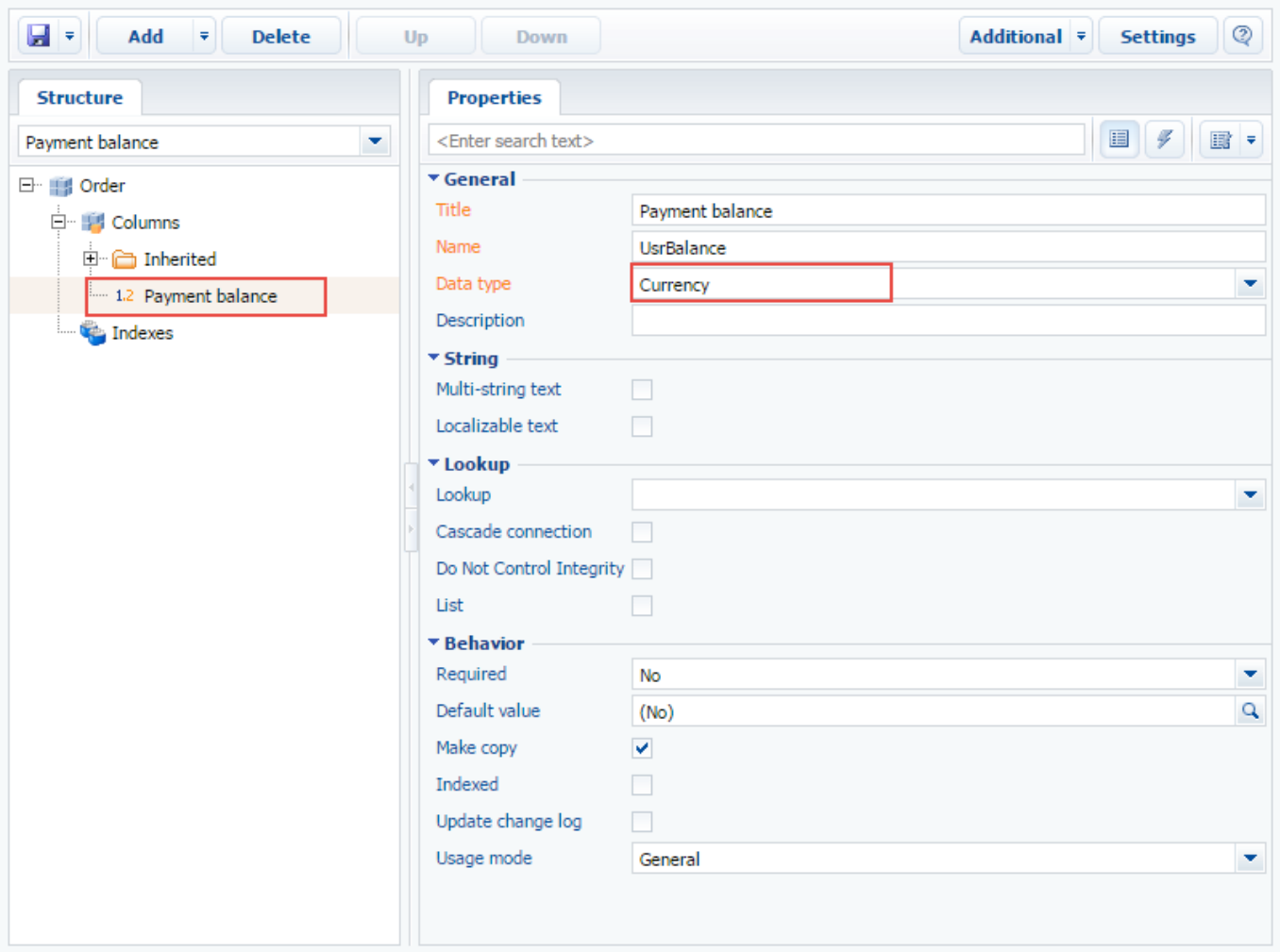
Fill the new object properties by specifying the [Order] object as the parent object (Fig. 2).

Fig. 2. — Properties of the [Order] replacement object



Add the new [Payment balance] column with the Money data type to the replacement object (Fig. 3.).

Fig. 3. – Adding the user column to the replacement object



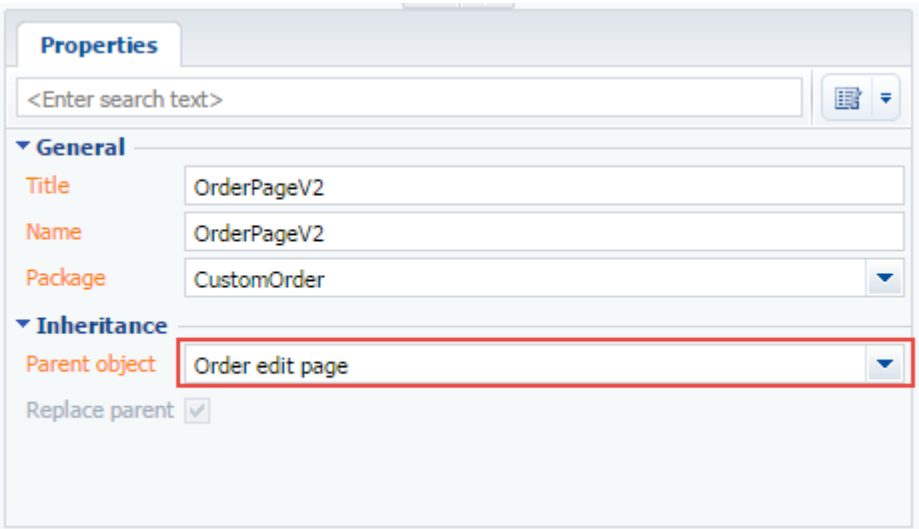
Publish the object.

2. Create the replacing client module of the order edit page

A replacement client module must be created and [Order edit page] must be specified as the parent object (Fig. 4).

The process of creating the replacement page is described in the article **Creating a custom client module schema**.

Fig. 4. — Properties of the replacement edit page



3. Set the visual image of the [Payment balance] field

For this purpose a configuration object with required parameters must be described in the diff property of the view model.

4. Add the [UsrBalance] column to the attributes properties of the view model

Specify the dependence on the [Amount] and [PaymentAmount] columns, as well as the calculateBalance handler method calculating the value for the [UsrBalance] column in the configuration object of the [UsrBalance] column.

5. Add the calculateBalance handler method to the methods collection of the model.

6. Redefine the onEntityInitialized primary virtual method.

This step is optional in the case implementation algorithm. The onEntityInitialized methos is the primary method actuating when the initialization of the edit page object schema is completed. Adding the call of the calculateBalance handler method to this method will ensure the calculation of the payment balance amount when the edit page is opened, but not only when dependencies columsn are changed.

The source code of the module is shown below.

```
define("OrderPageV2", ["OrderConfigurationConstants"],
    function(OrderConfigurationConstants) {
        return {
            // Object schema name of the edit page.
            entitySchemaName: "Order",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            // The attributes property of the view model.
            attributes: {
                // Name of the view model column.
                "UsrBalance": {
                    // Data type of the view model column.
                    dataValueType: Terrasoft.DataValueType.FLOAT,
                    // Array of configuration objects that determine dependencies of
                    // the [UsrBalance] column.
                    dependencies: [
```

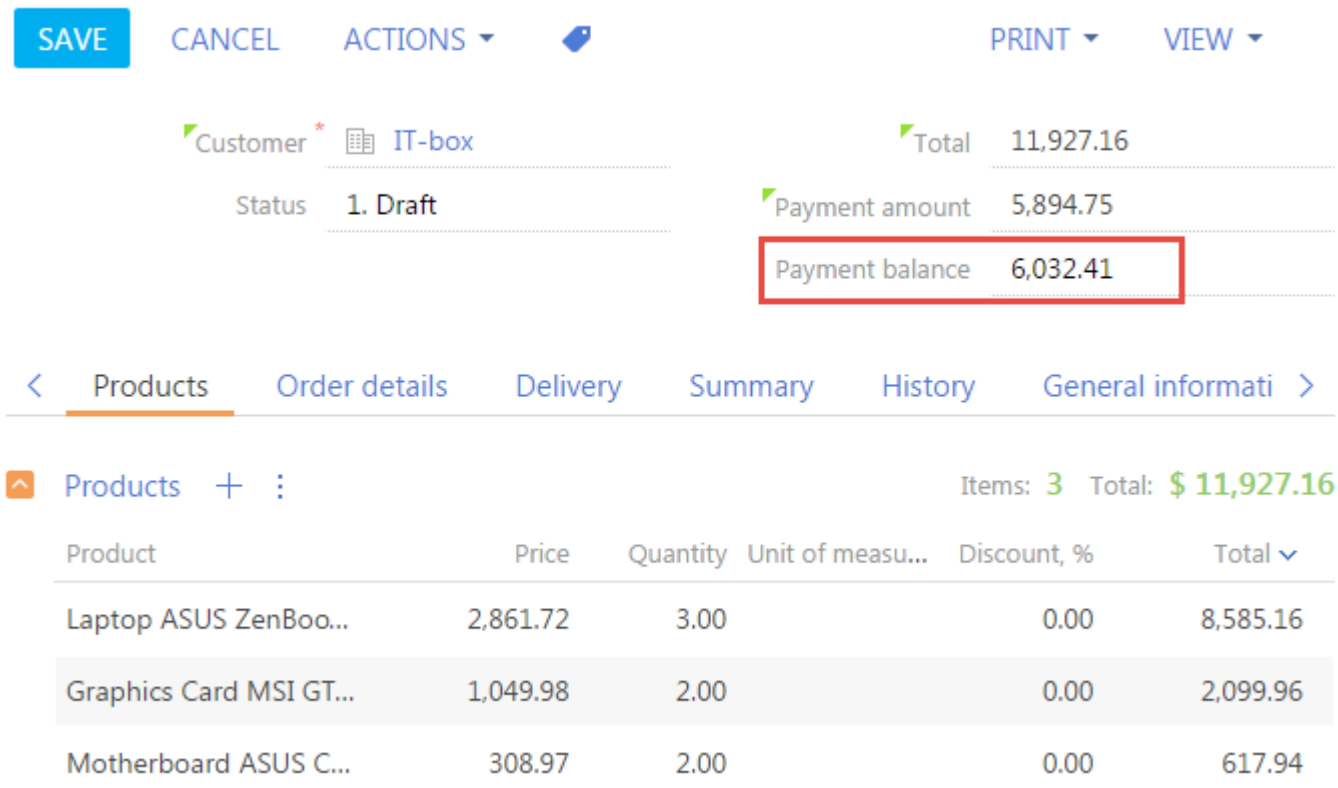
```
                        {
                            // Value of the [UsrBalance] column depends on the values
of the
                            // [Amount] and [PaymentAmount] columns.
                            columns: ["Amount", "PaymentAmount"],
                            // Handler method that is called when the value is
changed
                            // either in the [Amount] or [PaymentAmount] column.
                            methodName: "calculateBalance"
                        }
                    ]
                }
            },
                        // Collection of methods of the edit page view model.
            methods: {
                // Overriding base Terrasoft.BasePageV2.onEntityInitialized method
                // that triggers after initialization of edit page object schema.
                onEntityInitialized: function () {
                    // Parent implementation of the method is called.
                    this.callParent(arguments);
                    // Calling handler method, that calculates the [UsrBalance]
column value.
                    this.calculateBalance();
                },
                // The handler method, that calculates the [UsrBalance] column value.
                calculateBalance: function () {
                    // Check, if the [Amount] and [PaymentAmount] columns are
initialized
                    // at the moment of opening the edit page.
                    // If not, set their values to zero.
                    if (!this.get("Amount")) {
                        this.set("Amount", 0);
                    }
                    if (!this.get("PaymentAmount")) {
                        this.set("PaymentAmount", 0);
                    }
                    // Calculating the margin between the values in the [Amount] and
                    // [PaymentAmount] columns.
                    var result = this.get("Amount") - this.get("PaymentAmount");
                    // The resulting value is written to the [UsrBalance] column.
                    this.set("UsrBalance", result);
                }
            },
            // Visual display of the [Payment balance] field 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*/
        };
    });
```

7. Save the created replacement schema of the page.

When a schema is saved and the web-page of the system is updated, additional [Payment balance] field will appear

on the order oage. The value in tis field will depend on the values in the [Total] and [Payment amount] fields (Fig. 5).

Fig. 5. – Demonstrating the case implementation result



How to set a default value for a field

Bpm'online has the possibility to set default values for control items of the edit page.

A default value may be set by two ways:

1. Set at the level of business object columns.

In most cases, when creating a new object, specific fields of the page must be filled in with certain values. In such cases, these values must be specified in the object designer as default values for the object columns corresponding to these fields.

2. Set in the source code of the edit page.

In some cases, a default value cannot be set using object column properties. For example, this may be a computed value that is calculated using values in other object columns, etc. In this case, the default value may be set using software only.

Both cases of setting a default value are described further below using specific examples.

Example of setting a default value for a field using object column properties.

Case description

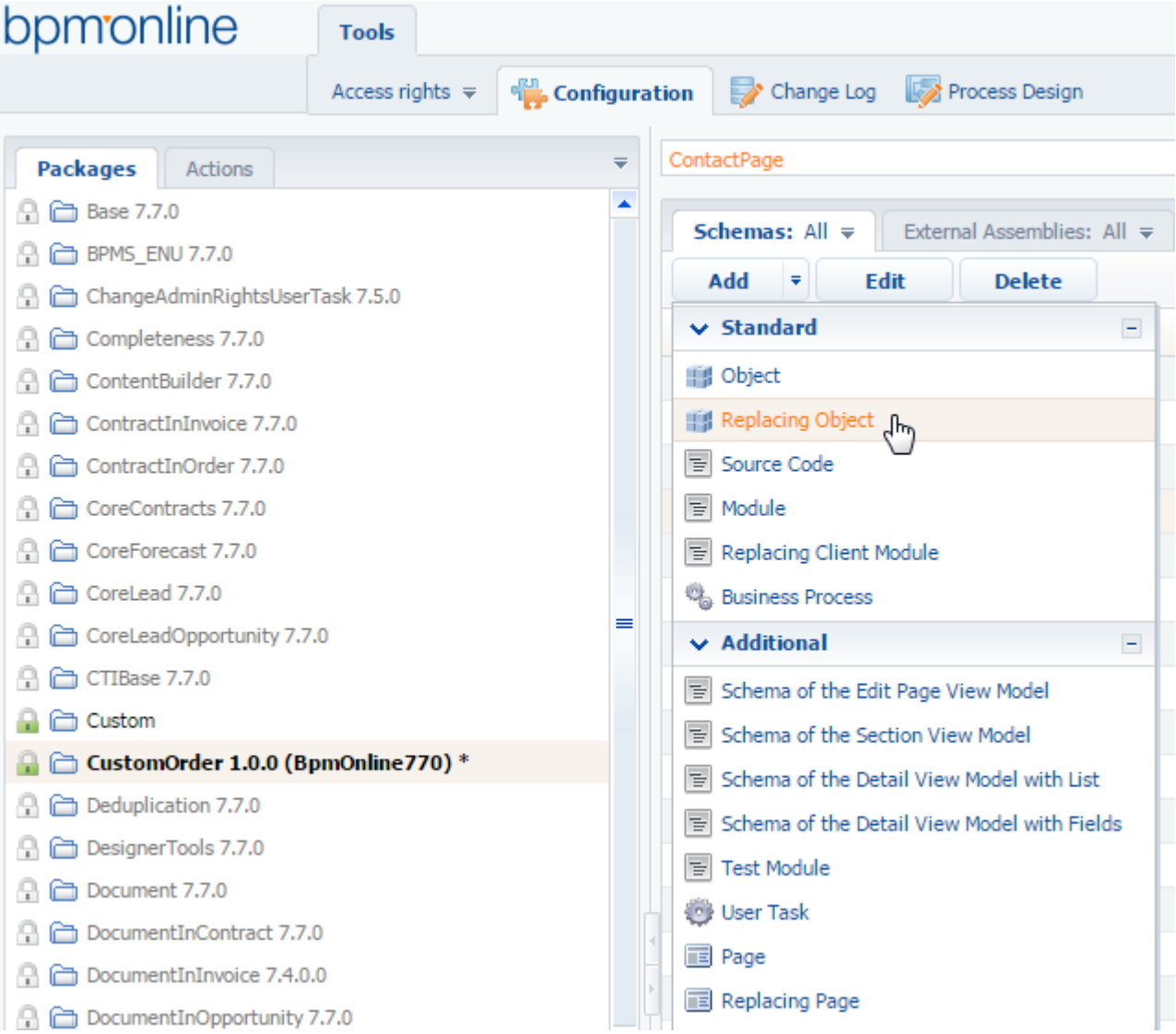
Create a new activity with the [Show in calendar] checkbox selected by default.

Case implementation algorithm

1. Create an [Activity] replacement object in a custom package (or open an existing object)

For this purpose, select a custom package and run the [Add] > [Replacing Object] menu item on the [Schemas] tab (Fig. 1).

Fig. 1. – Creating a replacement object

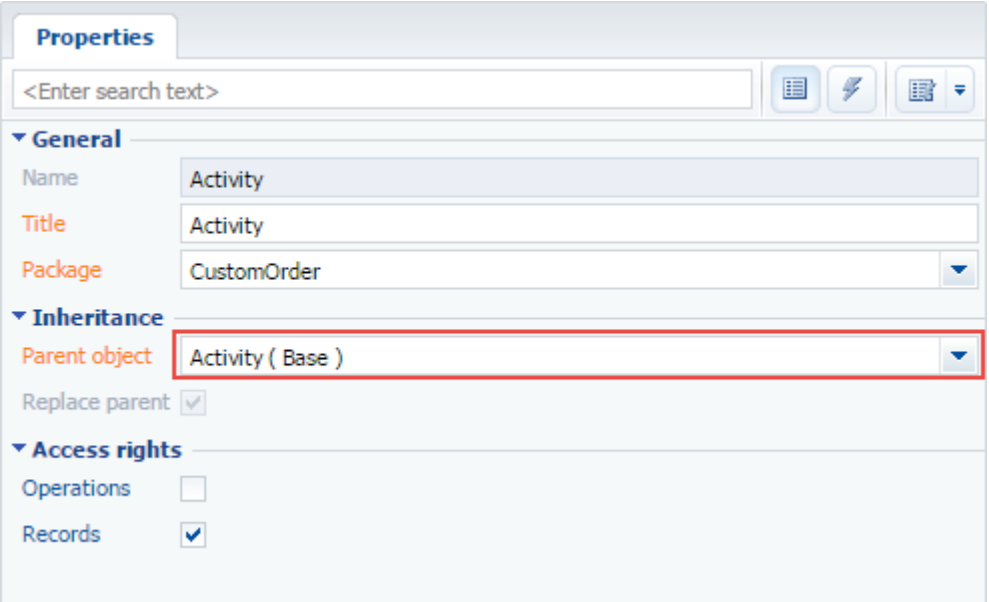


To allow a module to replace a certain object, the base object name to be replaced must be specified in the [Parent Object] field.

This means that to create the [Activity] replacement object, it must be specified as the parent object.

When the selected parent object is confirmed, the remaining fields of properties are filled in automatically (Fig. 2).

Fig. 2. — Properties of the [Activity] replacement object



2. Set a default value for the [Display in Calendar] column

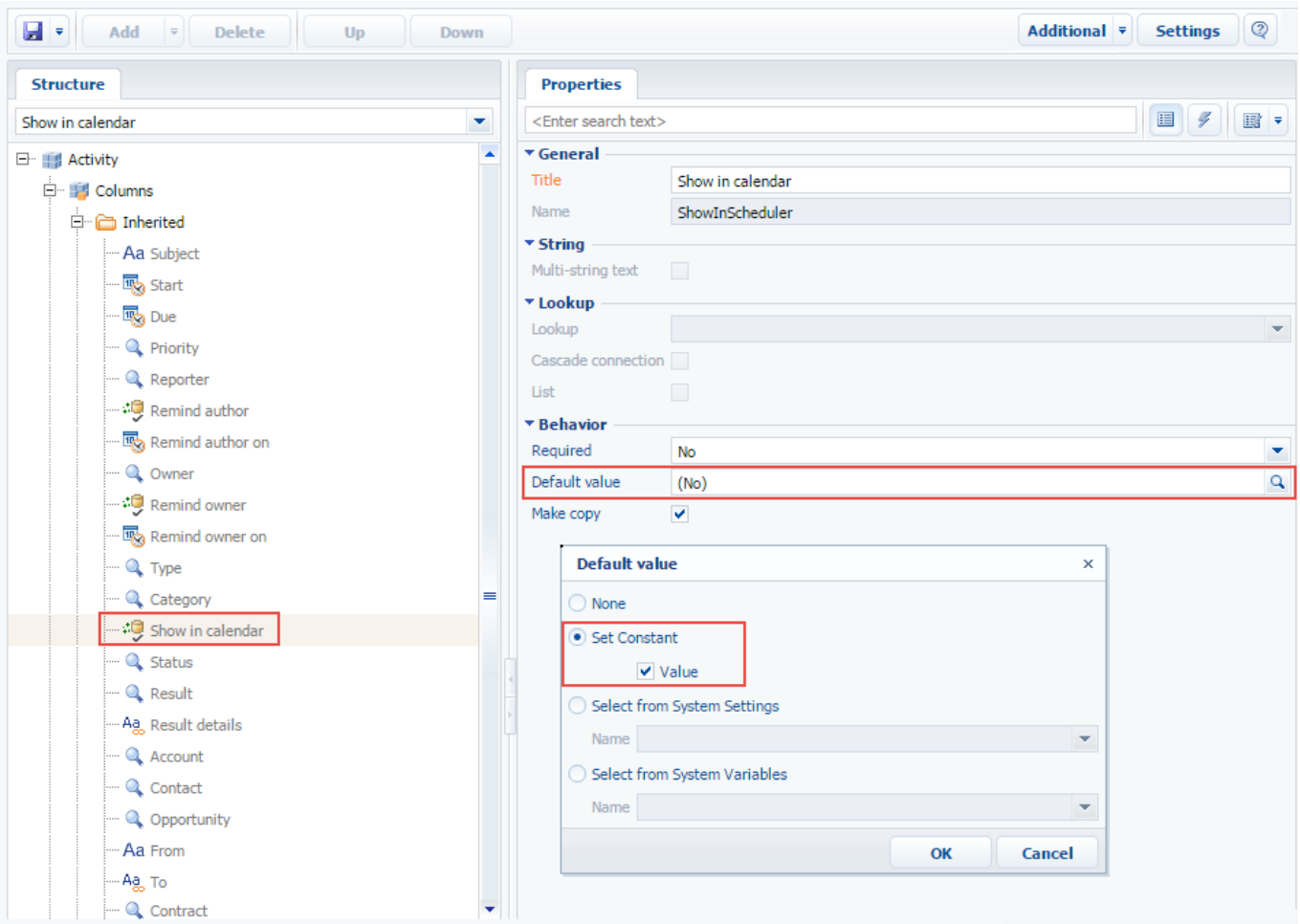
For this purpose, select the [Show in calendar] column in the list of inherited object columns and edit its [Default Value] property as shown in Figure 3.

In general, the system proposes the following types of default values:

Constant	String, number, value from lookup, logic value.
System setting	The full list of settings is available in the configuration in the [System Settings] section and it may be supplemented by a custom system setting.
System variable	The bpm'online system variables are global variables storing data on the system settings. However, unlike system settings whose values may vary for various users, the values of system variables are fixed for all users. The full list of system variables is implemented at the core level and cannot be modified by a user: • New ID; • New successive ID; • Current user; • Current user contact; • Current user account; • Current date and time value; • Current date value; • Current time value.
No default value is specified	

To implement the case, the constant value is selected as the default value.

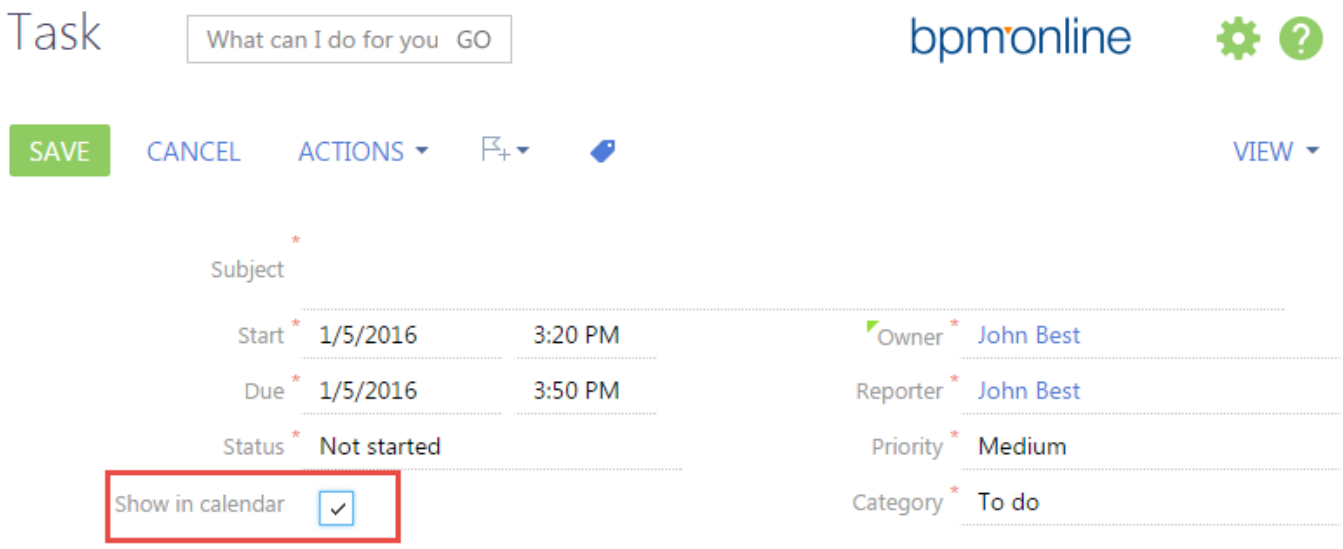
Fig. 3. – Setting a default value for the [Show in calendar] column



3. Publish the object schema

When the schema is published and the system web-page is updated, the [Display in Calendar] checkbox will be selected automatically on the activity edit page when creating a new activity (Fig. 4).

Fig. 4. Demonstrating the default value setting



Example of setting a default value in the source code of the edit page

Case description

When creating a new project, the [Deadline] field value on the project edit page must be the [Start] value + 10 days by default.

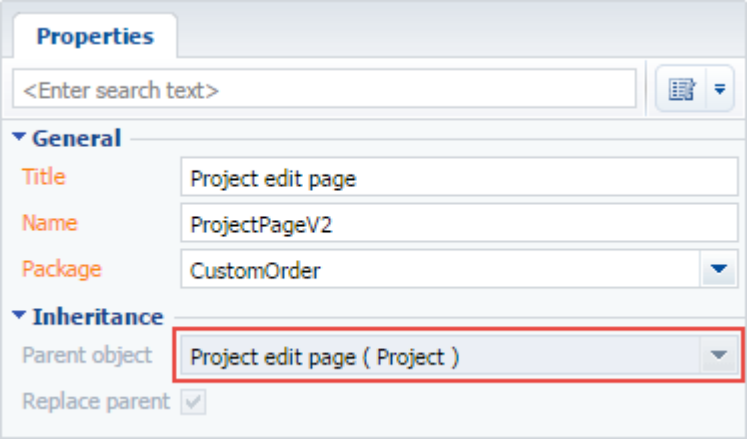
Case implementation algorithm

1. Create a replacement project edit page in a custom package

A replacement client module must be created and [Project edit page] must be specified as the parent object in it (Fig. 5).

The procedure for creating the replacement page is described in the article **Creating a custom client module schema**.

Fig. 5. — Properties of the replacement edit page



2. Add the setDeadline handler method to the methods collection of the view model.

3. Add the redefining of the onEntityInitialized base method to the methods collection of the view model.

Below is the full code for the replacement schema of the project edit page.

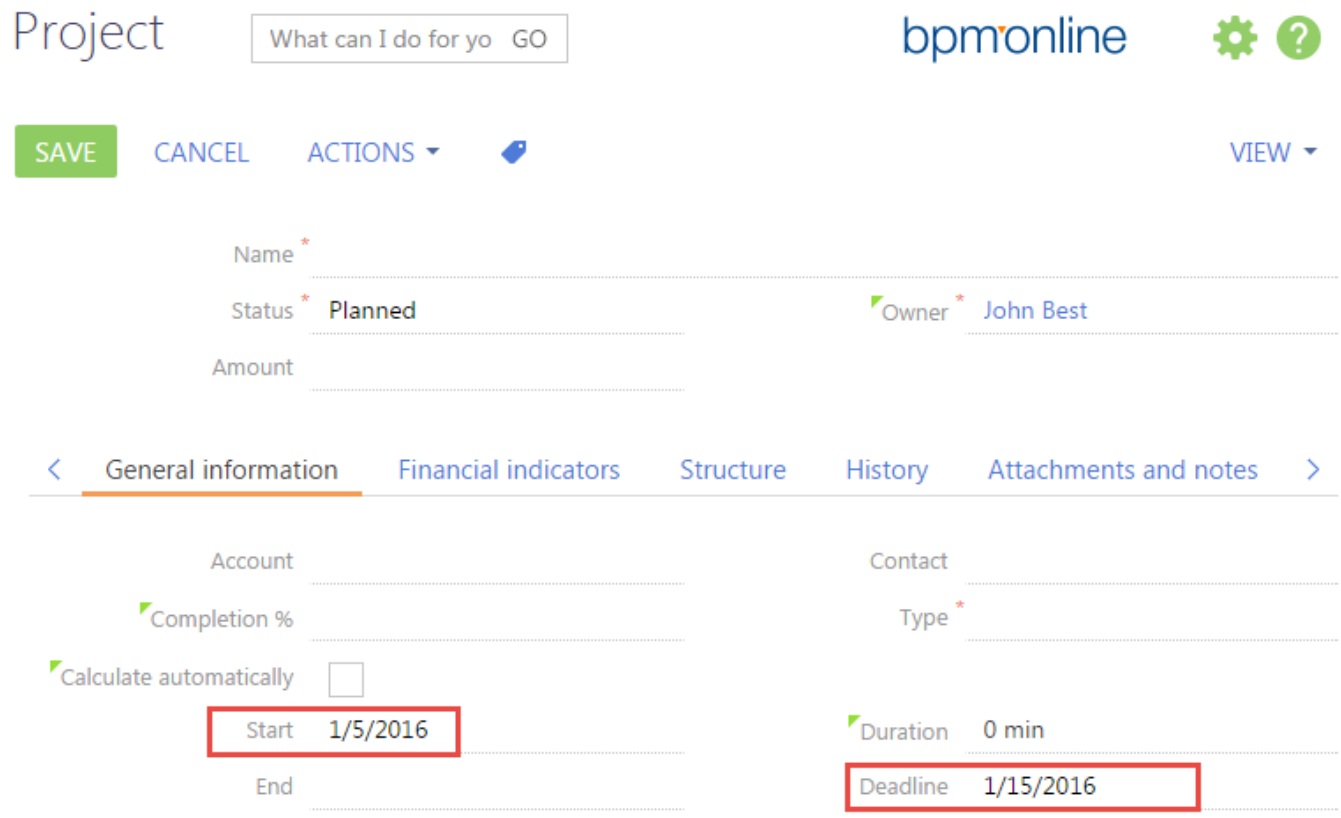
```
define("ProjectPageV2", [],
    function() {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Project",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            methods: {
                // Redefining of the base method
Terrasoft.BasePageV2.onEntityInitialized
                // which actuates upon the completion of the edit page object schema
                // initialization.
                onEntityInitialized: function () {
                    // The parent implementation of the method is called.
                    this.callParent(arguments);
                    // Calling of the handler method calculating the value for the
                    // [Deadline] field.
                    this.setDeadline();
                },
                // Handler method calculating the value for the [Deadline] field.
                setDeadline: function() {
                    if (!this.get("Deadline") && this.isNewMode()) {
                        var startDate = this.get("StartDate");
                        var differ = 10 *
```

```
this.Terrasoft.DateRate.MILLISECONDS_IN_DAY;
                        this.set("Deadline", new Date(startDate.getTime() + differ));
                    }
                }
            },
            diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
        };
    });
```

4. Save the schema.

When the schema is saved and the system web-page is updated, the date complying with the date in the [Start] field + 10 days will be set on the project edit page when creating a new project (Fig. 6).

Fig. 6. – Demonstrating the setup of a computed default value



How to add the field validation

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 analysed for compliance with business requirements. This method is featured by the object return with validation results that have two properties:

- *fullInvalidMessage* — a string displayed in the data window when making an attempt to save a page with an invalid field value.
- *invalidMessage* — a string with a message displayed under a control item when making an attempt to enter incorrect data into a field.

If the value validation is successful, the validator method returns the object with empty strings.

To start the field validation, the corresponding view model column must be bound to a specific validator. For this purpose, the *setValidationConfig* base method must be overridden in the view model by calling the *addColumnValidator* method in it.

The addColumnValidator method has two parameters:

- name of the view model column, to which the validator is bound;
- name of the column value validate method.

> NOTE
>
> 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.

Sequence of actions for adding the validation of field values:

1. Add the validator method to the collection of methods of the view model.
2. Override the setValidationConfig method and connect the validator to the corresponding view model column in it.

Below are several examples of adding the validation to the edit page field.

Example 1

Case description

Set the validation on the opportunity edit page as follows: the value in the [Created on] field must be less than the value in the [Closed on] field.

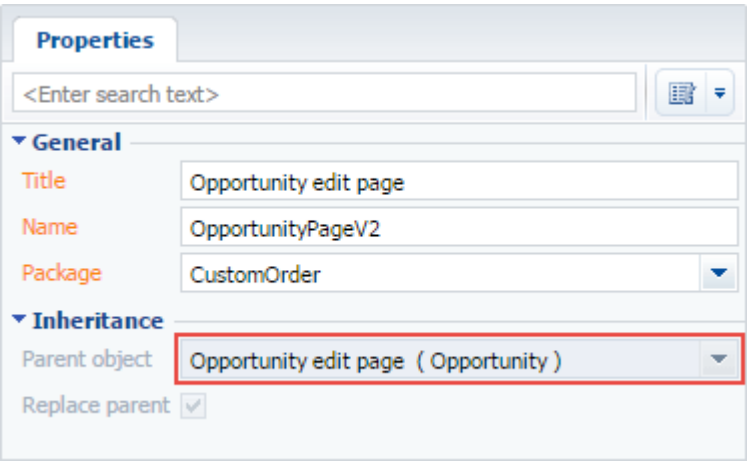
Case implementation algorithm

1. Create a replacement client module of the opportunity edit page

A replacement client module must be created and [Opportunity edit page] must be specified as the parent object (Fig. 1).

The procedure for creating the replacement page is described in the article **Creating a custom client module schema**.

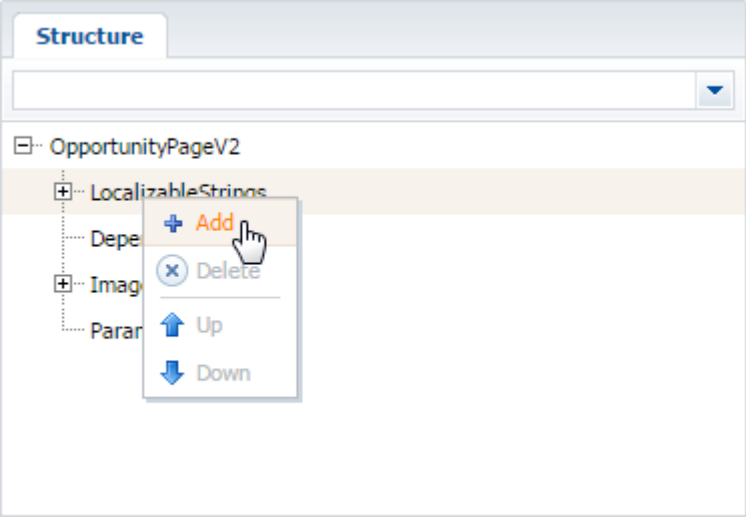
Fig. 1. — Properties of the replacement edit page



2. Add the "Created on must be less than Closed on" string to the localizable strings collection of the replacement page schema

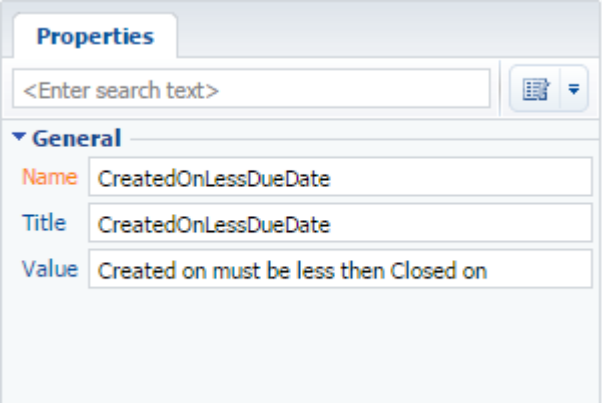
For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure item (Fig. 2).

Fig. 2 — Adding a localizable string to the schema



Fill in the properties for the created string as shown in Fig. 3.

Fig. 3. — Properties of a custom localized string



3. Add the required methods to the collection of methods of the view model

The dueDateValidator validation method, which checks whether the condition is fulfilled, must be added. The setValidationConfig base method must be also overridden and the validate method must be connected to the [DueDate] and [CreatedOn] columns.

```
define("OpportunityPageV2", [],
    function () {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Opportunity",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/,
            // Methods collection of the view model.
            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 displayed in the data window
                        // when saving a page.
                        fullInvalidMessage: invalidMessage,
                        // Validation error message displayed under the control item.
                        invalidMessage: invalidMessage
                    };
                },
                // Redefining the base function 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);
                }
            }
        };
    });
```

4. Save the created replacement page schema

When a schema is saved and the system web-page is updated, 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 less than the date of closing). The data window will appear when making an attempt to save such a sale (Fig. 5).

Fig. 4. – Demonstrating the results

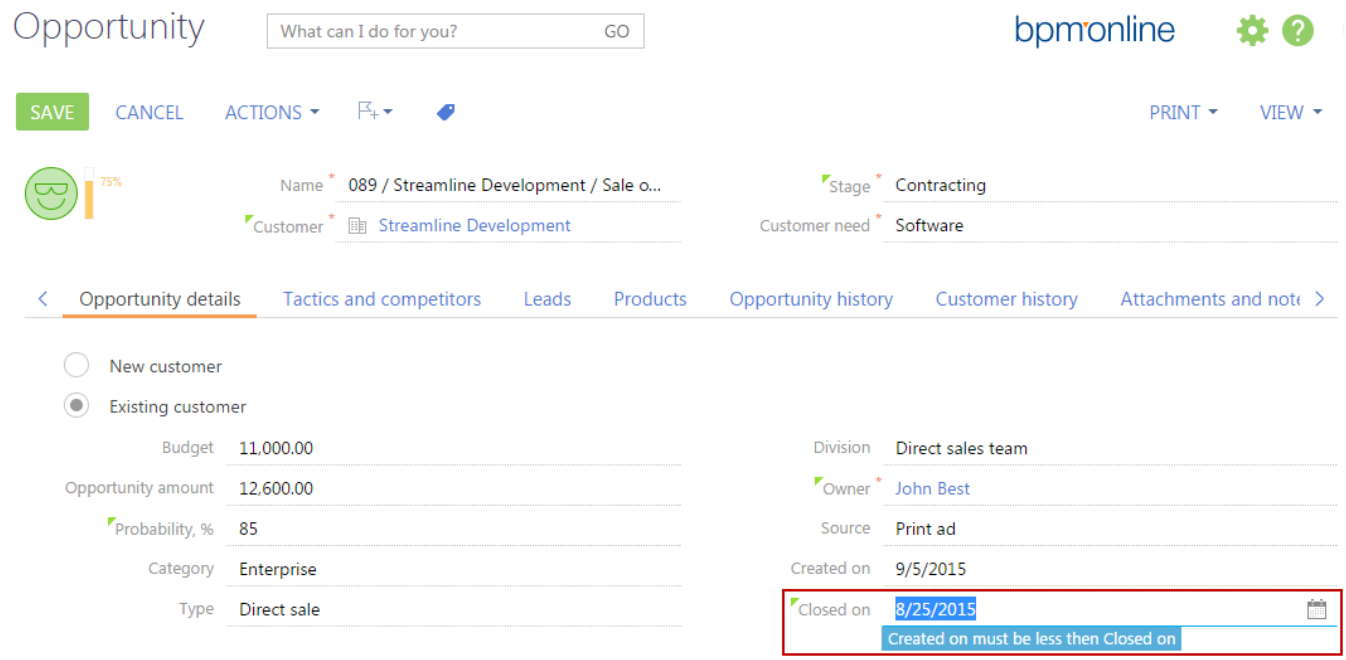
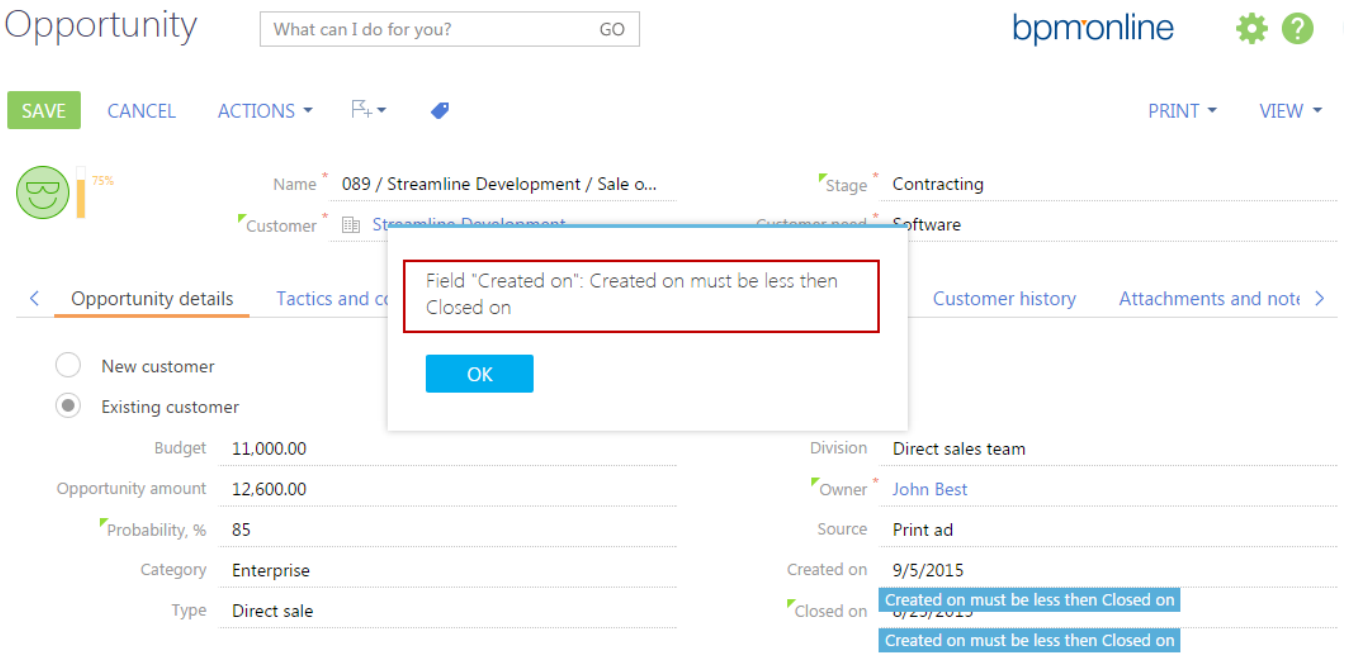


Fig. 5. – Demonstrating the results



Example 2

Case description

Set the [Communication Options] detail validation as follows: telephone number must correspond to the [+44 (99) 9999 9999] mask, otherwise the "Bad Number Format" message will appear.

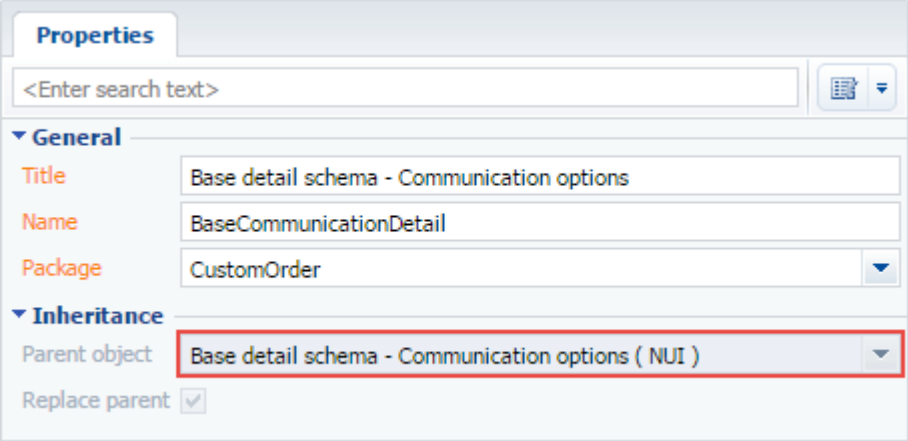
Case implementation algorithm

1. Create a replacement client module for the base schema of the communication options detail

A replacement client module must be created and the [Base detail schema - Communication options] must be specified as the parent object (Fig. 6).

The procedure for creating the replacement page is described in the article **Creating a custom client module schema**.

Fig. 6. — Properties of the replacement edit page

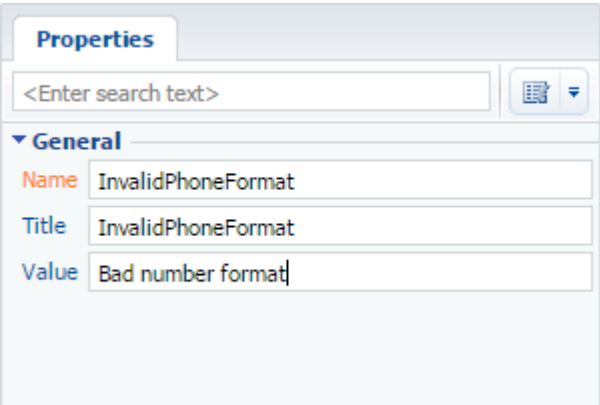


2. Add the "Bad number format" string to the localizable strings collection of the replacement page schema

For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure node.

Fill in the properties for the created string as shown in figure 7.

Fig. 7. — Properties of a custom localized string



3. Add required methods to the methods collection of the view model

Add the phoneValidator validation method which determines whether the condition is fulfilled. Override the *addItem* method for the base schema of the communication options detail where the validate method must be bound to the [Number] column.

> NOTE
>
> The base schema of the communication options detail is featured by creating a new instance of the communication option view model when adding a new communication option. The addItem method is responsible for this process; it returns this instance to the detail view model. The method overriding enables to bind the validate method directly to the newly created instance of the view model using the *addColumnValidator* method.

```
define("BaseCommunicationDetail", ["ConfigurationConstants"],
    function (ConfigurationConstants) {
        return {
            // Methods collection of the view model.
            methods: {
```

```
                // Validate method for the value in the [Number] communication
options view model.
                phoneValidator: function (value) {
                    // Variable for stroing a validation error message.
                    var invalidMessage = "";
                    // Variable for stroing the number check result.
                    var isValid = true;
                    var communicationType = this.get("CommunicationType");
                    var number = value || this.get("Number");
                    // Check of the entered number for its compliance with the enter
mask.
                    if
(ConfigurationConstants.PhonesCommunicationTypes.indexOf(communicationType.value) !==
-1) {
                        isValid = (Ext.isEmpty(number) ||
                            new RegExp("^\\+44\\s\\([0-9]{2}\\)\\s[0-9]{4}\\s[0-9]
{4}$").test(number));
                        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 {
                        fullInvalidMessage: invalidMessage,
                        invalidMessage: invalidMessage
                    };
                },
                // Overriding the base function intiniating the creation of a new
                // instance for the communication option view model.
                addItem: function () {
                    // Passing the instance created via calling parent implementation
to the item variable.
                    var item = this.callParent(arguments);
                    // The phoneValidator validate method is added for the [Number]
column of the created
                    // instant for the communication options view model.
                    this.addColumnValidator("Number", this.phoneValidator, item);
                    return item;
                }
            }
        };
    });
```

4. Save the created replacement page schema

When the schema is saved and the system web-page is updated, the number format validity will be verified on the contact or account edit page when a new phone number is added. If the format is bad, a string with a corresponding message will appear (Fig. 8, 9).

Fig. 8. — Demonstrating the result – the validation condition failed

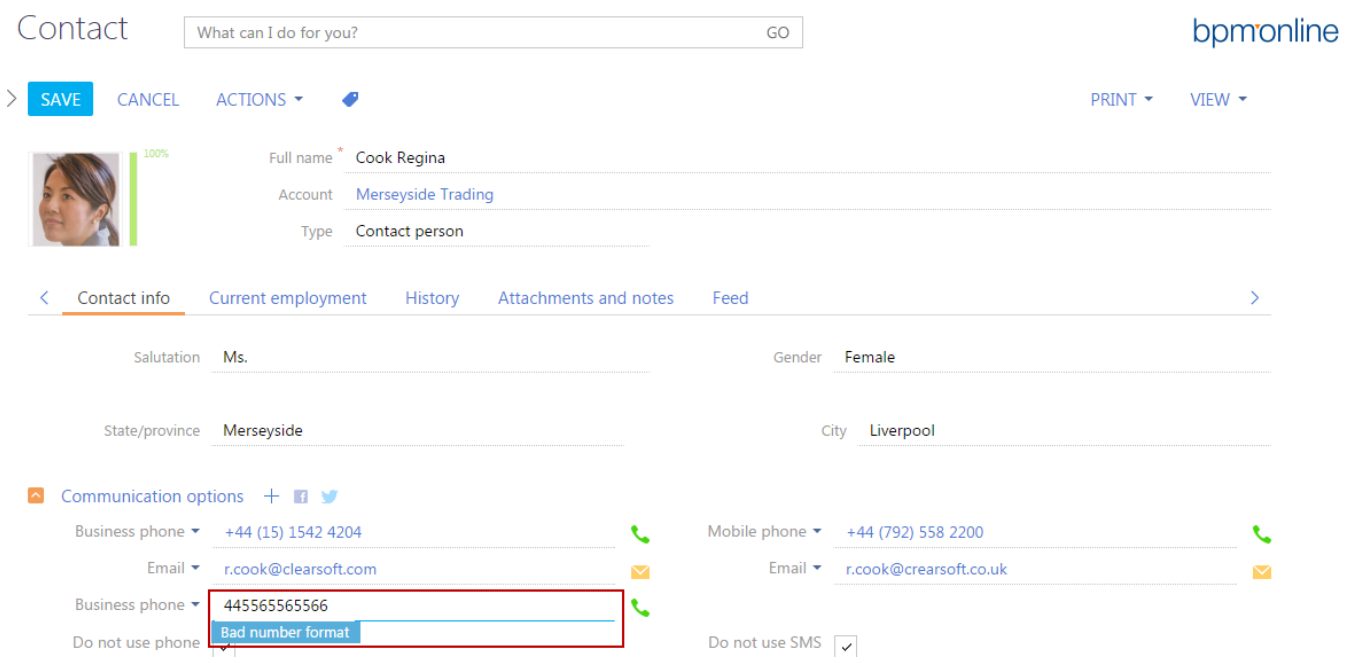
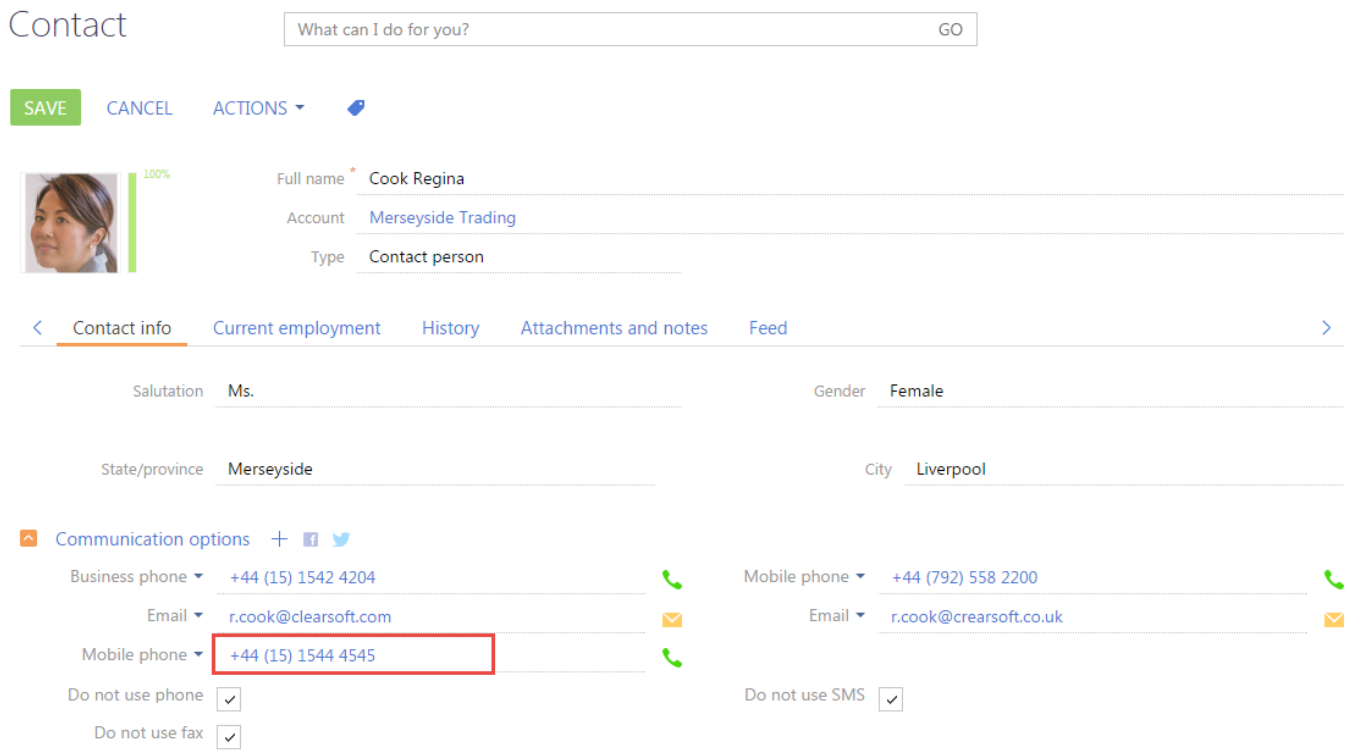


Fig. 9. — Demonstrating the result – the validation condition is fulfilled



Using filtration for lookup fields. Examples

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**

article. The detailed case for using the [FILTRATION] business rule is set forth in the **Adding the [FILTRATION] rule to the edit page** 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

The name of the column for which filters are set must be added to the attributes property of the view model.

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. These columns will be also displayed additionally in the drop-down list;
- *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	Item.
index	Number	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 article **EntitySchemaQuery filters handling**.

> NOTE
>
> 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.

Below is an example of using the filtration via explicit description of filters in the attributes 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.

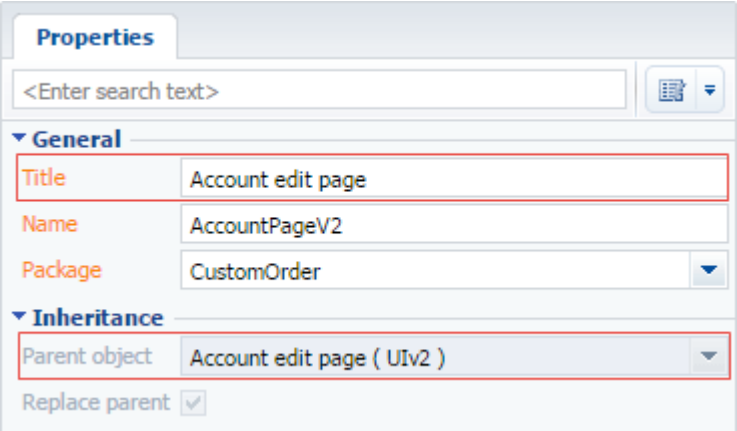
Case implementation algorithm

1. Create a replacement account edit page

A replacement client module must be created and [Account edit page] must be specified as the parent object in it (Fig. 1).

The procedure for creating the replacement page is described in the article **Creating a custom client module schema**.

Fig. 1. — Properties of the replacement edit page



2. Add the [Owner] column to the attributes property of the view model

Specify the type of column data (dataValueType property), namely the Terrasoft.DataValueType.LOOKUP in the configuration object, in the [Owner] column 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.

Below is the code of the replacement page schema

```
define("AccountPageV2", ["BusinessRuleModule", "ConfigurationConstants",
"ProcessModuleUtilities"],
    function (BusinessRuleModule, ConfigurationConstants, ProcessModuleUtilities) {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Account",
            // List of the schema attributes.
            attributes: {
                // Name of the view model column.
                "Owner": {
                    // Type of this attribute data.
                    dataValueType: Terrasoft.DataValueType.LOOKUP,
                    // The configuration item 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 core schema.
                                // Contact 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;
                            }
                        ]
                    }
                }
            },
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            methods: {},
            diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
        };
    });
```

3. Save the created replacement page schema

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.

Adding an action panel

Starting with version 7.8.0, bpm'online has a new edit page module called "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.

Case implementation algorithm

1. Create the [OrderActionsDashboard] Schema of the Edit Page View Model

Specify [SectionActionsDashboard] as a parent object. Enter the following code in the source code edit window:

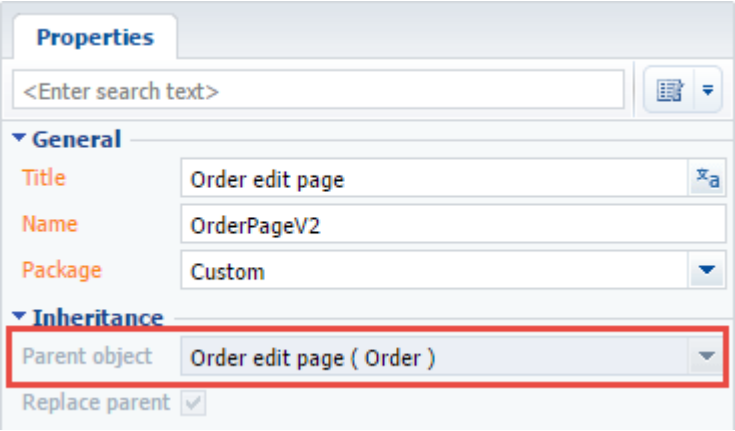
```
define("OrderActionsDashboard", [], function () {
    return {
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        methods: {},
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    };
});
```

2. Create a replacing order edit page

Create a replacing client module and specify the [Order edit page] module as parent (Fig. 1).

For more information on the process of creating replacing pages, please see the "**Creating a custom client module schema**" article.

Fig. 1. Order edit page replacing schema properties



3. Add a configuration object with the module settings in the modules collection of the page schema

To do this, add the source code of the replacing page module on the source code tab. Add a configuration object with the module settings in it to the modules collection of the view model.

```
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"
                    },
                    "dashboardConfig": {
                        // Connection between activities and page object.
                        "Activity": {
                            // Page object column name.
                            "masterColumnName": "Id",
                            // COlumn name in the [Activity] object.
                            "referenceColumnName": "Order"
                        }
                    }
                }
            }
        }
    }
}/**SCHEMA_MODULES*/,
```

4. Add a configuration object with the settings determining the module position in the diff

array

```
diff: /**SCHEMA_DIFF*/[
    {
        "operation": "insert",
        "name": "ActionsDashboardModule",
        "parentName": "ActionDashboardContainer",
        "propertyName": "items",
        "values": {
            "classes": { wrapClassName: ["actions-dashboard-module"] },
            "itemType": Terrasoft.ViewItemType.MODULE
        }
    }
]/**SCHEMA_DIFF*/
```

Below is the full source code of the replacing page module.

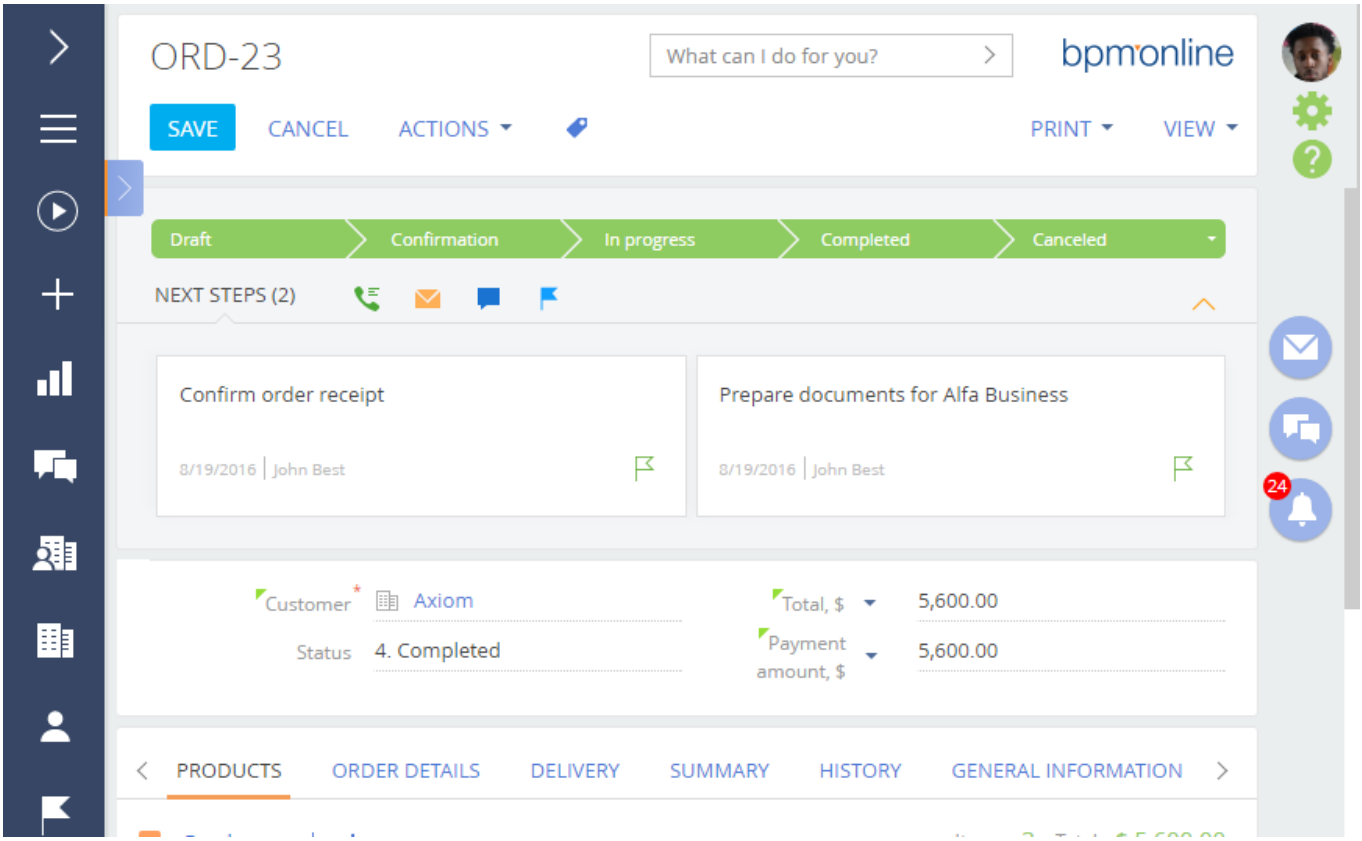
```
define("OrderPageV2", [],
    function () {
        return {
            entitySchemaName: "Order",
            attributes: {},
            modules: /**SCHEMA_MODULES*/{
                "ActionsDashboardModule": {
                    "config": {
                        "isSchemaConfigInitialized": true,
                        // Schema name.
                        "schemaName": "OrderActionsDashboard",
                        "useHistoryState": false,
                        "parameters": {
                            // Configuration object of the module schema.
                            "viewModelConfig": {
                                // Page entity schema name.
                                "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"
                                },
                                // Displays action panel module, [true] bu default.
                                "useDashboard": true,
                                // Displays Content block, [true] bu default.
                                "contentVisible": true,
                                // Displays Header block, [true] bu default
                                "headerVisible": true,
                                // Configuration obkject of the panel items.
                                "dashboardConfig": {
                                    // Connection between activities and page object.
                                    "Activity": {
                                        // Page object column schema name.
                                        "masterColumnName": "Id",
                                        // Column name in the [Activity] object.
                                        "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*/
        };
    });
```

5. Save the created replacing page schema

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. 2).

Fig. 2. Case result demonstration



Adding a new channel to the action dashboard

General Information

A new edit page module has been implemented in bpm'online 7.8 — the action dashboard. The action dashboard is designed to display information about the current state of a record

This page element is described in the "**Action dashboard**" article.

The ways of adding the action dashboard 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

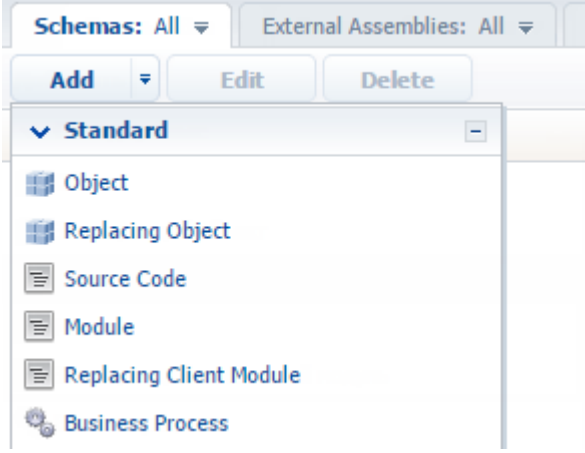
Add a new custom channel to the action dashboard of the contact edit page. The channel must have the same functionality as the call results channel (CallMessagePublisher channel).

Case execution algorithm

1. Add the CallsMessagePublisher source code schema

Open the system settings. Go to the [Configuration] section, and on the [Schema] tab, execute the [Add] > [Source code] menu command (Fig. 1).

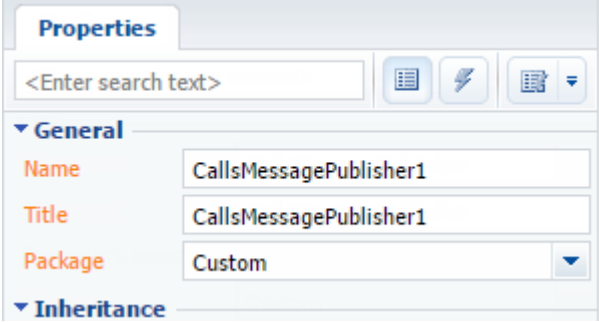
Fig. 1. Added source code schemas



To create a schema, set the following settings (Fig. 2):

- The [Title] property — "Call messages logging publisher"
- The [Name] property — "CallsMessagePublisher"

Fig. 2. Added source code schemas



In the created schema, add the new *CallsMessagePublisher* 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

Open the system settings. Go to the [Configuration] section, and on the [Schema] tab, execute the [Add] > [Replacing client module] menu command (Fig. 1). Set the *SectionActionsDashboard* schema of the *ActionsDashboard* package as the parent of the replacing client schema. Set the *SectionActionsDashboard* value as the title.

Read more about replacing schemas in the "**Creating a custom client module schema**" article.

> 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. The new channel will be visible on the edit pages of those sections, which are connected to *SectionActionsDashboard*. Set the operations of inserting the *CallsMessageTab* tab and message container in this property.

```
//
diff: /**SCHEMA_DIFF*/[
    // Adding 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 CallsMessageModule module.
    {
        "operation": "insert",
        "name": "CallsMessageModule",
        "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": "callsMessagePublisherModule",
            // 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*/
```

You should also 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.

```
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": "Calls"
        };
        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("Calls");
        return publishers;
    }
},
```

Save and publish the schema.

3. Create the CallsMessagePublisherModule module

The *CallsMessagePublisherModule* serves as container that renders the *CallsMessagePublisherPage* in the *SectionActionsDashboard*.

Open the system settings. Go to the [Configuration] section, and on the [Schema] tab, execute the [Add] > [Module] menu command (Fig. 1).

To create a module, configure the following settings (Fig. 2):

- The [Title] property — "Call messages logging publisher module"
- The [Name] property — "CallsMessagePublisherModule"

Add the following source code to the created module.

```
define("CallsMessagePublisherModule", ["BaseMessagePublisherModule"],
    function() {
        // Defining the class.
        Ext.define("Terrasoft.configuration.CallsMessagePublisherModule", {
            // Basic class.
            extend: "Terrasoft.BaseMessagePublisherModule",
            // Short class name.
            alternateClassName: "Terrasoft.CallsMessagePublisherModule",
```

```
            // Initialization of the page that will be rendered in this module.
            initSchemaName: function() {
                this.schemaName = "CallsMessagePublisherPage";
            }
        });
        // Returns the class object defined in the module.
        return Terrasoft.CallsMessagePublisherModule;
    });
```

Save the module.

4. Create the CallsMessagePublisherPage page

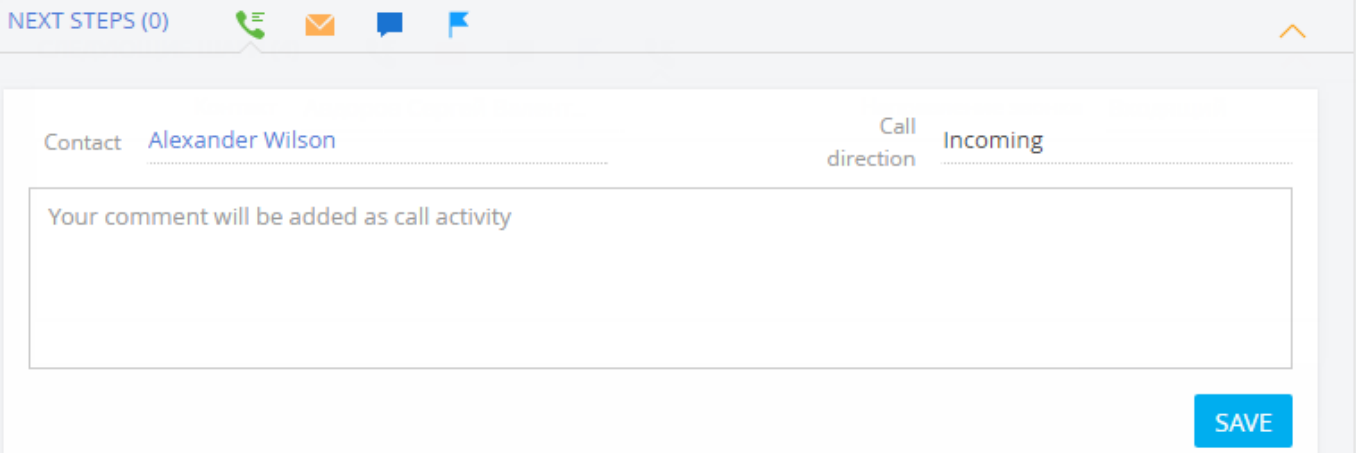
Open the system settings. Go to the [Configuration] section, and on the [Schema] tab, execute the [Add] > [Page view model schema] menu command (Fig. 1). Set the *BaseMessagePublisherPage* schema of the *MessagePublisher* package as the parent object for the created page. Set the "CallsMessagePublisherPage" 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"
    };
}
```

As a result you will get the new channel in the *SectionActionsDashboard* (Fig. 3).

Fig. 3. An example of a custom CallsMessagePublisher channel in the SectionActionsDashboard of the [Contacts] section.



Displaying contact's time zone

General information

Working with different timezones was introduced in version 7.8. Contact pages now display information about contact's timezone. Timezone values, such as time difference between the current 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 on a page using the section wizard. Below is a sequence of adding a contact timezone display element on a custom page.

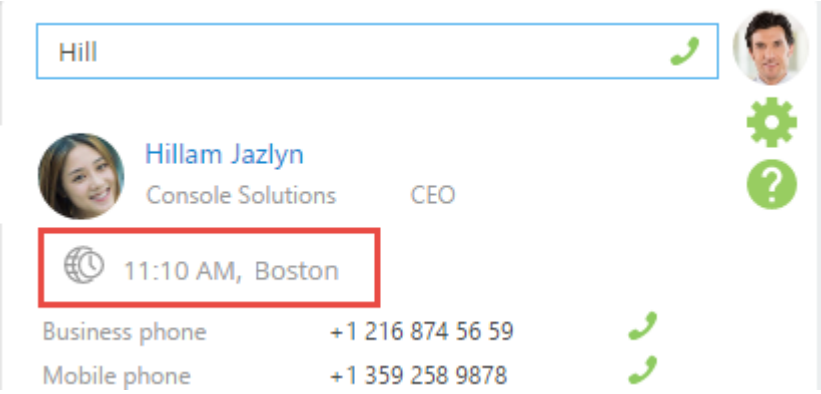
General procedure of adding a timezone element

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 on the call panel. 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



Case implementation algorithm

1. Create a replacing page schema module

The call panel used the *SubscriberSearchResultItem* schema to display subscribers found by phone number. 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).

Fig. 2. Creating a replacing module

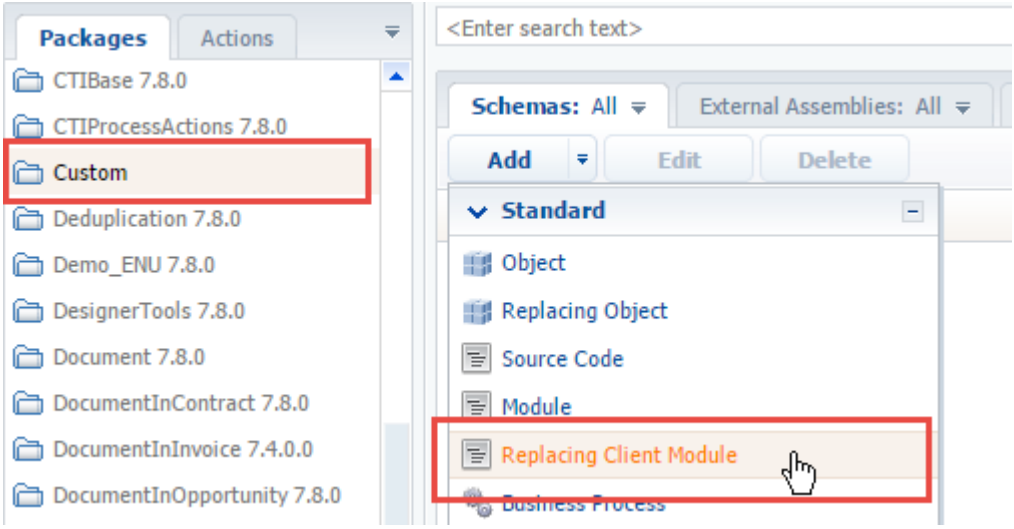
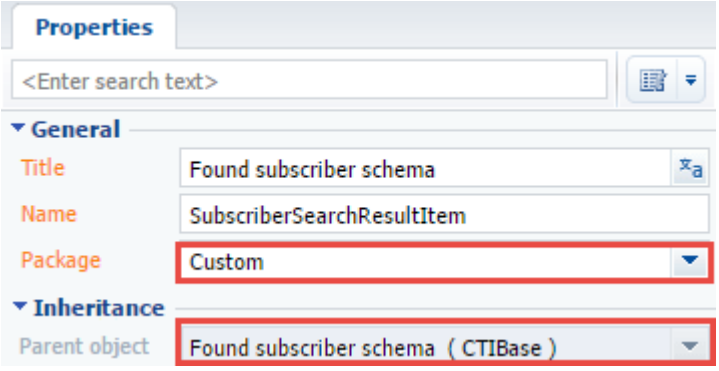


Fig. 3 Replacing module properties



For more information about replacing client modules please see the "**Creating a custom client module schema**" article.

The contact timezone display item is created with the help pf the *TimezoneGenerator* module. *TimezoneMixin* is used for searching contact timezone. Add these modules to the *SubscriberSearchResultItem* schema dependencies.

```
define("SubscriberSearchResultItem",
    // Dependency from TimezoneGenerator, TimezoneMixin.
    ["TimezoneGenerator", "TimezoneMixin",
    ...
```

2. Add timezone display element

To display a contact timezone element, add configuration object with the following properties to the diff array:

```
{
    // Adding new element.
    "operation": "insert",
    // Parent element is SubscriberSearchResultItemContainer.
    "parentName": "SubscriberSearchResultItemContainer",
    // New element is added to the collection of the parent.
    "propertyName": "items",
    // Element name.
    "name": "TimezoneContact",
    // Element properties.
    "values": {
        // Element type.
        "itemType": Terrasoft.ViewItemType.CONTAINER,
        // Generator method is called for generating configuration.
        "generator": "TimezoneGenerator.generateTimeZone",
        // Container visibility is bound to attribute.
        "visible": {"bindTo": "IsShowTimeZone"},
        // Element style.
        "wrapClass": ["subscriber-data", "timezone"],
        // Bunding title to attribute.
        "timeZoneCaption": {"bindTo": "TimeZoneCaption"},
        // Binding city to attribute.
        "timeZoneCity": {"bindTo": "TimeZoneCity"}
    },
    // Element position in parent container.
    "index": 2
}
```

After saving the schema and refreshing the application page, the subscriber search results will display timezone icon without contact's current local time.

3. Connect timezone search

To run the contact timezone search, pass contacts unique Id to the *init* method. *TimezoneMixin* sets results to the following attributes:

- *TimeZoneCaption* – contact's timezone name and current local time.
- *TimeZoneCity* – name of the city for which the timezone is set.

The schema of found subscribers the identifier value is stored in the *Id* attribute. Also, the *Type* attribute determines whether the entity is a contact, and display the element for required type only.

The source code for the timezone search function:

```
// Block for creating attributes.
attributes: {
    // Name of the attribute responsible for timezone element display status.
    "IsShowTimeZone": {
        // Attribute data type.
        "dataValueType": Terrasoft.DataValueType.BOOLEAN,
        // Attribute type in the model.
        "type": Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
        // Default value.
        "value": true
    }
},
// Mixing connection block.
mixins: {
    // Connecting mixin.
    TimezoneMixin: "Terrasoft.TimezoneMixin"
},
// Method definition block.
methods: {
    // Class constructor.
    constructor: function() {
        // Calling base constructor.
        this.callParent(arguments);
        // Indicats that 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");
            // Contact timezone search.
            this.mixins.TimezoneMixin.init.call(this, contactId);
        }
    },
    // Determines if the subscriber is a contact.
    isContactType: function() {
        // Subscriber type.
        var type = this.get("Type");
        // Gets comparison result.
        return type === "Contact";
    }
}
```

To display result, save the schema and update the subscriber search page. A hint pop-up is also displayed when the cursor hovers over the element.

The complete source code of the schema is available below:

```
// 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, contact's current local time and city are displayed. Now, display styles must be set up.

4. Display style setup

During the second step, the configuration object placed in the diff array already has preliminary display settings.

Use the *inex* 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* class to set styles for text elements in the schema. The element generator provides the *wrapClass* parameter for ability to manage styles.

The standard styles are insufficient for positioning of the element and its visual highlighting. In this case custom styles must be added. For this purpose, create *UsrSubscriberSearchResultItemCss* module in the custom package.

> **Note**
>
> The module name prefix can be modified or deleted if you modify the [Object name prefix] system setting. By default, the "Usr" value is specified in the [Object name prefix] system setting.

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;
}
```

Lo load module with styles, add the following code to the module on the [Source code] tab and save the schema.

```
define("UsrSubscriberSearchResultItemCss", [],
        function() {
            return {};
        });
```

To connect styles to the subscriber search page schema, which was created during steps 1-3, add the *UsrSubscriberSearchResultItemCss* module to dependencies:

```
define("SubscriberSearchResultItem",
    // Dependency from TimezoneGenerator, TimezoneMixin.
    ["TimezoneGenerator", "TimezoneMixin",
    // Dependency from the module with styles.
    "css!UsrSubscriberSearchResultItemCss"],
    ...
```

After saving the schema and refreshing the application page, the styles will be applied and subscriber search resutls will be displayed as shown on Fig. 1.

Adding details

- **Adding an edit page detail**
- **Adding a detail with an editable list**
- **Adding a detail with a 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**
- **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

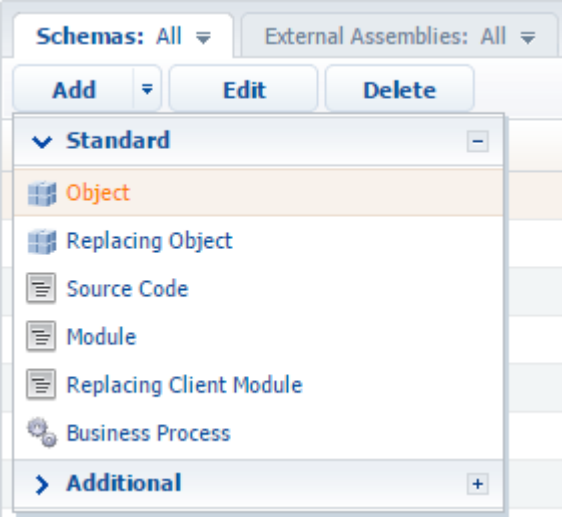
Create a custom [Couriers] detail in the [Orders] section. The detail must contain the list of couriers for the current order.

Case implementation algorithm

1. Create detail object schema

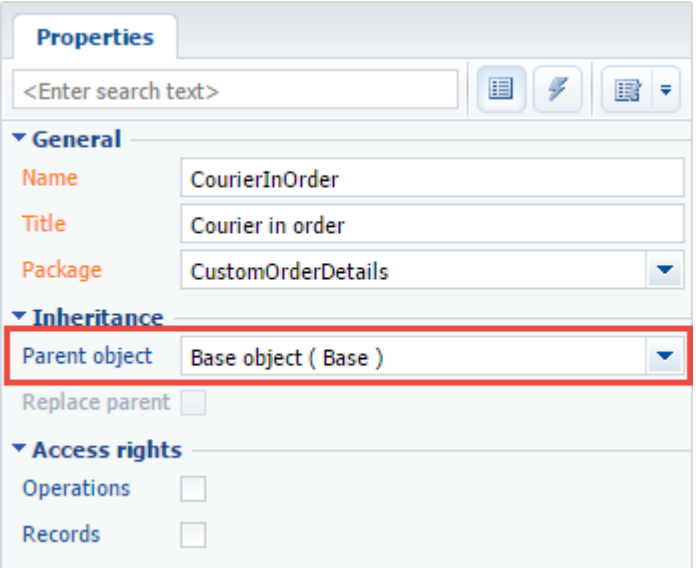
In the settings mode, go to the [Configuration] section, open the [Schemas] tab and execute the [Add] > [Object] menu command (Fig. 1).

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

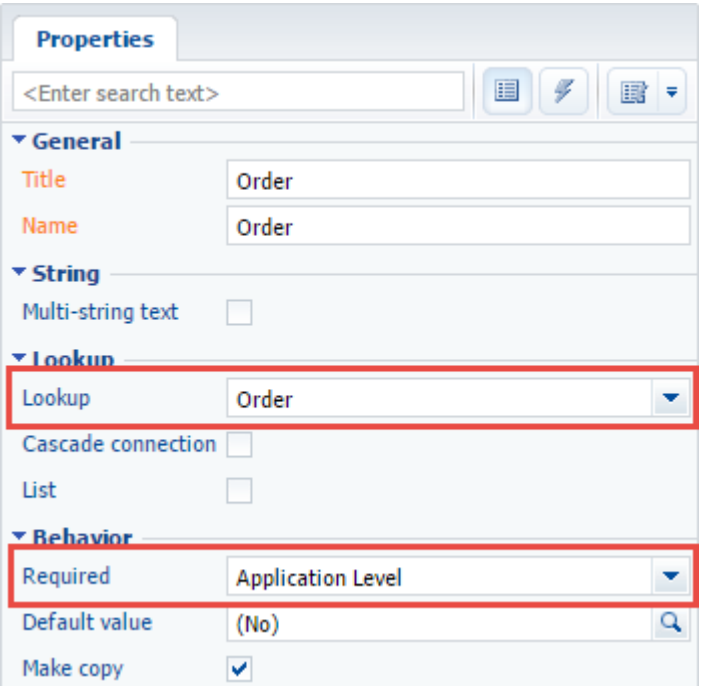
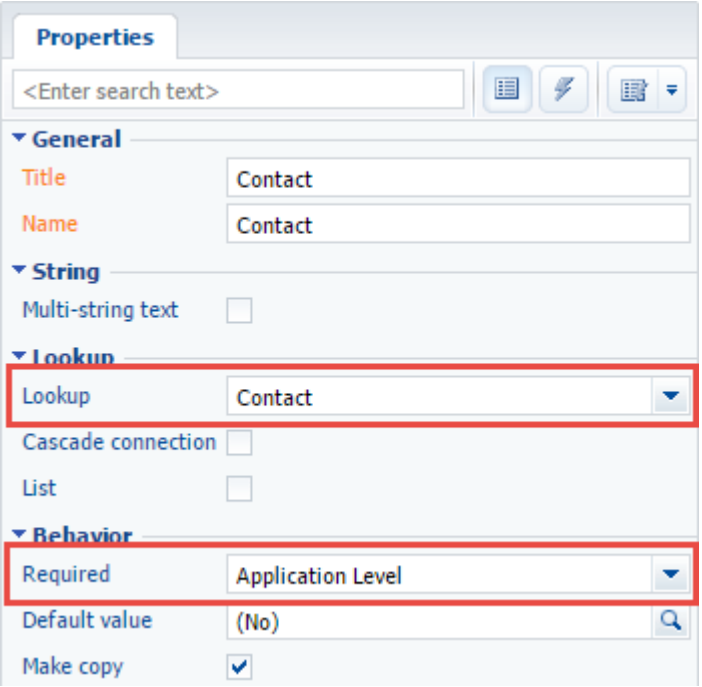


Fig. 4. Specifying the properties of the [Contact] column



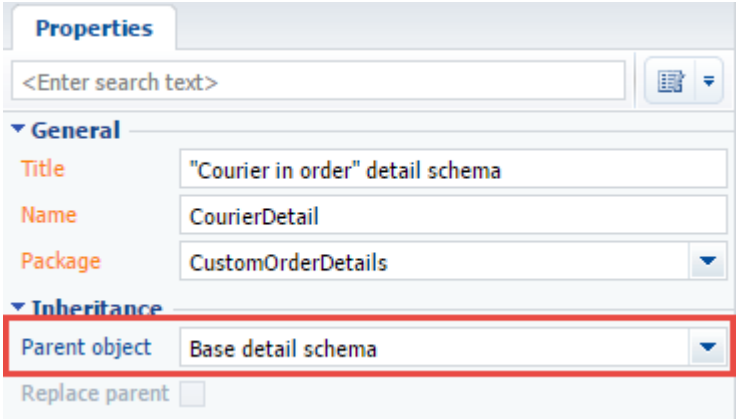
Object schema must be saved and published.

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).

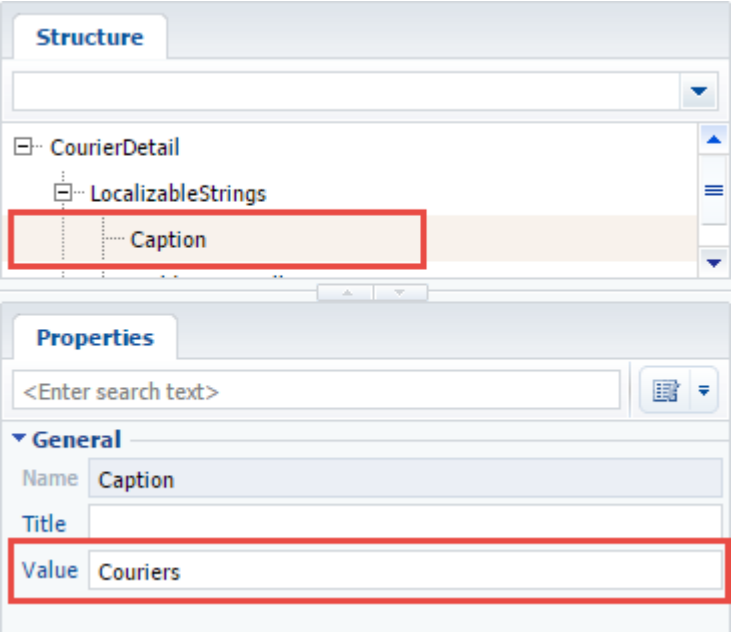
The new module must inherit the functionality of the base detail schema with list *BaseGridDetailV2*, which is available in the *NUI* package. To do this, specify this schema as the parent for the created detail schema (Fig. 5). The rest of the properties must be set up as shown on Fig. 5. In the [Package] 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*/
    };
});
```

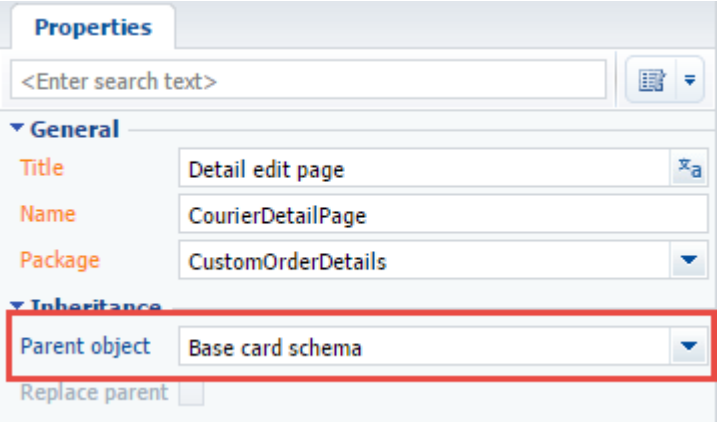
Save the detail schema to apply the changes.

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.

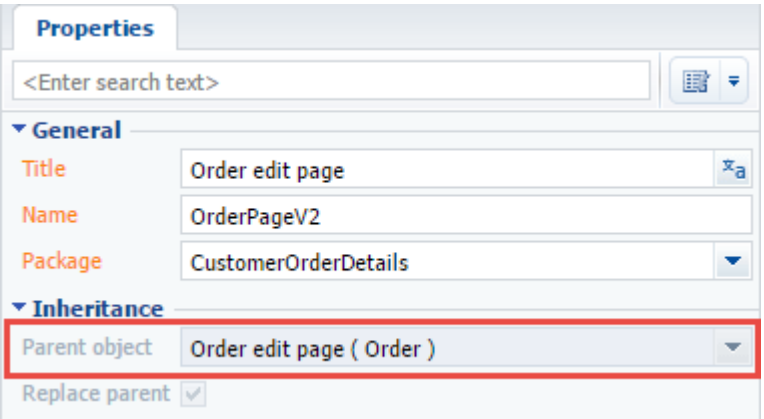
```
define("CourierDetailPage", [], function() {
    return {
        // Detail object schema name.
        entitySchemaName: "CourierInOrder",
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        // Array with modifications.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding 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.

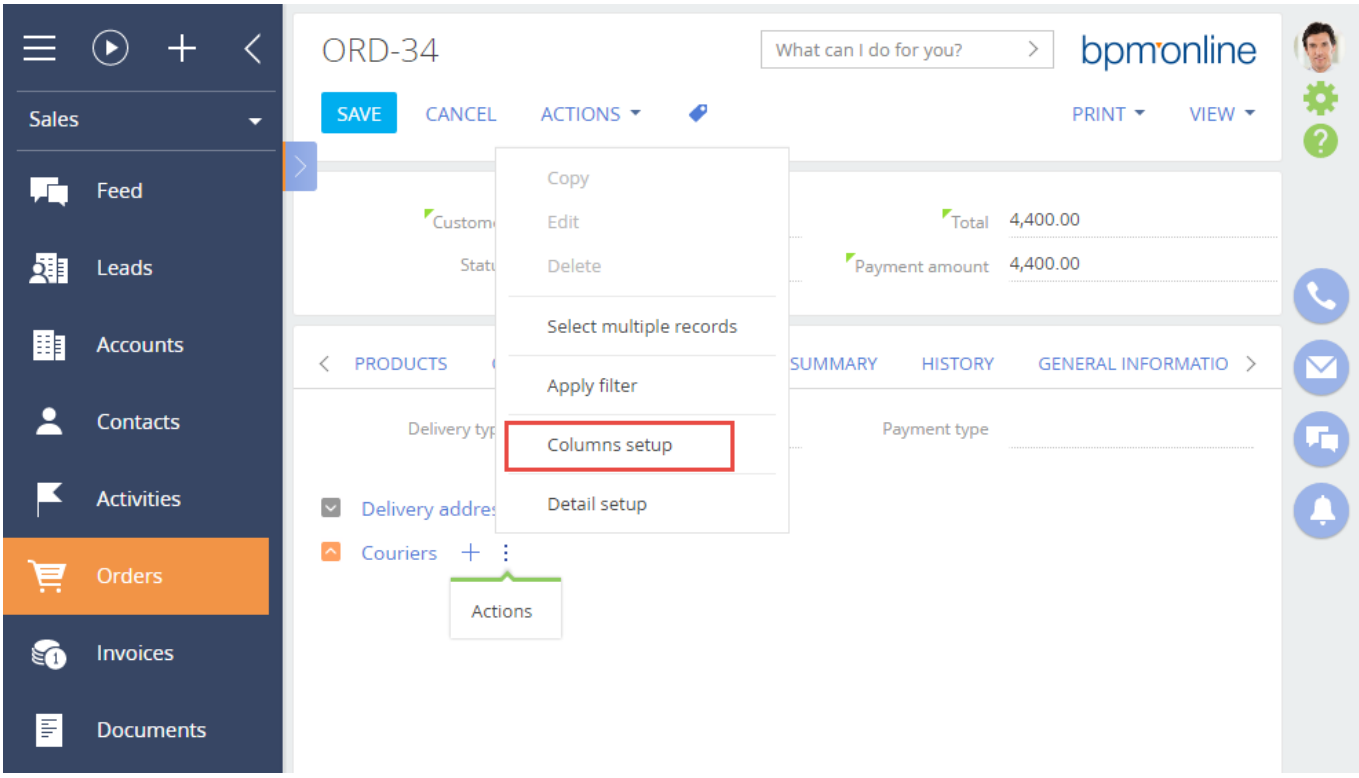
```
DECLARE
-- Schema name of the detail page
@CardSchemaName NCHAR(100) = 'CourierDetailPage',
-- Object schema.
@EntitySchemaName NCHAR(100) = 'CourierInOrder',
-- Detail page caption.
@PageCaption NCHAR(100) = 'Page of detail "Courier In Order"',
-- Empty string.
@Blank NCHAR(100) = ''

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
)
```

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



Adding a detail with an editable list

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.

A detail with an editable list is not a standard bpm'online detail and can not be added to sections using the Detail Wizard. Below is a step-by-step procedure for adding a detail with editable list to an edit page.

General procedure for adding a detail with an editable list to an existing section

To add a detail with an editable list:

1. Create a detail object schema.
2. Create and configure the detail schema.
3. Set up the detail in a replacing section edit page schema.
4. Set up the detail fields.

Case description

Create a custom [Courier services] detail in the [Orders] section. The detail must contain the list of courier services

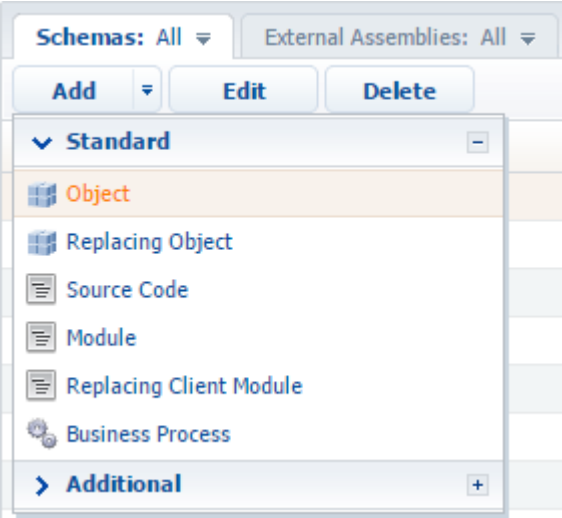
for the current order.

Case implementation algorithm

1. Create detail object schema

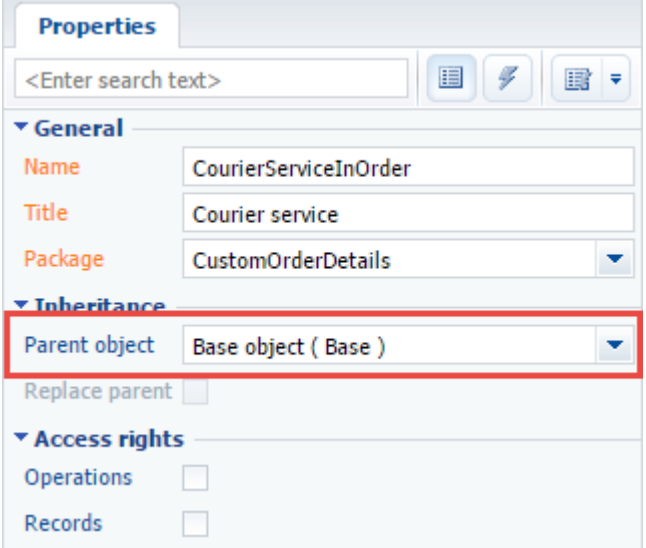
In the settings mode, go to the [Configuration] section, open the [Schemas] tab and execute the [Add] > [Object] menu command (Fig. 1).

Fig. 1. Adding a detail object schema



Select the base *BaseEntity* object (located in the *Base* package) as the parent for the created schema (Fig 2). The rest of the properties must be set up as shown on Fig. 2. In the [Package] property, the system will specify the current package name. For more information on setting schema properties in the Object Designer, please see "**Workspace of the Object Designer**" article.

Fig. 2. Setup of detail object schema properties



Add an [Order] lookup column, which will connect the detail object with the section object, and an [Account] lookup column where the courier service account will be stored. Both columns must be made to be required. Column properties setup is shown in Fig. 3 and Fig. 4.

Fig. 3. Specifying the properties of the [Order] column

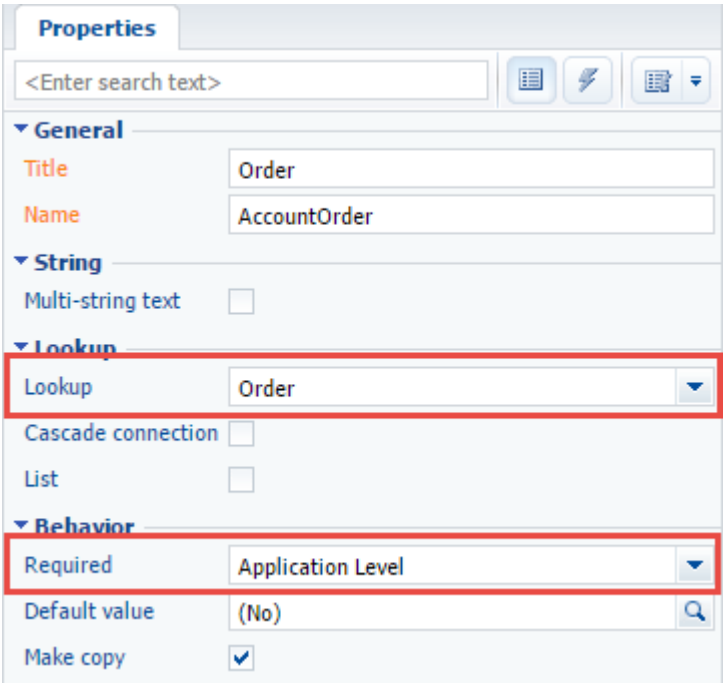
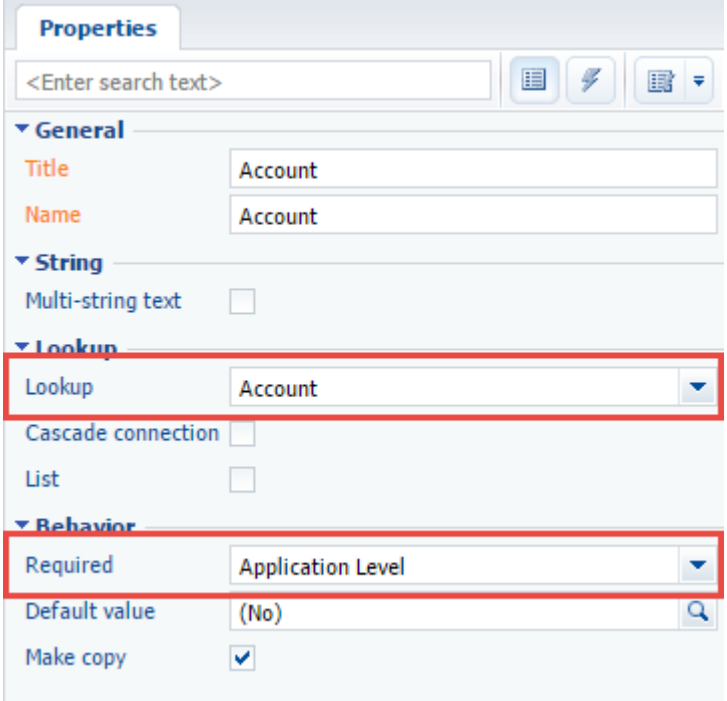


Fig. 4. Specifying the properties of the [Account] column



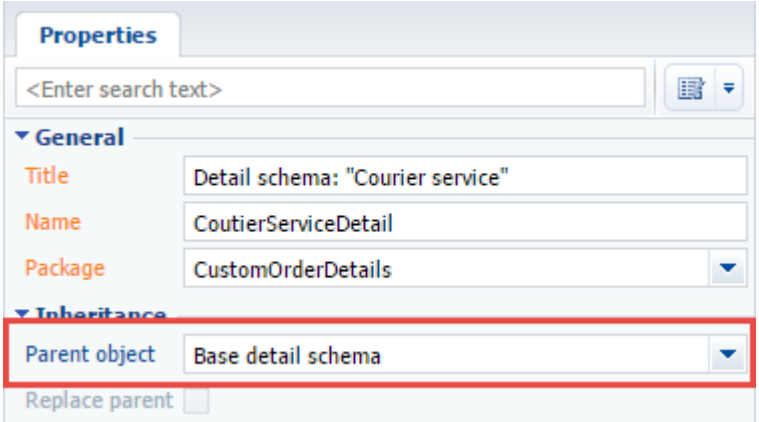
Object schema must be saved and published.

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).

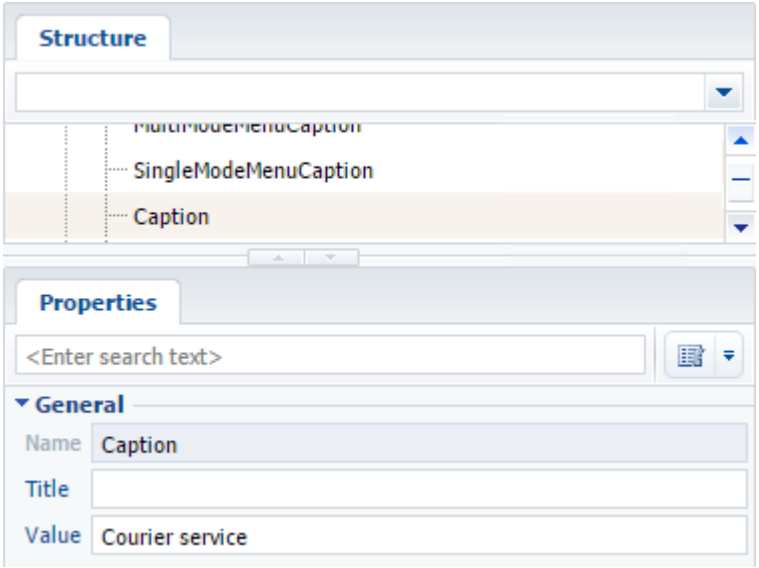
The new module must inherit the functionality of the *BaseGridDetailV2* base schema of details with lists, which is available in the *NUI* package. To do this, specify this schema as the parent for the created detail schema (Fig. 5). The rest of the properties must be set up as shown in Fig. 5. In the [Package] property, the system will specify the current package name.

Fig. 5. Detail schema properties



Set the *Courier service* value for the [Caption] localizable string of the detail schema (Fig. 6). The [Caption] string contains the detail title, shown on the edit page.

Fig. 6. Setting the value of the [Caption] string



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 ConfigurationGridUtilites and OrderUtilities mixins.
3. Set the IsEditable attribute value to "true".
4. In the 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, which must be added to the [Source code] section of the client module designer.

```
// Defining schema and setting its dependencies from other modules.
define("CourierServiceDetail", ["ConfigurationGrid", "ConfigurationGridGenerator",
    "ConfigurationGridUtilities"], function() {
    return {
        // Detail object schema name.
        entitySchemaName: "CourierServiceInOrder",
        // 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
            }
        },
        details: {},
        // Array with modifications.
        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":
```

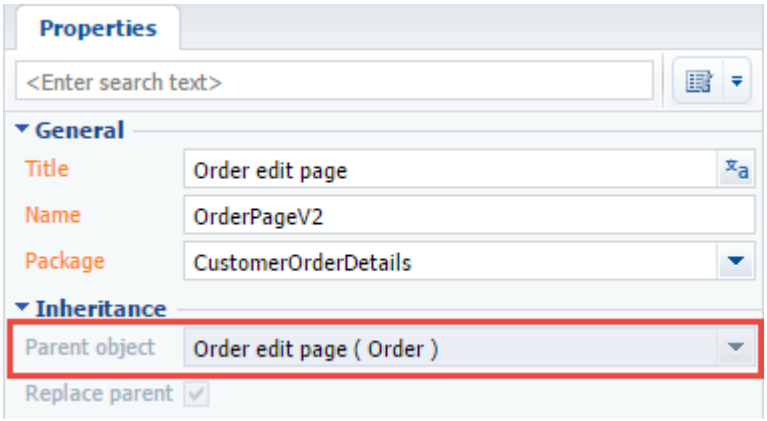
```
this.Terrasoft.controls.ButtonEnums.style.TRANSPARENT,
                            "tag": "remove",
                            "markerValue": "remove",
                            "imageConfig": {"bindTo": "Resources.Images.RemoveIcon"}
                        }
                    ],
                    // 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
                }
            }
        ],
        // Used classes with additional functionality.
        mixins: {
            ConfigurationGridUtilites: "Terrasoft.ConfigurationGridUtilities",
            OrderUtilities: "Terrasoft.OrderUtilities"
        },
        methods: {}
    };
});
```

Save the detail schema to apply the changes.

3. Set up the detail in the 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. 7). For more information on creating replacing schemas, please see the "**Creating a custom client module schema**" article.

Fig. 7. Order edit page replacing schema properties



To display the [Courier services] detail on the [Delivery] tab of the order edit page, add the following source code. In the *details* section, a new *CourierServiceDetail* 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*/{
            // [Courier services] detail setup.
            "CourierServiceDetail": {
                // Detail schema name.
```

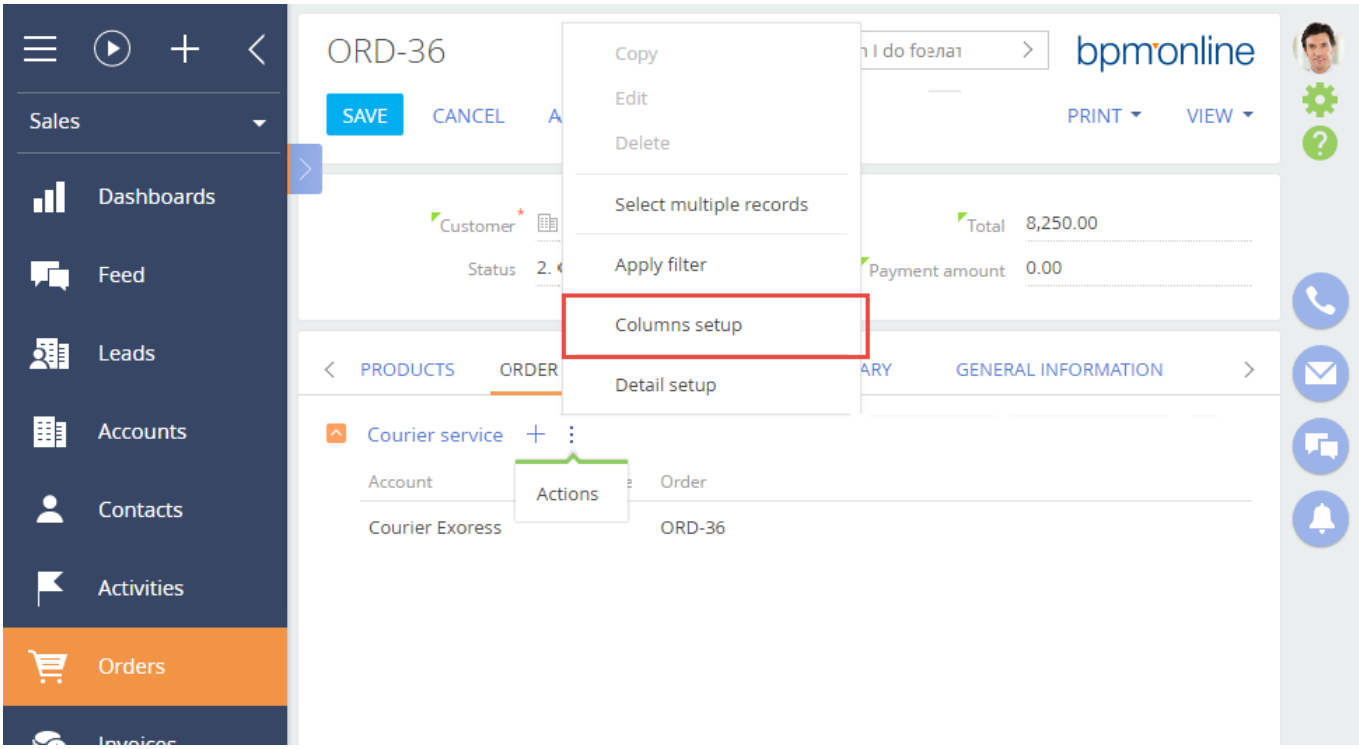
```
                "schemaName": "CourierServiceDetail",
                // Detail object schema name.
                "entitySchemaName": "CourierServiceInOrder",
                // Filtering displayed contacts for current order only.
                "filter": {
                    // Detail object schema column.
                    "detailColumn": "Order",
                    // 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": "CourierServiceDetail",
                "values": {
                    "itemType": 2,
                    "markerValue": "added-detail"
                },
                // Containers, where the detail is located.
                "parentName": "OrderDeliveryTab",
                "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.

4. 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 the columns to be displayed (Fig. 8).

Fig. 8. Detail actions menu



Adding a detail with a lookup

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 an editable list;
- details with a lookup.

For more information on details, please see the "**Details**" article in the "**Application interface and structure**" section.

A detail with a lookup not a standard bpm'online detail and can not be added to sections using the Detail Wizard. Below is a step-by-step procedure for adding a detail with editable list to an edit page.

General procedure for adding a detail with a lookup to an existing section

To add a detail with a lookup:

1. Create a detail object schema.
2. Create and configure the detail schema.
3. Set up the detail in a replacing section edit page schema.
4. Set up the detail fields.

Case description

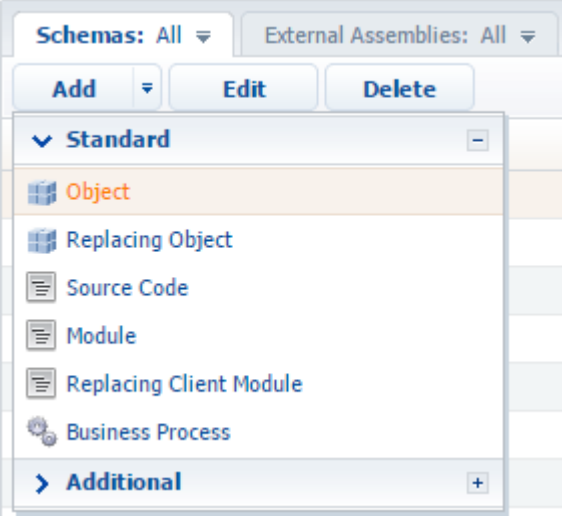
Create a custom detail [Acceptance certificates] on the [Delivery] tab of the [Orders] section record page. The detail must display a list of documents (Acceptance certificates) for the current order.

Case implementation algorithm

1. Create detail object schema

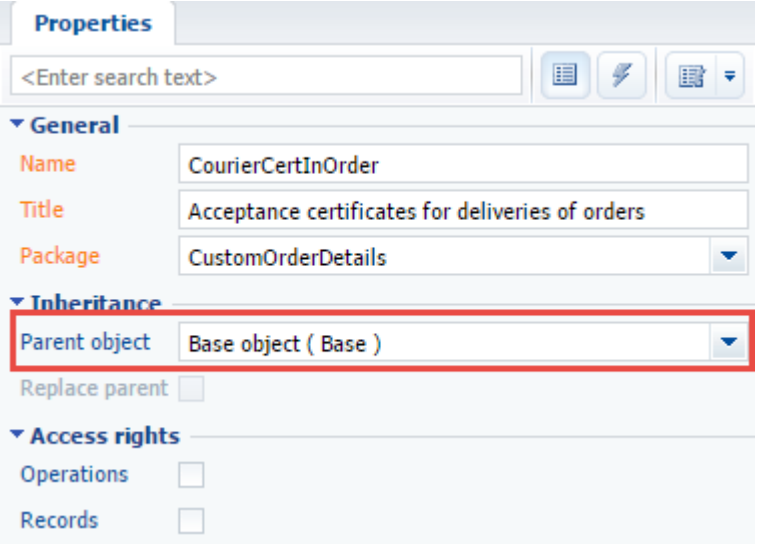
In the settings mode, go to the [Configuration] section, open the [Schemas] tab and execute the [Add] > [Object] menu command (Fig. 1).

Fig. 1. Adding a detail object schema



Select the base *BaseEntity* object (located in the *Base* package) as the parent for the created schema (Fig 2). The rest of the properties must be set up as shown in Fig. 2. In the [Package] property, the system will specify the current package name. For more information on setting schema properties in the Object Designer, please see "**Workspace of the Object Designer**" article.

Fig. 2. Setup of detail object schema properties



Add an [Order] lookup column which will connect the detail object with the section object, and a [Document] lookup column where the acceptance certificate document will be stored. Both columns must be made to be required. Column properties setup is shown in Fig. 3 and Fig. 4.

Fig. 3. Specifying the properties of the [Order] column

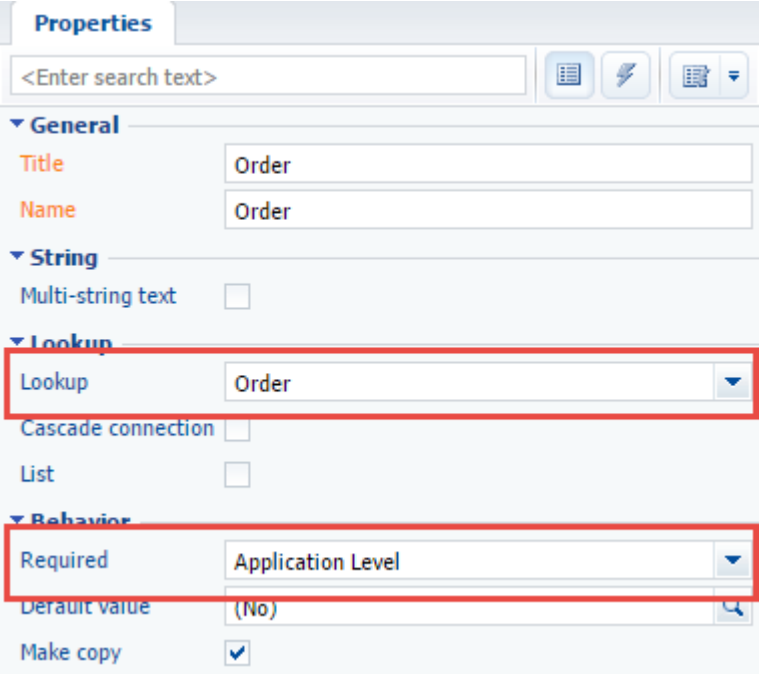
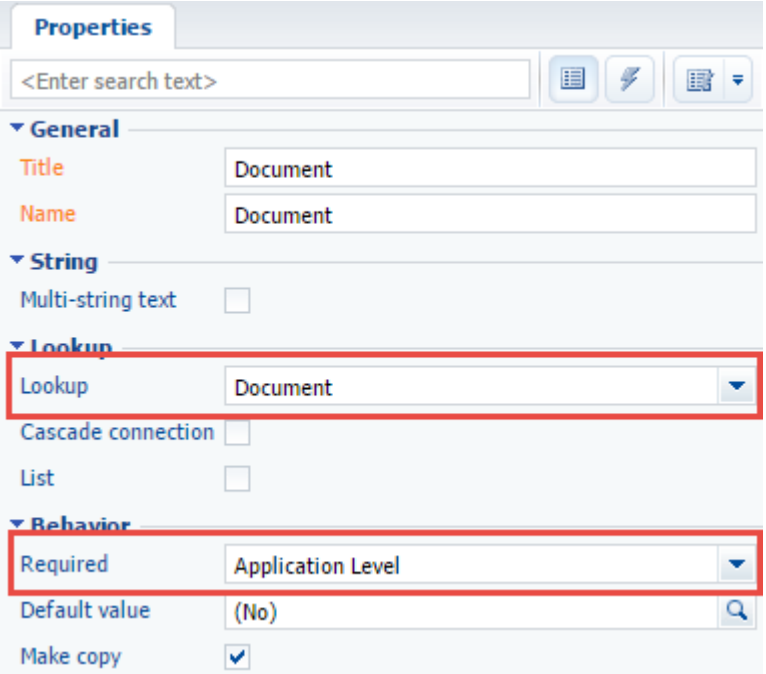


Fig. 4. Specifying the properties of the [Document] column



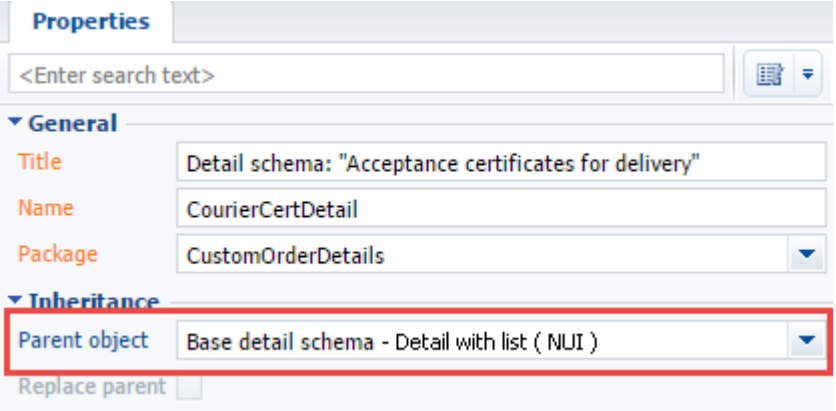
Object schema must be saved and published.

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).

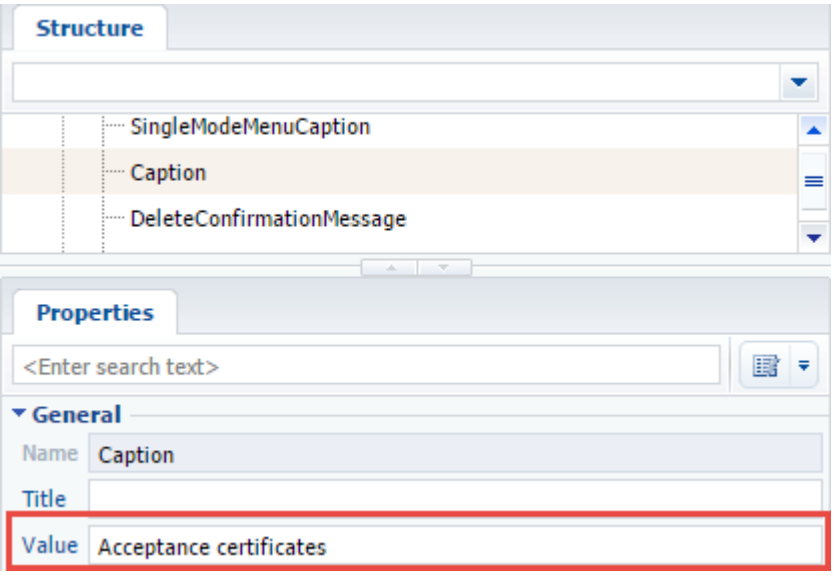
The new module must inherit the functionality of the *BaseGridDetailV2* base schema for details with lists, which is available in the *NUI* package. To do this, specify this schema as the parent for the created detail schema (Fig. 5). The rest of the properties must be set up as shown in Fig. 5. In the [Package] property, the system will specify the current package name.

Fig. 5. Detail schema properties



Set the *Acceptance certificates* value for the [Caption] localizable string of the detail schema (Fig. 6). The [Caption] string contains the detail title, shown on the edit page.

Fig. 6. Setting the value of the [Caption] string



To make the detail be filled in from a lookup, make the following changes to the detail schema:

1. Add dependencies from the ConfigurationEnums module.
2. Add the onDocumentInsert and onCardSaved event handler methods, openDocumentLookup lookup window method and auxiliary data management methods.
3. Add required configuration objects to the array of modifications.

Below is the detail schema source code (with comments), which must be added to the [Source code] section of the client module designer. The example below uses the Terrasoft.EntitySchemaQuery class to perform queries to the database. For more information on how to use EntitySchemaQuery, please see the "**The use of EntitySchemaQuery for creation of queries in database**" article in the "**Working with data**" section. Detail menu items for editing and copying records were disabled to simplify the case.

```
// Define schema and set its dependencies from other modules.
define("CourierCertDetail", ["ConfigurationEnums"],
    function(configurationEnums) {
        return {
            // Detail object schema name.
            entitySchemaName: "CourierCertInOrder",
            // Detail schema methods.
            methods: {
                //Returns columns, which were selected by query.
                getGridDataColumns: function() {
                    return {
                        "Id": {path: "Id"},
```

```
                        "Document": {path: "Document"},
                        "Document.Number": {path: "Document.Number"}
                    };
                },

                //Configures and displays lookup modal window.
                openDocumentLookup: function() {
                    //Configuration object
                    var config = {
                        // Object schema name, whose records are displayed in lookup.
                        entitySchemaName: "Document",
                        // Indicates whether multiple selection is available.
                        multiSelect: true,
                        // Columns that will be displayed in lookup
                        columns: ["Number", "Date", "Type"]
                    };
                    var OrderId = this.get("MasterRecordId");
                    if (this.Ext.isEmpty(OrderId)) {
                        return;
                    }
                    // Instance of the [EntitySchemaQuery] class.
                    var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
                        // Root schema setup.
                        rootSchemaName: this.entitySchemaName
                    });
                    // Adding the [Id] coluln.
                    esq.addColumn("Id");
                    // Adding the [Id] column of the [Document] schema.
                    esq.addColumn("Document.Id", "DocumentId");
                    // Creating and adding filters to the query collection.
                    esq.filters.add("filterOrder",
this.Terrasoft.createColumnFilterWithParameter(
                        this.Terrasoft.ComparisonType.EQUAL, "Order", OrderId));
                    // Getting the whole collection of records and displaying it in
the lookup modal window.
                    esq.getEntityCollection(function(result) {
                        var existsDocumentsCollection = [];
                        if (result.success) {
                            result.collection.each(function(item) {

existsDocumentsCollection.push(item.get("DocumentId"));
                            });
                        }
                        // Ading filter to 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 lookup modal window
                        this.openLookup(config, this.addCallBack, this);
                    }, this);
                },

                // Event handler for saving edit page.
                onCardSaved: function() {
                    this.openDocumentLookup();
                },
```

```
                // Opens document lookup if the order edit page was saved earlier.
                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 selected documents.
                addCallBack: function(args) {
                    // Instance of the BatchQuery batch request class.
                    var bq = this.Ext.create("Terrasoft.BatchQuery");
                    var OrderId = this.get("MasterRecordId");
                    // Collection of documents selected in the lookup.
                    this.selectedRows = args.selectedRows.getItems();
                    // Collection passed to query.
                    this.selectedItems = [];
                    // Copying 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 batch request if it is not empty.
                    if (bq.queries.length) {
                        this.showBodyMask.call(this);
                        bq.execute(this.onDocumentInsert, this);
                    }
                },

                //Returns query for adding current object.
                getDocumentInsertQuery: function(item) {
                    var insert = Ext.create("Terrasoft.InsertQuery", {
                        rootSchemaName: this.entitySchemaName
                    });
                    insert.setParameterValue("Order", item.OrderId,
this.Terrasoft.DataValueType.GUID);
                    insert.setParameterValue("Document", item.DocumentId,
this.Terrasoft.DataValueType.GUID);
                    return insert;
                },

                //Method called when adding records to the detail 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 selected detail records.
                deleteRecords: function() {
                    var selectedRows = this.getSelectedItems();
                    if (selectedRows.length > 0) {
                        this.set("SelectedRows", selectedRows);
                        this.callParent(arguments);
                    }
                },

                // Hide the [Copy] menu item.
                getCopyRecordMenuItem: Terrasoft.emptyFn,
                 // Hide the [Edit] menu item.
                getEditRecordMenuItem: Terrasoft.emptyFn,
                // Returns default column name for filter.
                getFilterDefaultColumnName: function() {
                    return "Document";
                }
            },
            // Array of modifications.
            diff: /**SCHEMA_DIFF*/[
                {
                    // Operation type — merge.
                    "operation": "merge",
                    // HName of the schema element, with which the action is
performed.
                    "name": "DataGrid",
                    // Object, whose properties will be combined wuth the shema
element properties.
                    "values": {
                        "rowDataItemMarkerColumnName": "Document"
                    }
                },
                {
                    // Operation type — merge.
                    "operation": "merge",
                    // Name of the schema element, with which the action is
performed.
                    "name": "AddRecordButton",
                    // Object, whose properties will be combined wuth the shema
element properties..
                    "values": {
                        "visible": {"bindTo": "getToolsVisible"}
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
```

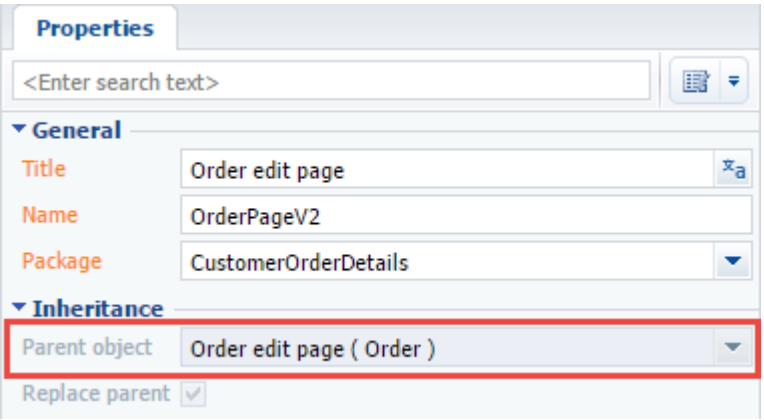
```
    }
);
```

Save the detail schema to apply the changes.

3. Set up the detail in the 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. 7). For more information on creating replacing schemas, please see the "**Creating a custom client module schema**" article.

Fig. 7. Order edit page replacing schema properties



To display the [Acceptance certificates] detail on the [Delivery] tab of the order edit page, add the following source code. In the *details* section, a new *CourierCertDetail* 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 added details of the edit page.
        details: /**SCHEMA_DETAILS*/{
            // Setup of the [Acceptance certificates] detail.
            "CourierCertDetail": {
                // Detail schema name.
                "schemaName": "CourierCertDetail",
                // Detail object schema name.
                "entitySchemaName": "CourierCertInOrder",
                // Filtering contacts displayed 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 [Acceptance certificates] detail.
            {
                "operation": "insert",
                // Detaail name.
                "name": "CourierCertDetail",
                "values": {
                    "itemType": 2,
                    "markerValue": "added-detail"
                },
```

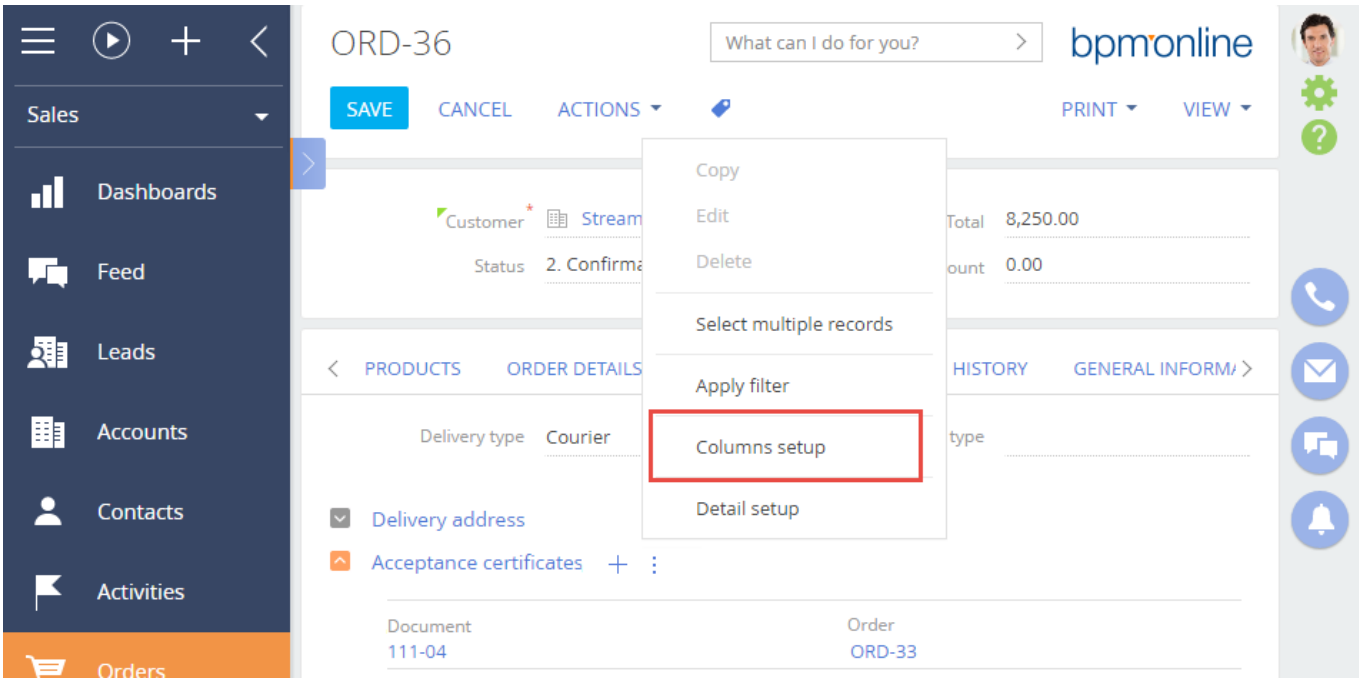
```
                // Containers where detail is places.
                "parentName": "OrderDeliveryTab",
                "propertyName": "items",
                // Index in the list of added elements.
                "index": 3
            }
        ]/**SCHEMA_DIFF*/,
        methods: {},
        rules: {}
    };
});
```

To apply the changes, the replacing page schema must be saved.

4. 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 the columns to be displayed (Fig. 8).

Fig. 8. Detail actions menu



Adding multiple records to a detail

Introduction

By default, a detail allows you to add only one record at a time. Adding multiple records 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 detail schemas. It provides an opportunity to add multiple lookup records to a detail at the same time.

The sequence of adding the multiple records selection functionality to a detail:

1. Create a replacing detail schema

2. Use mixin methods instead of base detail methods.

Case description

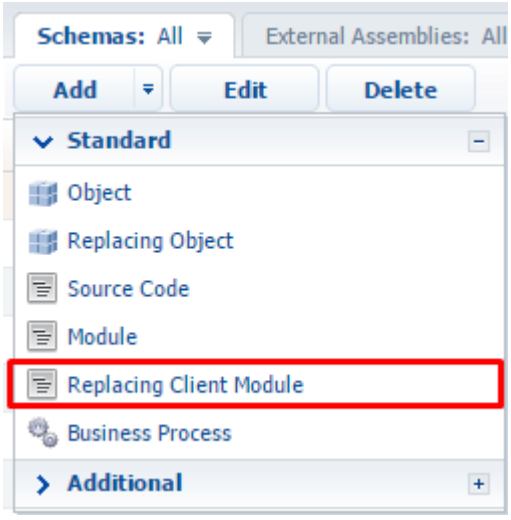
Implement multiple record selection in the [Contacts] detail on the [Opportunities] section records edit page.

Case implementation algorithm

1. Create a replacing view model schema of a detail

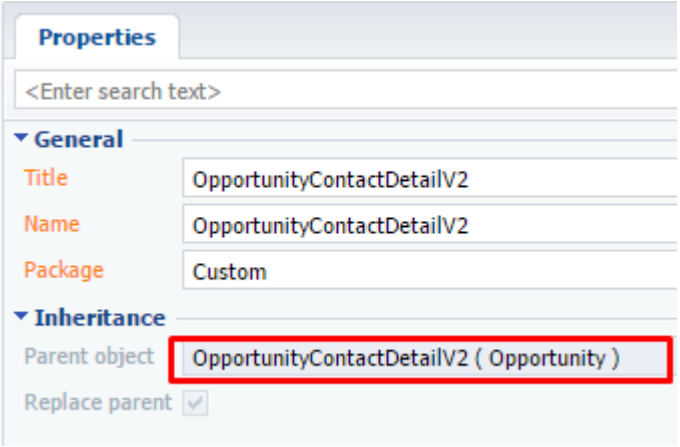
To do this, create a replacement client module in a custom package (Fig. 1). The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**" article.

Fig. 1. Adding a replacing client module



Select the *OpportunityContactDetailV2* schema as the parent object

Fig. 2. Adding replacing client module



2. Use mixin methods instead of base detail methods.

To use the *LookupMultiAddMixin* in a schema, add it to the *mixins* property and initialize it in the overridden *init()* schema method. Learn more about overriding the *init()* method in the "**Modular development principles in bpm'online**" article.

You will also need to override 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 *openLookupWithMultiSelect()* method for opening the help window for multiple selection and the associated *getMultiSelectLookupConfig()* method which configures the help window, is used. The description and parameters of these methods are given in Table 1.

Table 1. Methods for opening 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 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.

The source code of the detail view model 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 (with cache cleaning), 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 [Sales] section record edit page.

Fig. 3. Adding multiple records to a detail

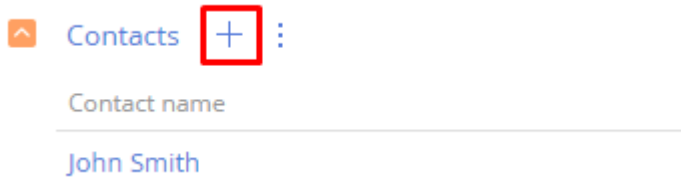


Fig. 4. Selecting necessary records from a lookup

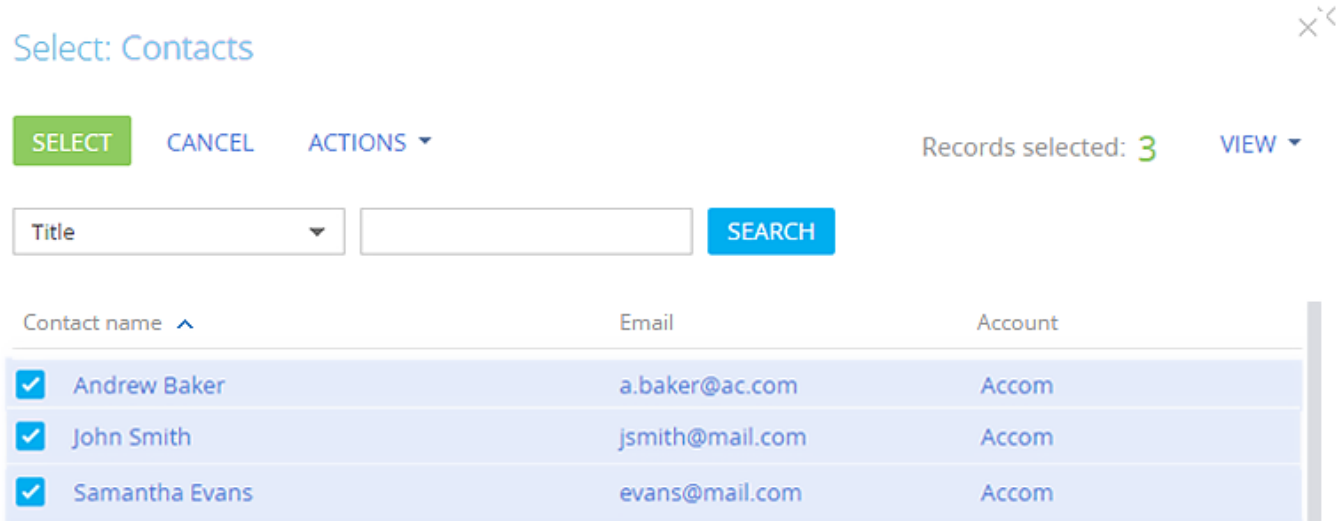
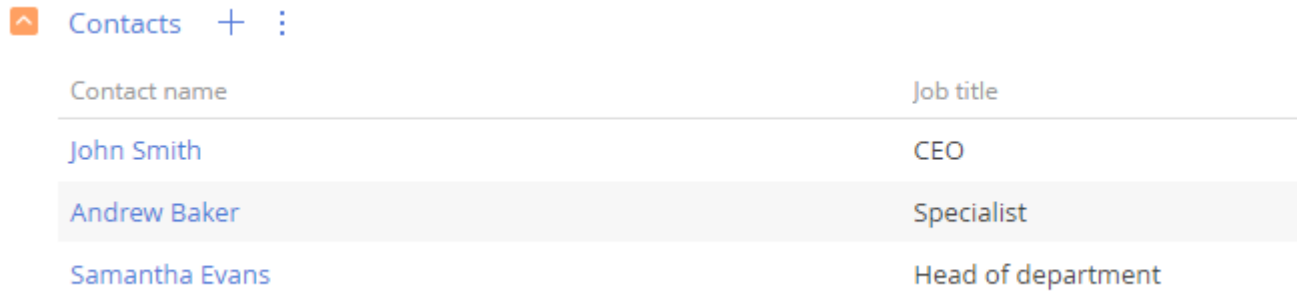


Fig. 5. Case result: all selected records are added to the detail



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.

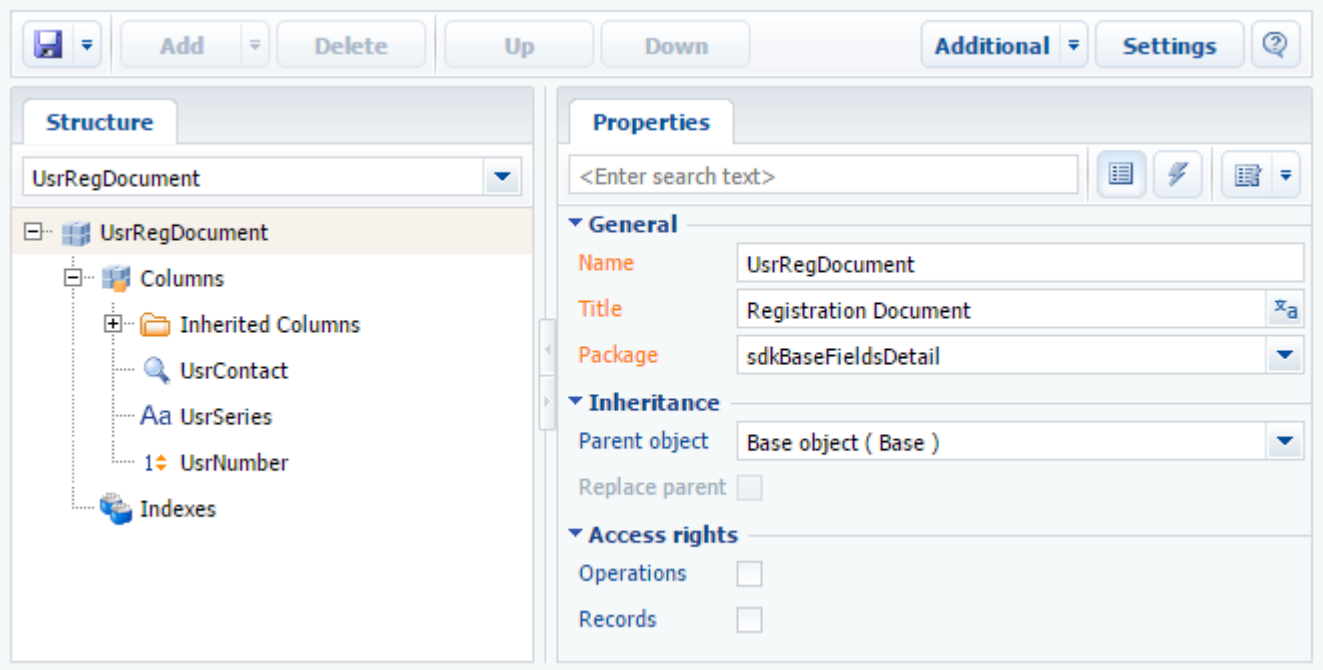
Case implementation algorithm

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

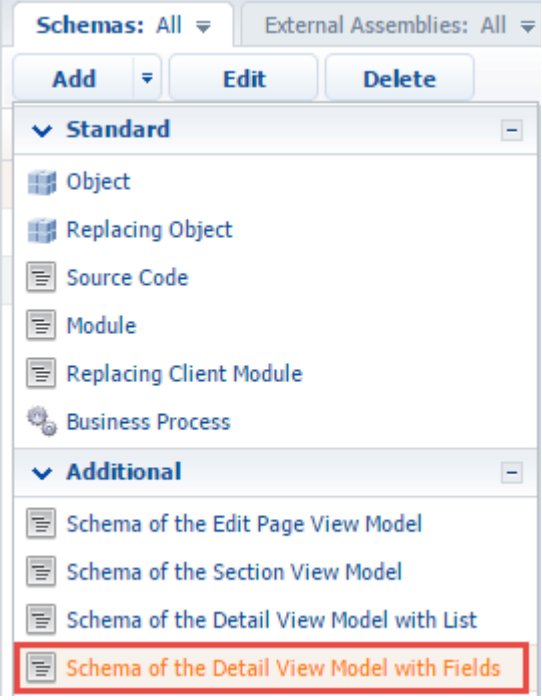
Title	Name	Data Type
Contact	UsrContact	Lookup
Series	UsrSeries	Text (50 characters)
Number	UsrNumber	Integer

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).

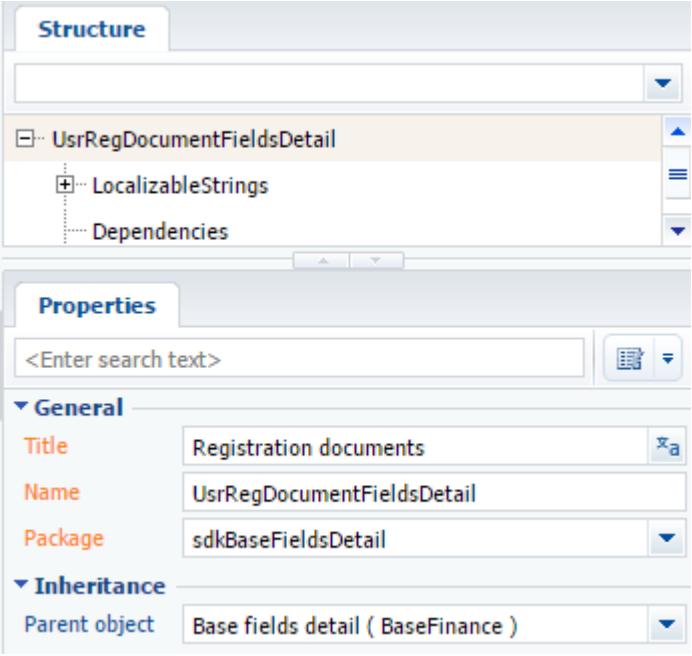
Fig. 2. Adding a custom view model schema for the custom detail with fields



Property values for the created schema (Fig. 3):

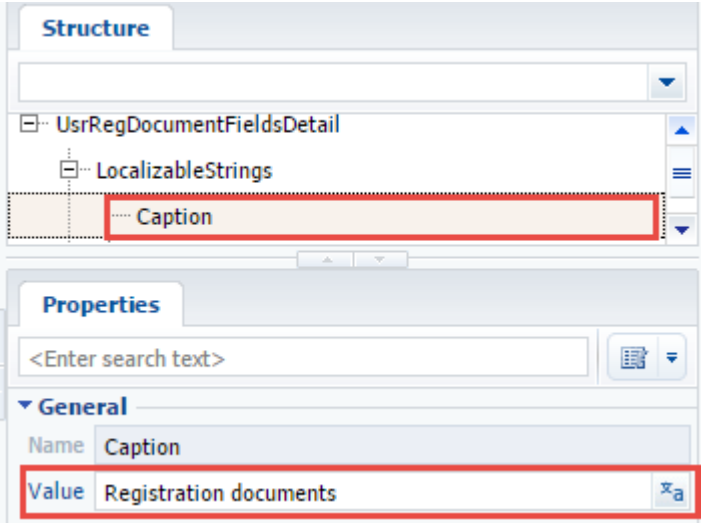
- [Title] – "Registration documents".
- [Name] – "UsrRegDocumentFieldsDetail".
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down 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"];
                }
            }
        };
    });
```

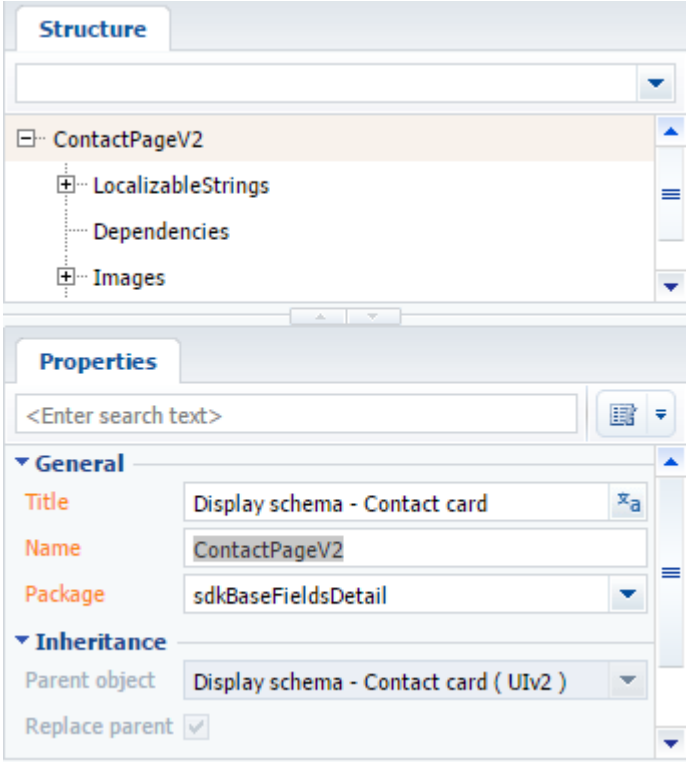
Save the schema to apply changes.

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:

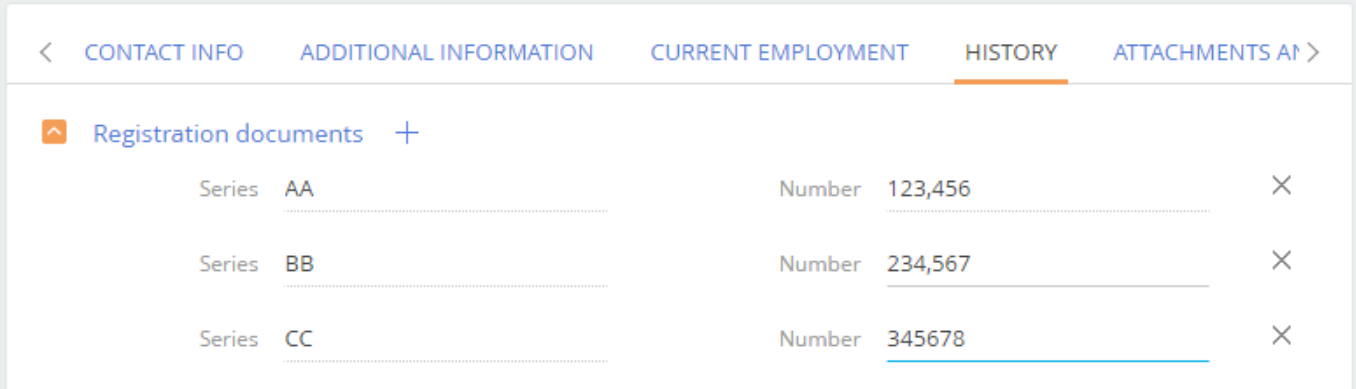
```
define("ContactPageV2", [], function() {
    return {
        entitySchemaName: "Contact",
        details: /**SCHEMA_DETAILS*/ {
            // Adding a field with details.
            "UsrRegDocumentFieldsDetail": {
                // Name of the custom detail schema.
                "schemaName": "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

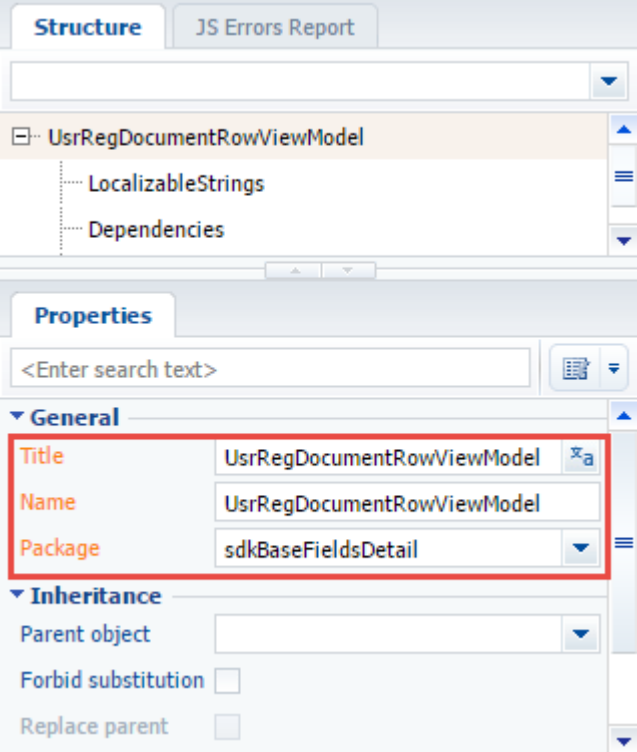
1. Create a module schema and define record view styles

> 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

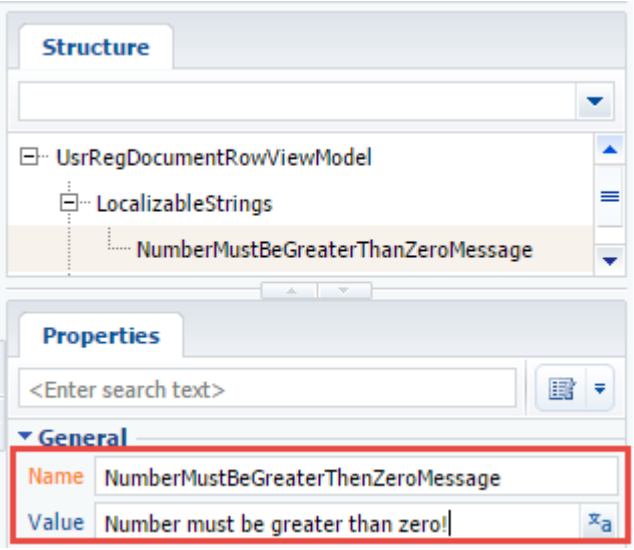
1. Adding a localizable string with the error message

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

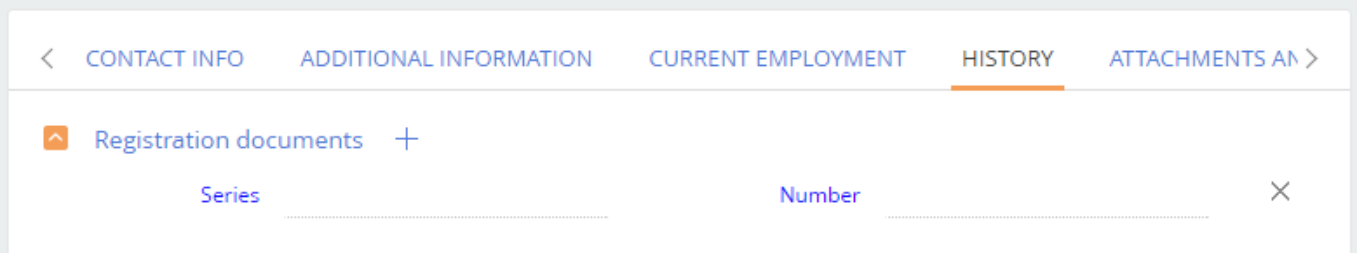
When a detail is loaded, adding virtual records enables you to display a field edit card immediately, without pressing the [Add] button.

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.

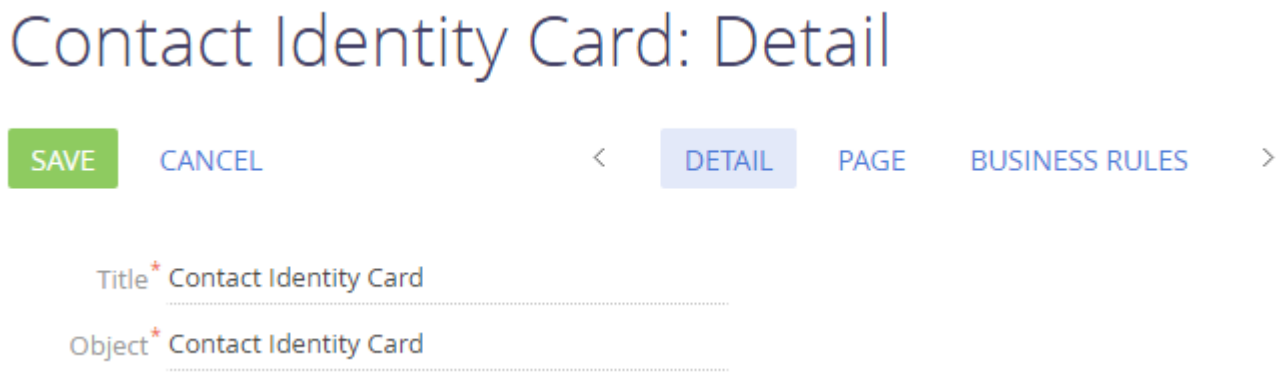
2. Creating a schema of the detail list client module and a schema of the detail edit page.

> 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.

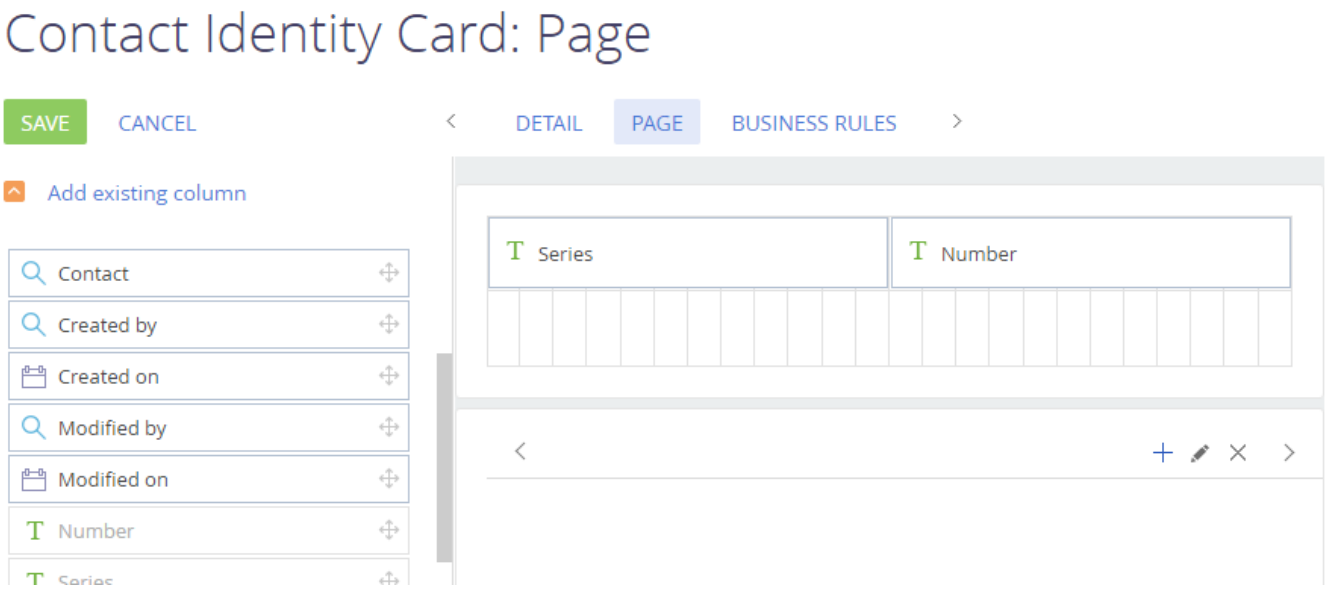
On the [DETAIL] step, specify the title of the detail and select the main detail object (Fig. 1).

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



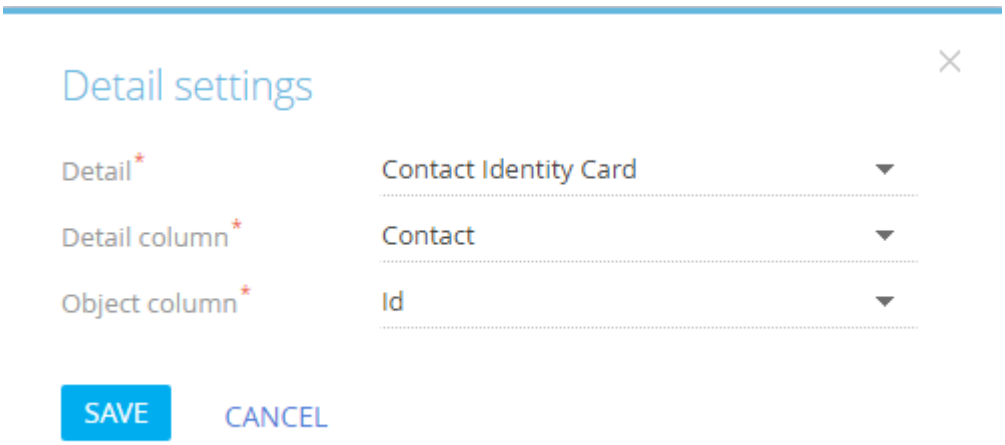
Save the detail when the setup is done.

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

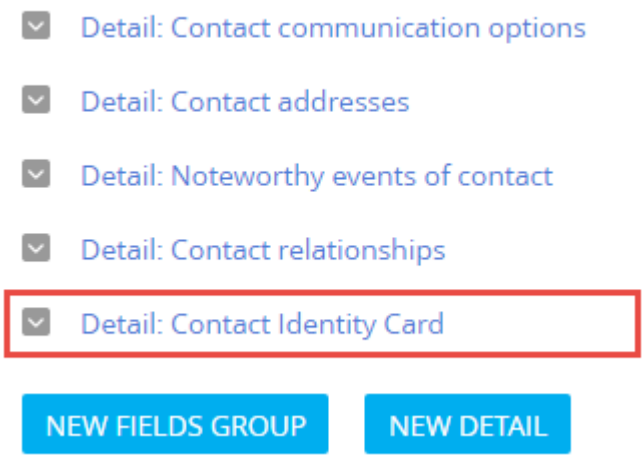
Open the detail wizard in the [Contacts] section, and select [NEW DETAIL] on the [PAGE] step. In the opened window, select the [Contact's ID] detail and configure the connection between detail object columns and the section object (Fig. 3).

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

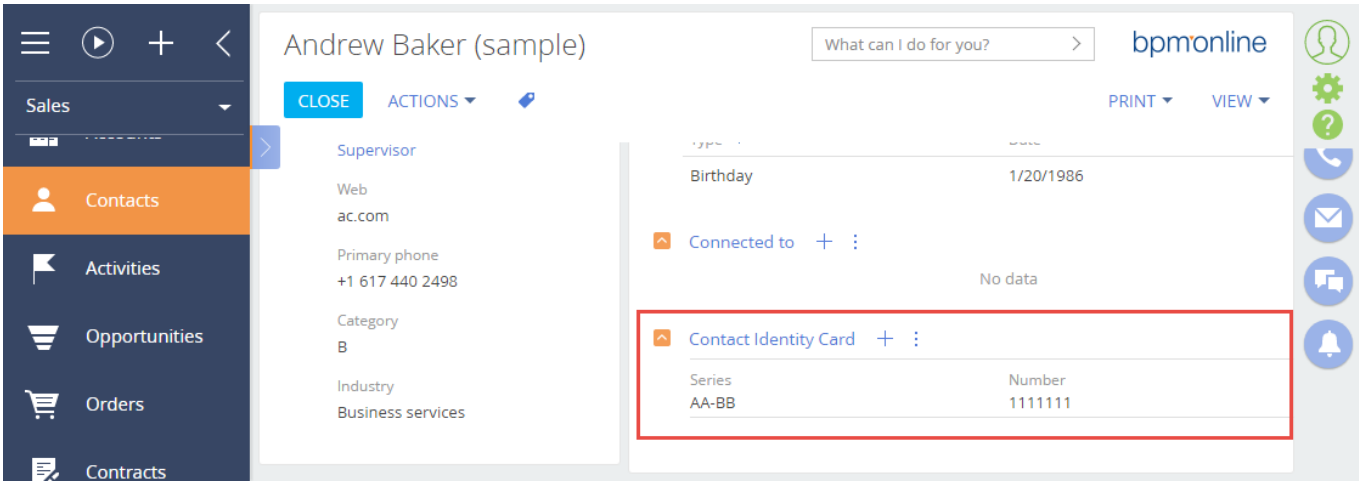


Save the changes when the section record page setup is done.

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.

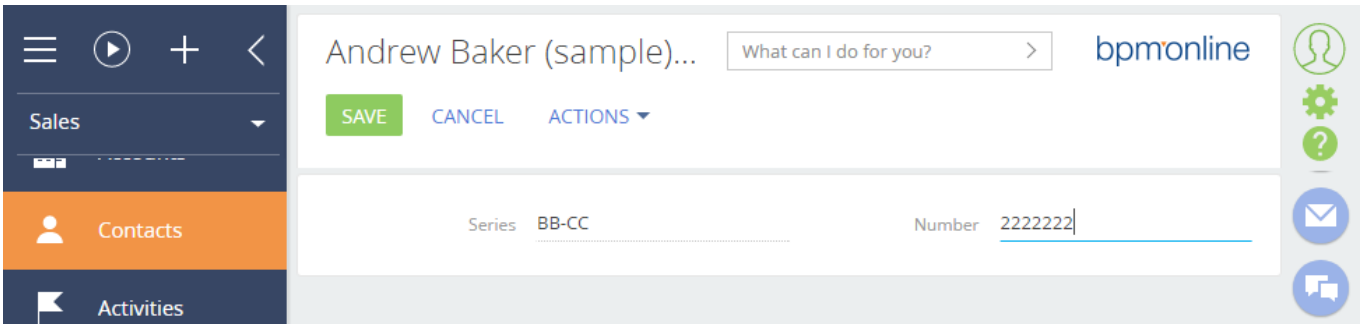
Fig. 5. Case result



ATTENTION

It is necessary to configure the columns in a detail menu, and add a few records to see the result.

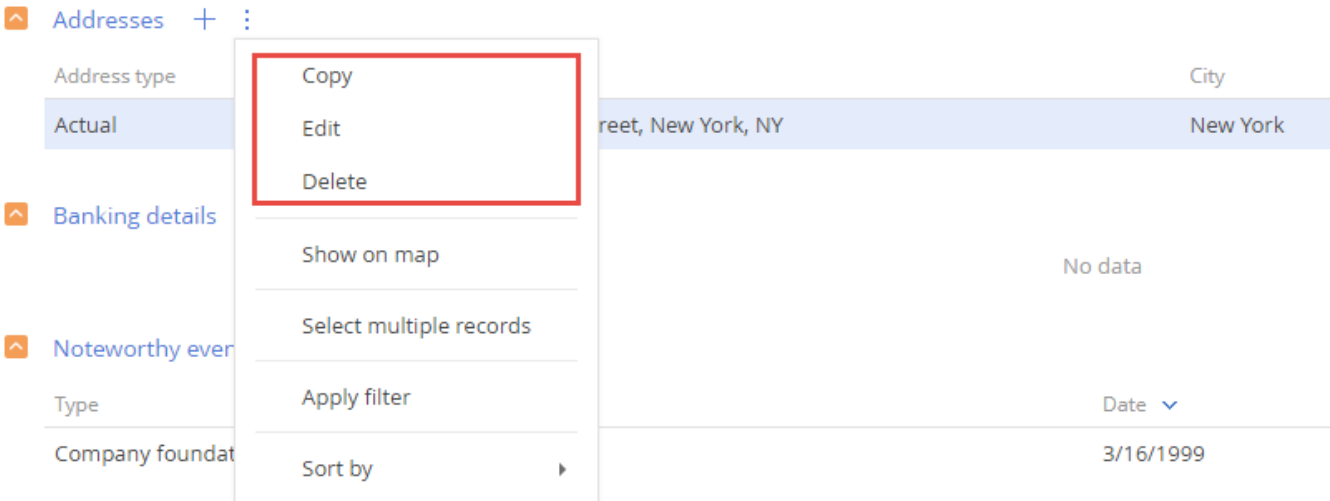
Fig. 6. Adding a record to a detail



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*/
    };
```

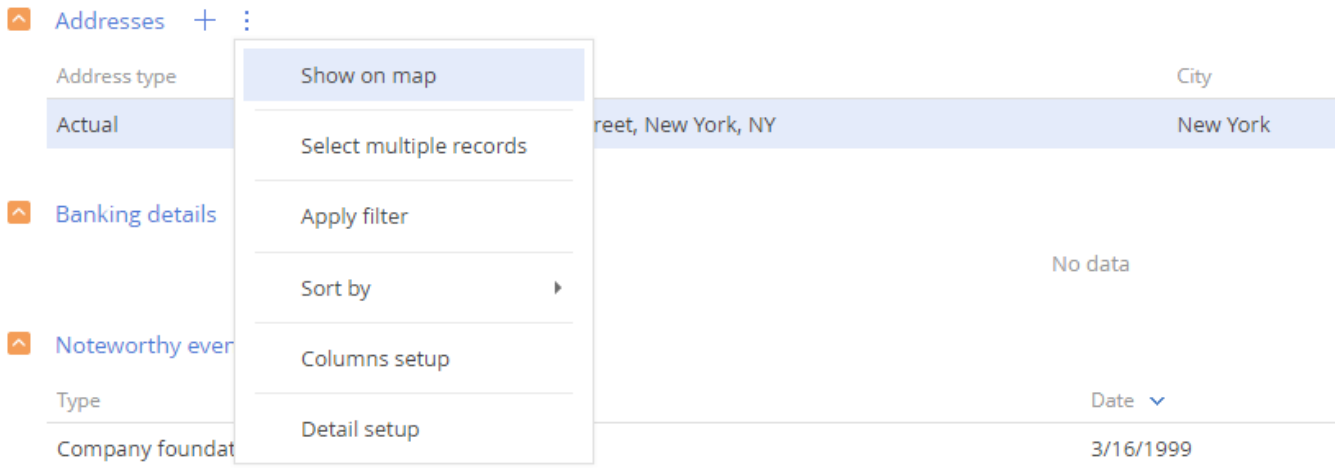
```
});
```

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

- **How to add auto-numbering to the edit page field**
- **How to run a process from client module**
- **Creating custom [User task] process element**

How to add auto-numbering to the edit page field

One of the typical configuration tasks is the setup of auto-numbering for a specific field of the edit page. The examples may be the [Documents], [Invoice], and [Contracts] sections where a newly created document receives an automatically generated number of a certain format.

To implement the auto-numbering task, we recommend to override the ***onEntityInitialized*** base virtual method and call the ***getIncrementCode*** method of the BasePageV2 card base schema in it.

The getIncrementCode method has two parameters:

- [*callback*] — a function to be called after a response from the server. The server response must be passed to the field which requires assigning the automatically generated value;
- [*scope*] — the context where the [callback] function will be called (unrequired parameter).

To use the auto-numbering mechanism, two system settings must be added:

- [Entity]CodeMask — an object number mask,
- [Entity]LastNumber — the current object number,

where [Entity] — the name of the object for whose column the auto-numbering will be used, for example, InvoiceCodeMask (Invoice number mask) and InvoiceLastNumber (Current invoice number).

For the auto-numbering to operate, not only on the client side but also on the server side (for example, to implement tasks for the bpm'online system integration with other systems), the [Before Record Adding] event handler must be added to the object for whose column the auto-numbering will be used. The following number generation parameters must be set in the business process. Specify the object schema for which the generation will be performed. Then, call the [Generate Number of Order] action. Finally, pass the generated value to the required column of the object.

Procedure for adding the auto-numbering:

1. Add system settings for the object number mask and current object number.
2. Override the onEntityInitialized base virtual method and call the getIncrementCode base method, specifying the field for which the number will be generated.
3. Add the [Before Record Adding] event handler to the object.

Below is the example of adding the auto-numbering to the edit page field.

Case description

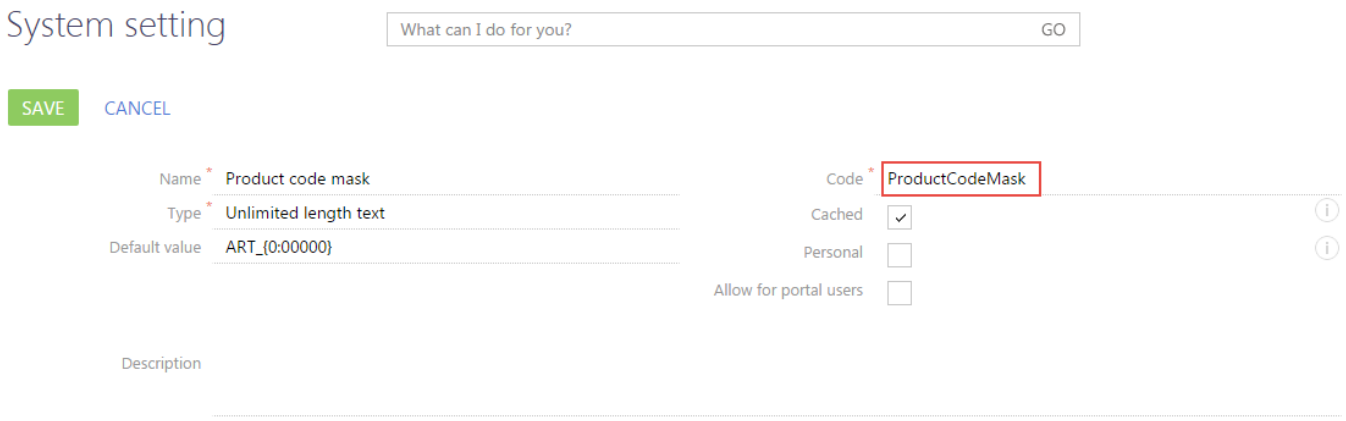
Set the auto-generation in the [Products] section for the [Code] field of the edit page. The product code must be formed as follows — ART_00000.

Case implementation algorithm

1. Create two system settings

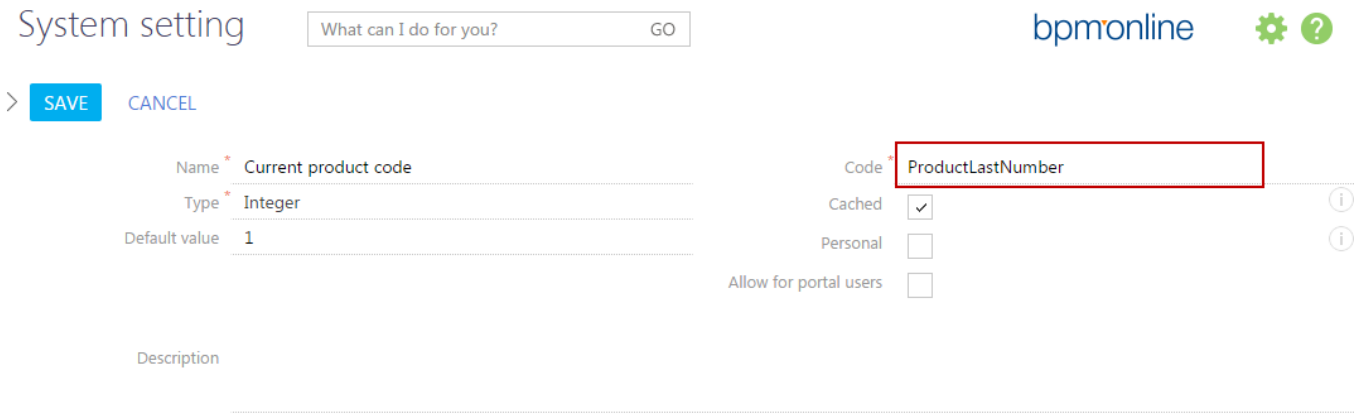
Create the ProductCodeMask system setting with the number mask "ART_{0:00000}" (Fig. 1).

Fig. 1. — [Product code mask] system setting



Create the ProductLastNumber system setting (Fig. 2).

Fig. 2. — [Current product code] system setting

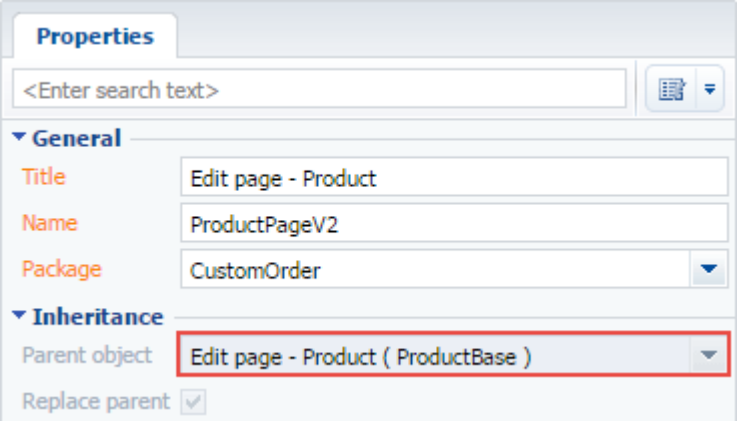


2. Create the [Products] replacement edit page in a custom package

A replacement client module must be created and the [Edit Page - Product] must be specified as the parent object (Fig. 3).

The procedure for creating a replacement page is described in the article **Creating a custom client module schema**.

Fig. 3. — Properties of the [Products] replacement edit page



3. Override the onEntityInitialized method in the collection of methods of the edit page view model

Call the getIncrementCode method in the onEntityInitialized method and specify a column for the [Code] auto-generation in its callback function.

```
define("ProductPageV2", [],
    function () {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Product",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            diff: /**SCHEMA_DIFF*/[],/**SCHEMA_DIFF*/
            // Methods collection of the view model page.
            methods: {
                // Redefining the Terrasoft.BasePageV2.onEntityInitialized Base
method which actuates
                // upon the completion of the initialization of the edit page object
schema.
                onEntityInitialized: function () {
                    // The code is generated only if a new item is generated or the
existing item is copied.
                    if (this.isAddMode() || this.isCopyMode()) {
                        // Calling the Terrasoft.BasePageV2.getIncrementCode base
method which generates a number
                        // based on the pre-set mask.
                        this.getIncrementCode(function (response) {
                            // The generated number is returned to the [Code] column.
                            this.set("Code", response);
                        });
                    }
                    // The parent implementation of the method is called
onEntityInitialized.
                    this.callParent(arguments);
                }
            }
        };
    });
```

4. Add the [Before Record Adding] event handler of the object

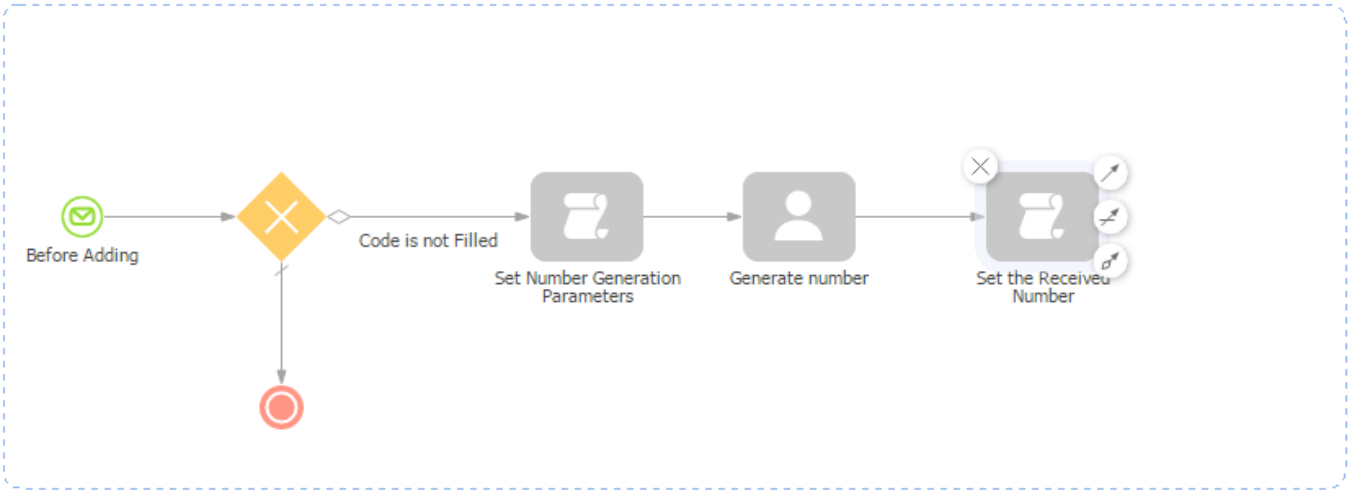
For this purpose, the [Product] replacement object must be created:

- Select a custom package and run the menu item [Add] > [Replacing Object] on the [Schemas] tab.

- Fill in properties of the new object by specifying [Product] as the parent object.

The [Before Record Adding] event handler must be added to the created object, in the list of events (Fig. 4).

Fig. 4. — Event sub-process — [Before Record Adding] event handler



Items of the [Before Adding] event sub-process:

1. [Before Adding] start message — this sub-process will be started when the [ProductInserting] message is received.

2. [Exclusive OR] Gateway from which two flows of the sub-process come:

- Default flow will be activated only if the passage by the conditional flow cannot be performed. This leg is finished by the [Stop] event .
- [Code is not Filled] conditional flow, wherein the the [Code] column is verified. Further implementation of the sub-process is possible only if it is filled in:

```
// Code to be added to the [Condition] field of the conditional flow.
string.IsNullOrEmpty(Entity.GetTypedColumnValue("Code"))
```

3. Script task [Script task [Set Number Generation Parameters] — the C# script software code will be executed in it:

```
// Setting the schema for a number generation.
// GenerateNumberUserTask is the [Generate Number] action.
GenerateNumberUserTask.EntitySchema = Entity.Schema;
return true;
```

4. [Generate Number] process action — the [GenerateNumberUserTask] system action which directly generates a current sequence number in accordance with the set mask.

5. Script task [Set the Received Number] — the C# script software code will be executed in it:

```
// The value generated by the GenerateNumberUserTask system action
// is recorded into the [Code] column.
Entity.SetColumnValue("Code", GenerateNumberUserTask.ResultCode);
return true;
```

5. Save the replacement page schema and publish the replacement object.

When the schema is saved, the object is published and the system web-page is updated, a generated code will be displayed automatically on the product edit page when adding a new product to the [Code] field (Fig. 5).

Fig. 5. – Result demonstration

How to run a process from client module

Running process from configuration schema code

In order to run a business process in JavaScript code, take the following actions:

1. Connect *ProcessModuleUtilities* module to a module of the page, from which the service is called. This module provides a convenient interface for execution of queries to the ProcessEngineService.svc service.
2. Call *executeProcess(args)* method from *ProcessModuleUtilities* module by transferring *args* object with such properties (table 1):

Table 1. — args object properties

Property	Description
sysProcessName	A name of called process (this property is optional if sysprocessid is determined).
sysProcessId	A unique identifier of called process (this property is optional if sysprocesname is determined).
parameters	An object, properties of which are represented by input parameters of called process.

Example of process running from configuration schema

Add actions, through which the "Holding a meeting" business process will run, to the account edit page. The main account contact will be transferred to the business process as a parameter.

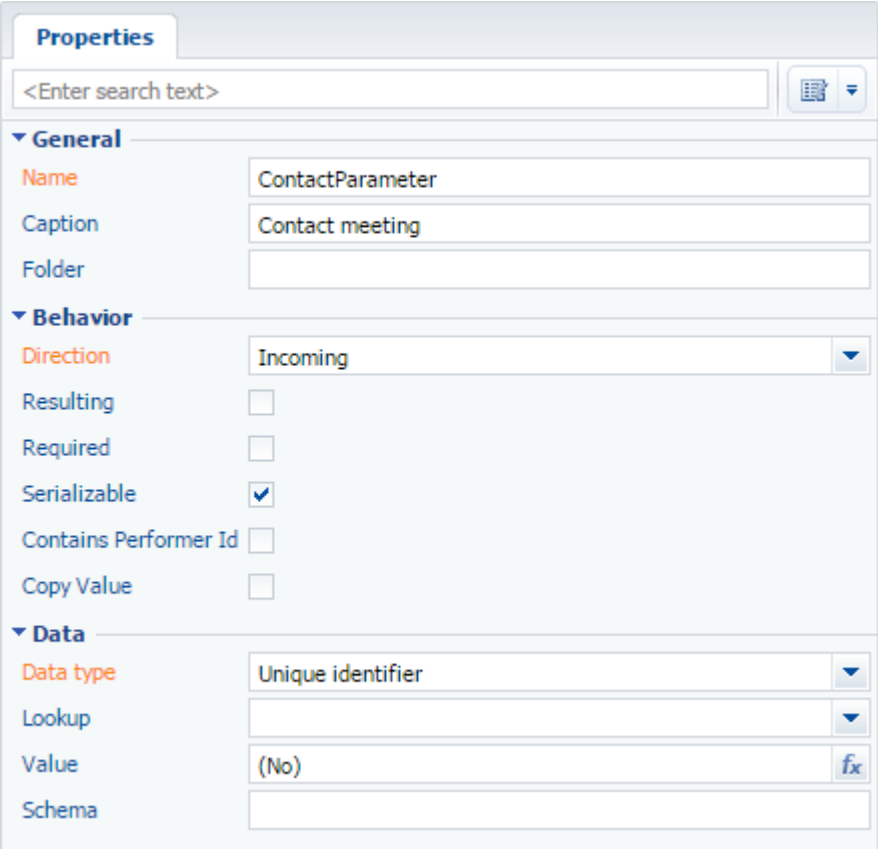
Case implementation algorithm

1. Create the "Holding a meeting" user business process

The example uses a business process, creation of which is described in the Designing a linear process.

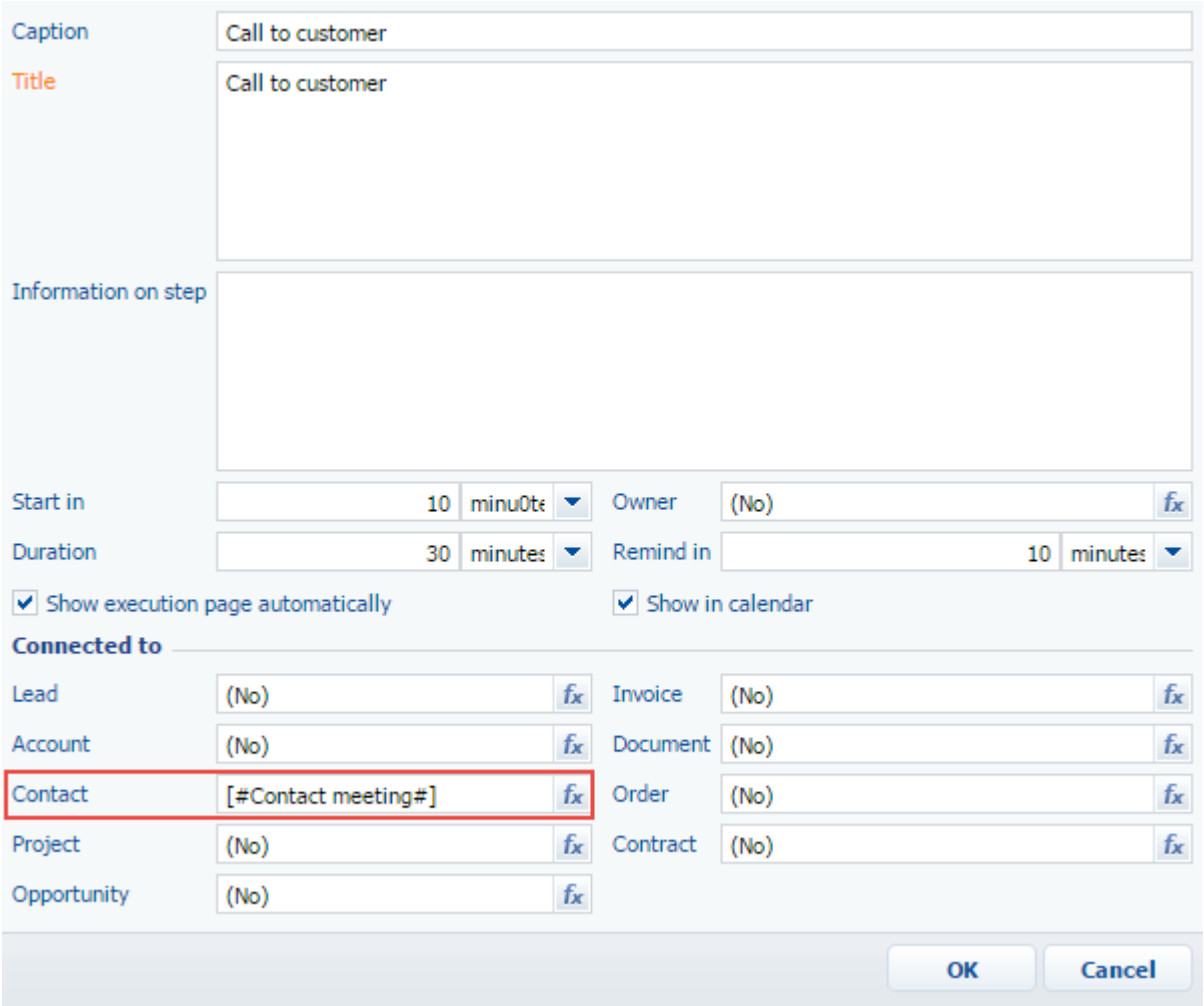
Once a business process is created, add the ContractParameter input parameter, in the properties of which the *Unique identifier* identifier is specified in the [Data type] field (figure 1).

Figure 1. — Adding of input parameter



Then fill in the [Contact] field with a given process input parameter in the properties of the first actions of the [Call to customer] process (Figure 2).

Figure 2. — Parameter transferring to the process item



2. Create replacing account edit page and add action to it

The process of adding an action to an edit page is described in the **Adding an action to the edit page** article.

Add *CallProcessCaption* localizable string with action title, for example, "Schedule a meeting" to the schema of replacing account edit page module.

Connect *ProcessModuleUtilities* module in declaration of edit page module as a dependency.

3. Add action handler and call process running method in it

To run a process, use executeProcess method of *ProcessModuleUtilities* module and transfer the object with the following properties into it: name of created business process and object with initialized input parameters for the process.

Source code of replacing module of edit page is represented below.

```
define("AccountPageV2", ["ProcessModuleUtilities"],
    function (ProcessModuleUtilities) {
        return {
            // Name of the schema of edit page object.
            entitySchemaName: "Account",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            // Collection of methods of view model of edit page.
            methods: {
                // The method verifies whether the [Main contact] field of the page
is completed.
                isAccountPrimaryContactSet: function () {
                    return this.get("PrimaryContact") ? true : false;
```

```
                },
                 // Overridding of basic virtual method that returns collection of
edit page actions
                getActions: function () {
                    // Parent implementation of the method is called for
                    // for getting collection of initialized actions of basic page
                     var actionMenuItems = this.callParent(arguments);
                    // Adding separator line for visual separation of user action
from the list of
                    // actions of basic page.
                    actionMenuItems.addItem(this.getActionsMenuItem({
                        Type: "Terrasoft.MenuSeparator",
                        Caption: ""
                    }));
                    // Adding [Schedule meeting] menu item to the lists of actions of
edit page.
                    actionMenuItems.addItem(this.getActionsMenuItem({
                        // Binding of menu item title to localizable schema string.
                        "Caption": { bindTo: "Resources.Strings.CallProcessCaption"
},
                        // Binding of action handler method.
                        "Tag": "callCustomProcess",
                         // Binding of visibility property of menu item to the value
that returns isAcciuntPrimaryContactSet method.
                        "Visible": { bindTo: "isAccountPrimaryContactSet" }
                    }));
                    return actionMenuItems;
                },
                // Method -handler of action selection.
                callCustomProcess: function () {
                    // Getting identifier of account main contact.
                    var contactParameter = this.get("PrimaryContact").value;
                    // The object that will be transferred to execureProcess method
as an argument.
                    var args = {
                        // Name of the process that should be run.
                        sysProcessName: "CustomProcess",
                        // Object with the value of ContractParameter input parameter
for CustomProcess parameter.
                        parameters: {
                            ContactParameter: contactParameter
                        }
                    };
                    // Running of user business process.
                    ProcessModuleUtilities.executeProcess(args);
                }
            }
        };
    });
```

4. Save schema in the configuration

Case execution result is shown on Fig. 3 and Fig. 4.

Figure 3. — Calling of business process upon action on the edit page

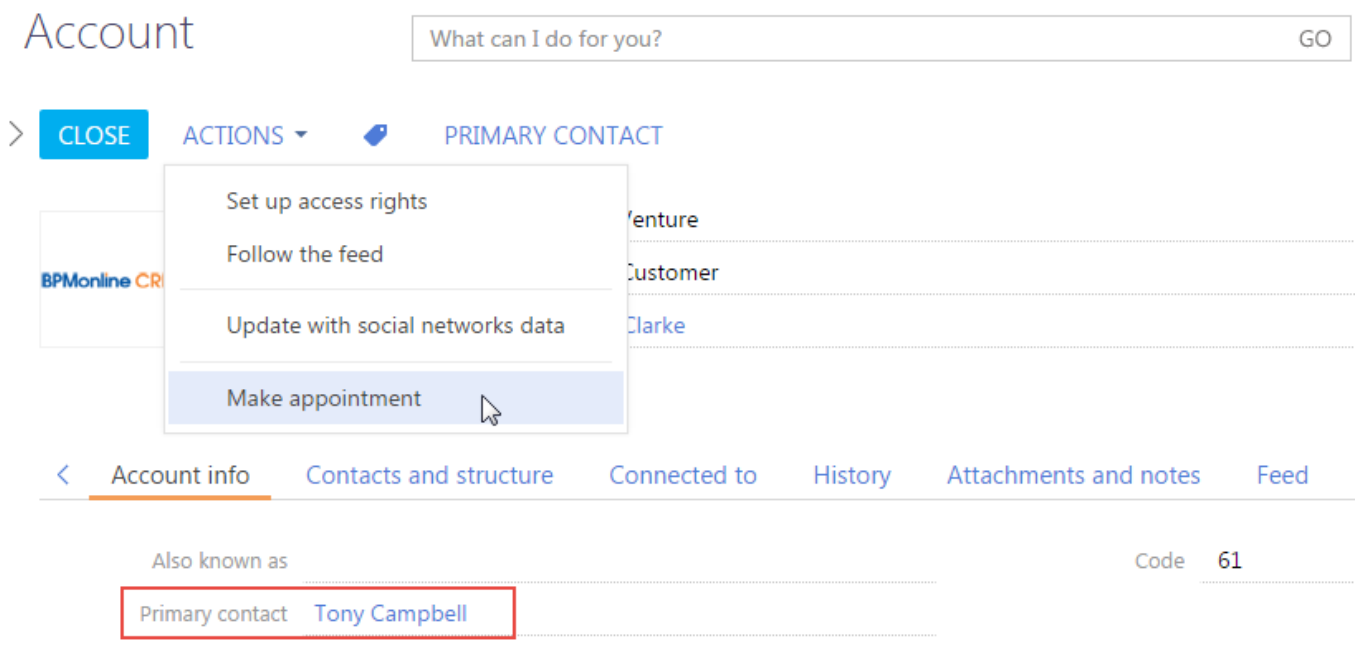
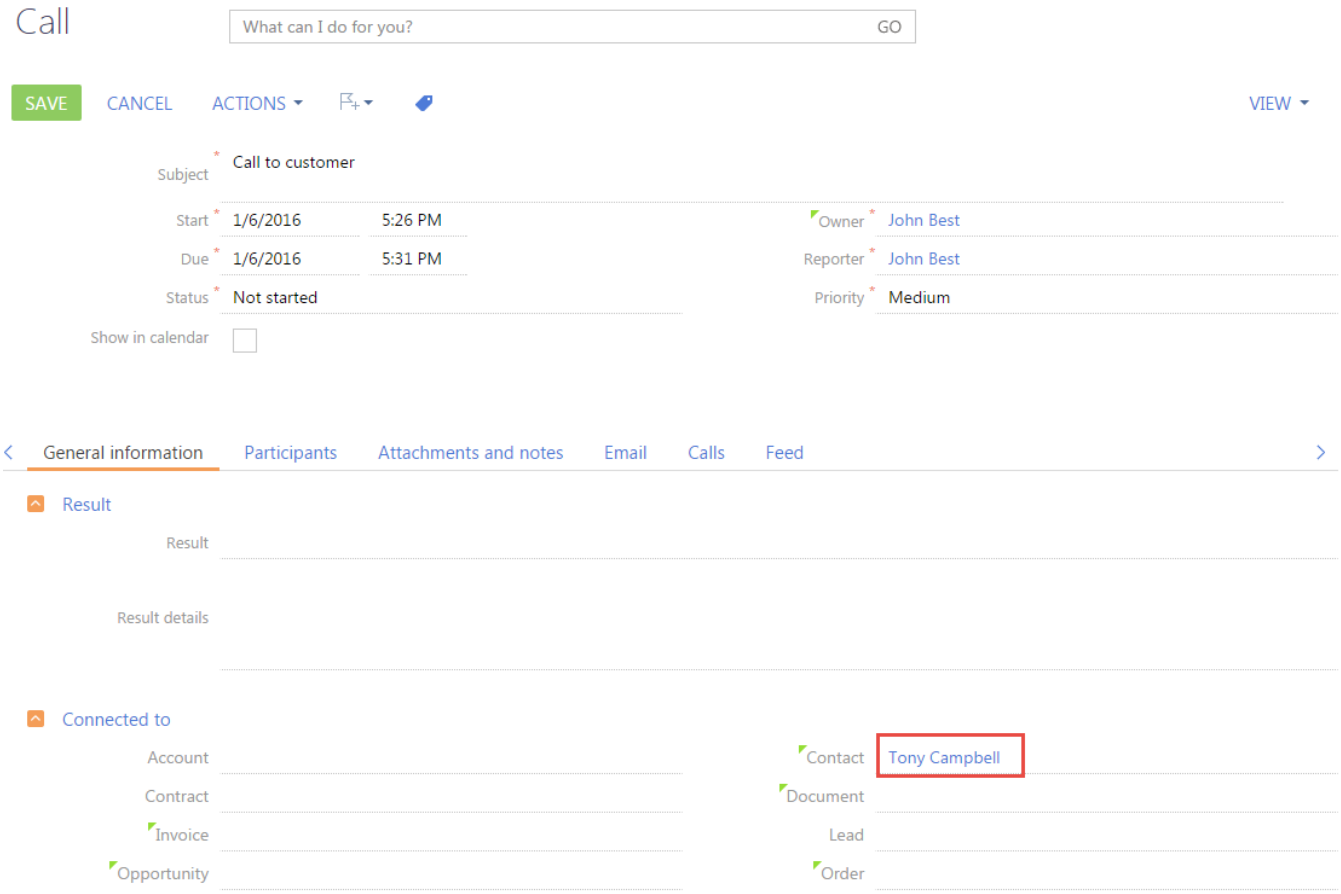


Figure 4. — Business process running result. Transferring of parameter for account edit page to business process



Creating custom [User task] process element

Introduction

It often becomes 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 custom 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 have the said task.

Case of creating a user task

Case description

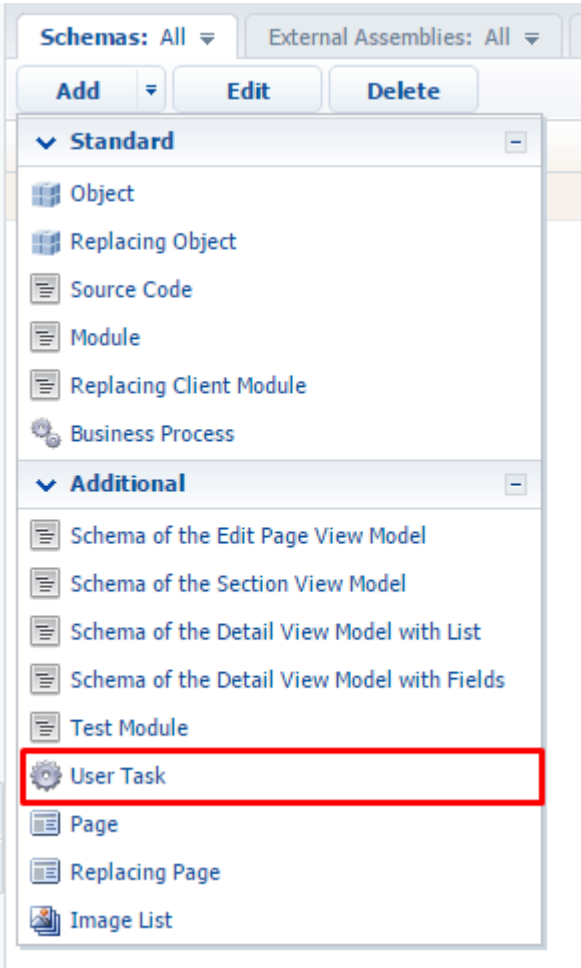
Create a simple process user task that will calculate the sum of two numbers. Use two numbers (specified as task parameters) to calculate the sum.

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



A user task designer window will open.

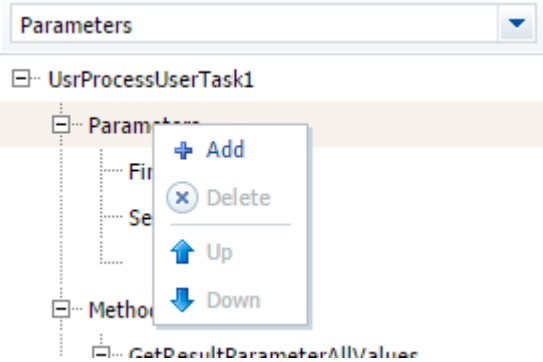
The default values for [Name] and [Title] are "UsrProcessUserTask1" and "User Task 1", respectively.

> ⚠ ATTENTION
>
> If the process task schema is added to a custom package, which is not the current one, you must add the package with the created schema to the dependencies of the current package.
>
> The default package is *Custom*. Use the [Current package] system setting to view and change the current package.

2. Add task parameters.

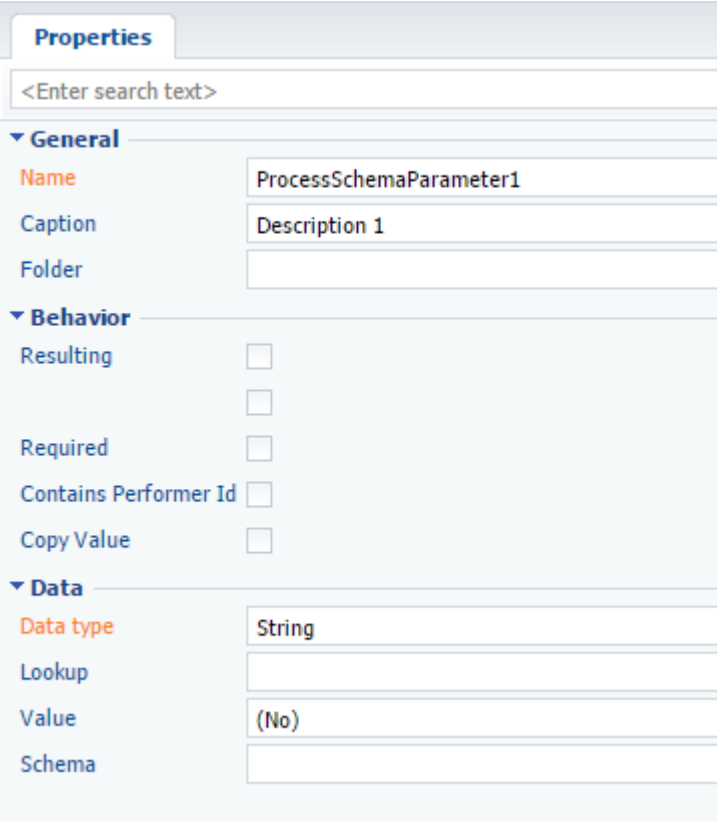
On the [Activity] tab of the process task designer, open the context menu on the [Parameters] element to add task result parameters. Execute the [Add] context menu command (Fig. 2).

Fig. 2. Create a process task parameter



A new parameter will be added as the result, and 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).

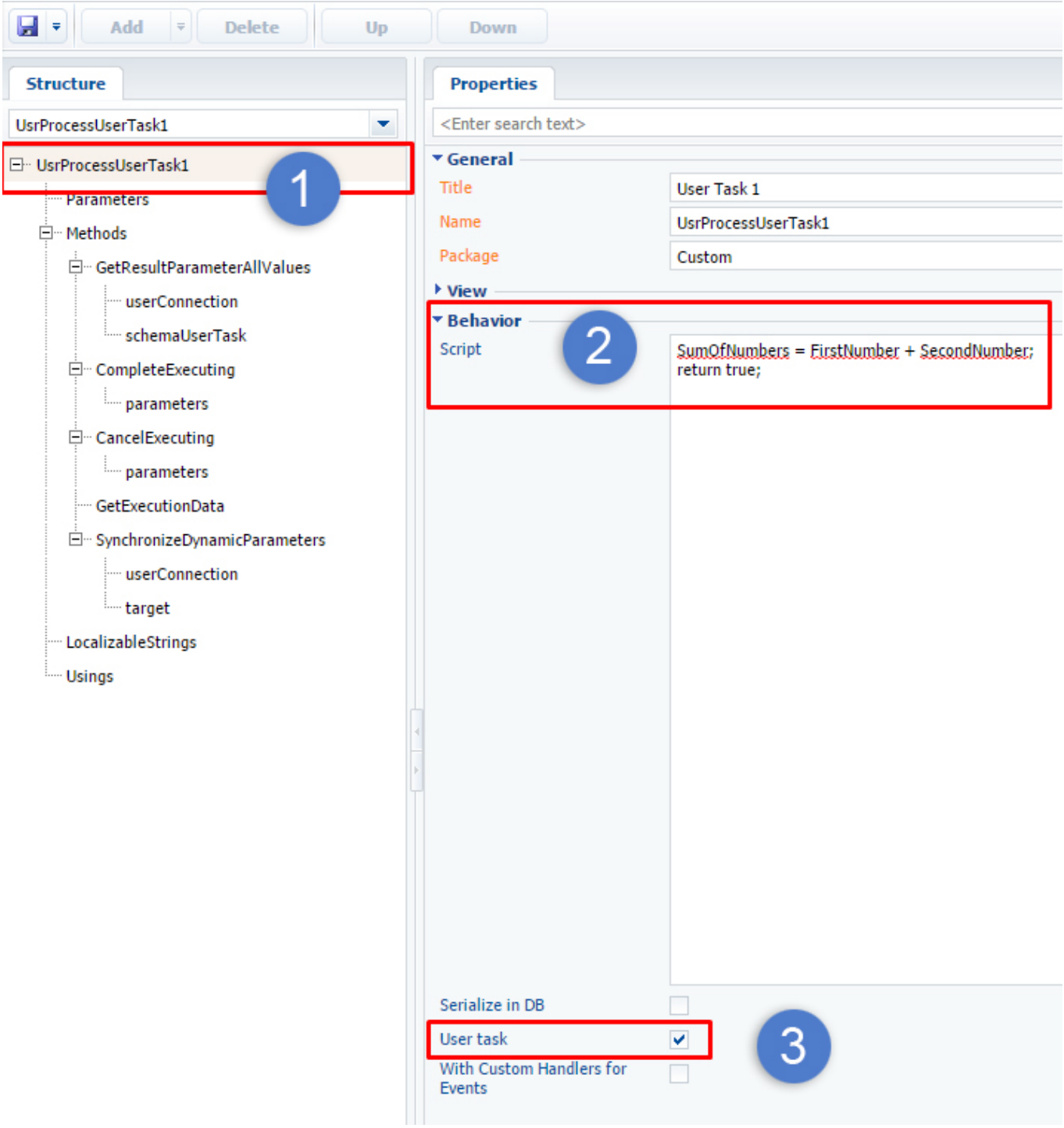
Table 1. Main parameters of the created process task

Name	Type	Description
FirstNumber	Integer	First number
SecondNumber	Integer	Second number
SumOfNumbers	Integer	Sum of numbers

3. Add 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) then 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;
```

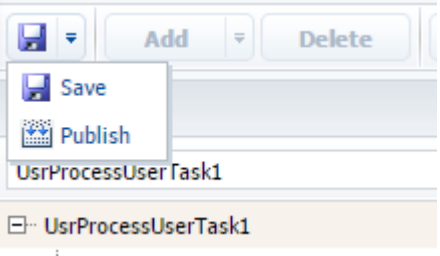
The parameters must be addressed directly by name. Return the *true* value at the end of the script to signal the successful execution of the element and continue the process.

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. Save and publish the schema

After assigning values to necessary properties of the created process task schema, save the schema and then publish it (Fig. 5).

Fig. 5. Saving and publishing a schema



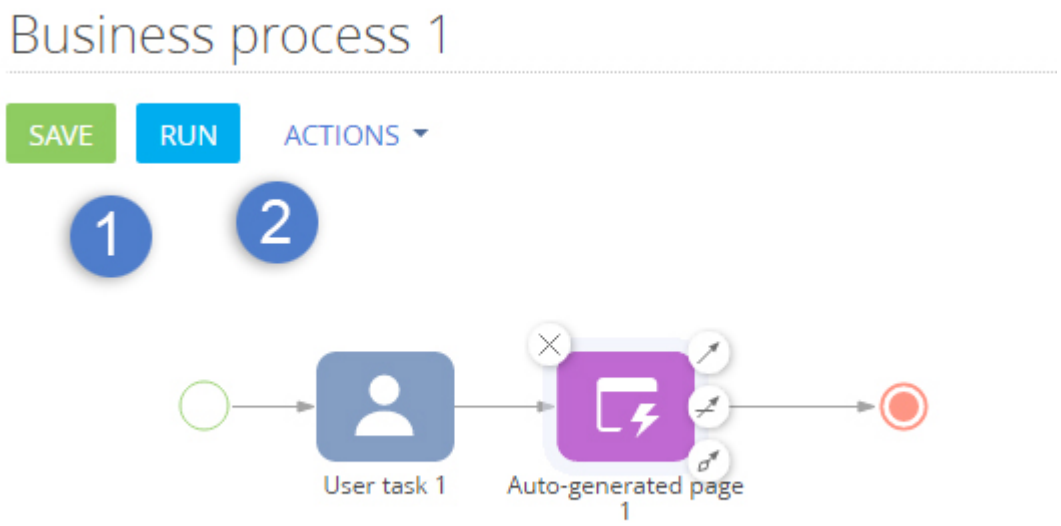
You can use the user task to create business processes after successfully publishing the schema.

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.

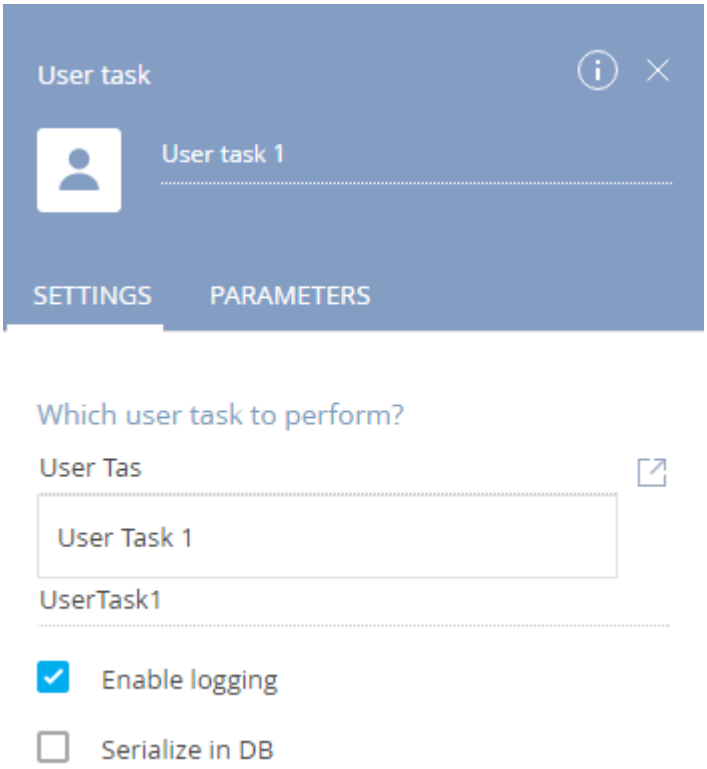
In the process designer, add the [User task] and the [Auto-generated page] elements to the process diagram (Fig. 6).

Fig. 6. Business process diagram.



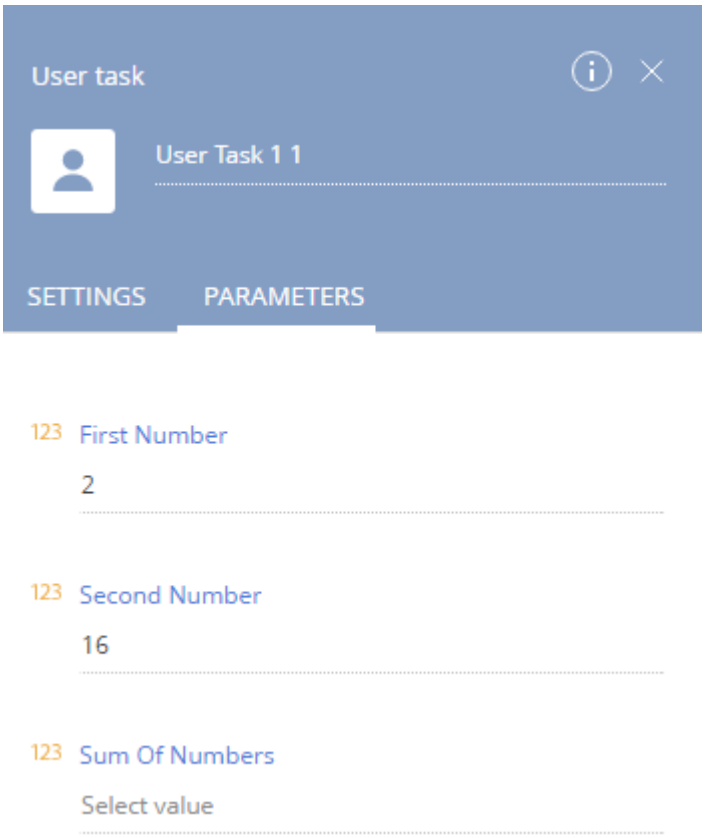
Specify the schema title (see step 1) in the [User task] property of the [User task 1] element (Fig. 7).

Fig. 7. Selecting a custom process task



Set the values of the sum parameters on the [Parameters] tab. The [Sum of numbers] parameter value will be overridden by the script (see step 3), so the value entry field for this parameter can be left blank.

Fig. 8. User task parameter values



The [Auto-generated page 1] element displays the task result, i.e. parameter sum values of the user task. To display

the sum of the parameters, you must first add an integer to the page element collection (Fig. 9), and then set the title of the auto-generated page element (Fig. 10, 1).To set the displayed value, call the value formula dialog by clicking the lightning icon in [Value] field. (Fig.10, 2).

Fig. 9. Adding page elements

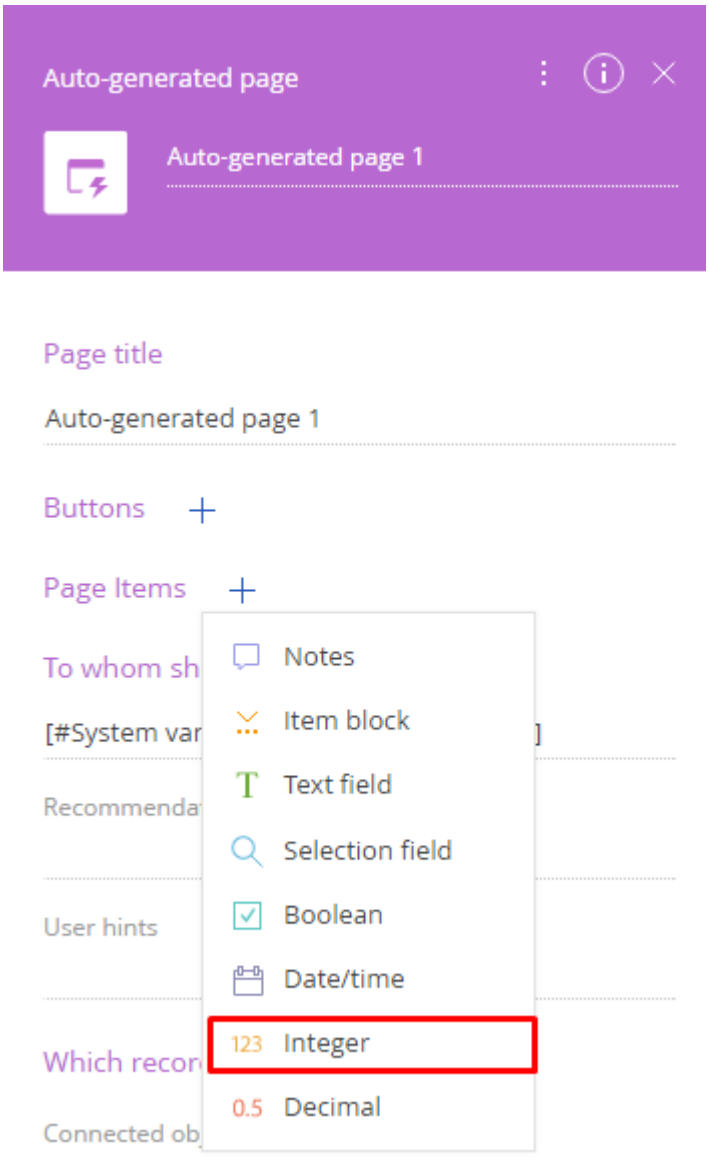
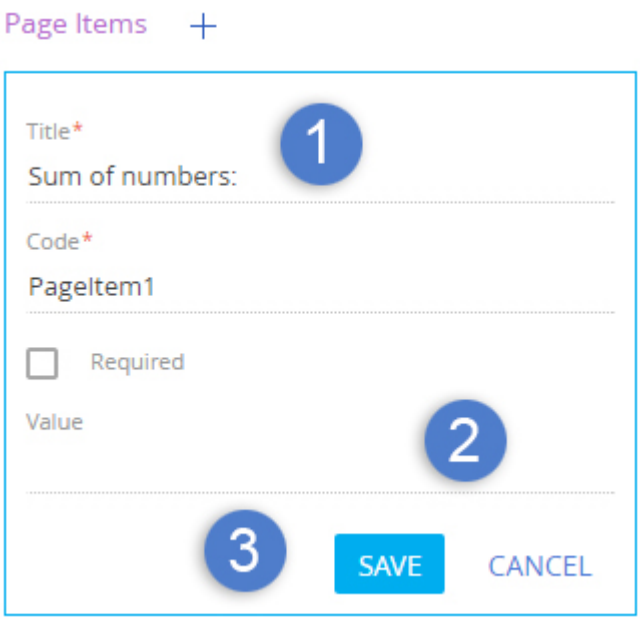
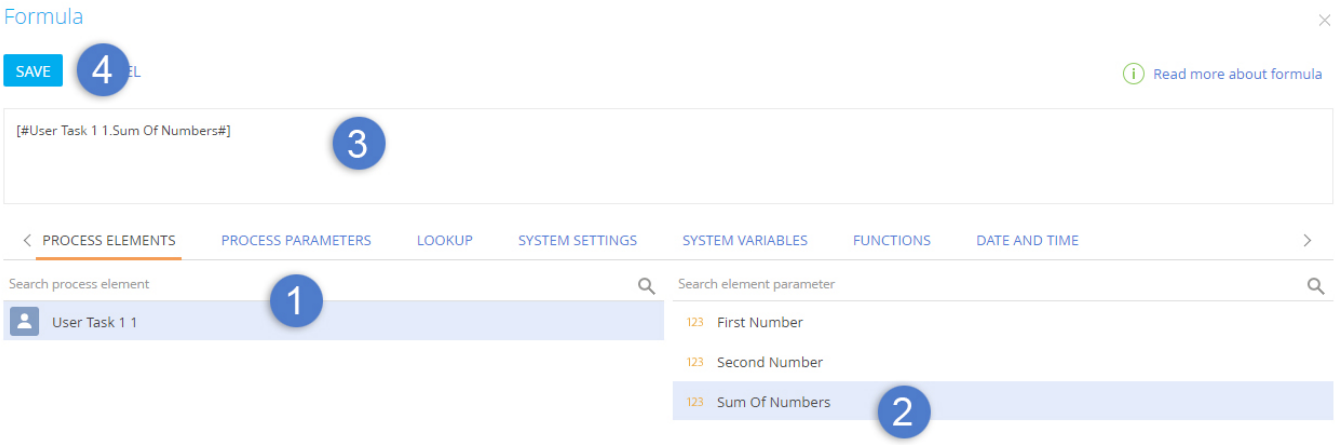


Fig. 10. Page element parameters



In the [Formula] dialog box that appears on the [Process Elements] tab, select the [Process task 1 1] element (Fig. 11, 1), and then double-click the [Sum of Numbers] item (Fig. 11, 2). The formula used to calculate the auto-generated page value will be displayed (Fig. 11, 3).

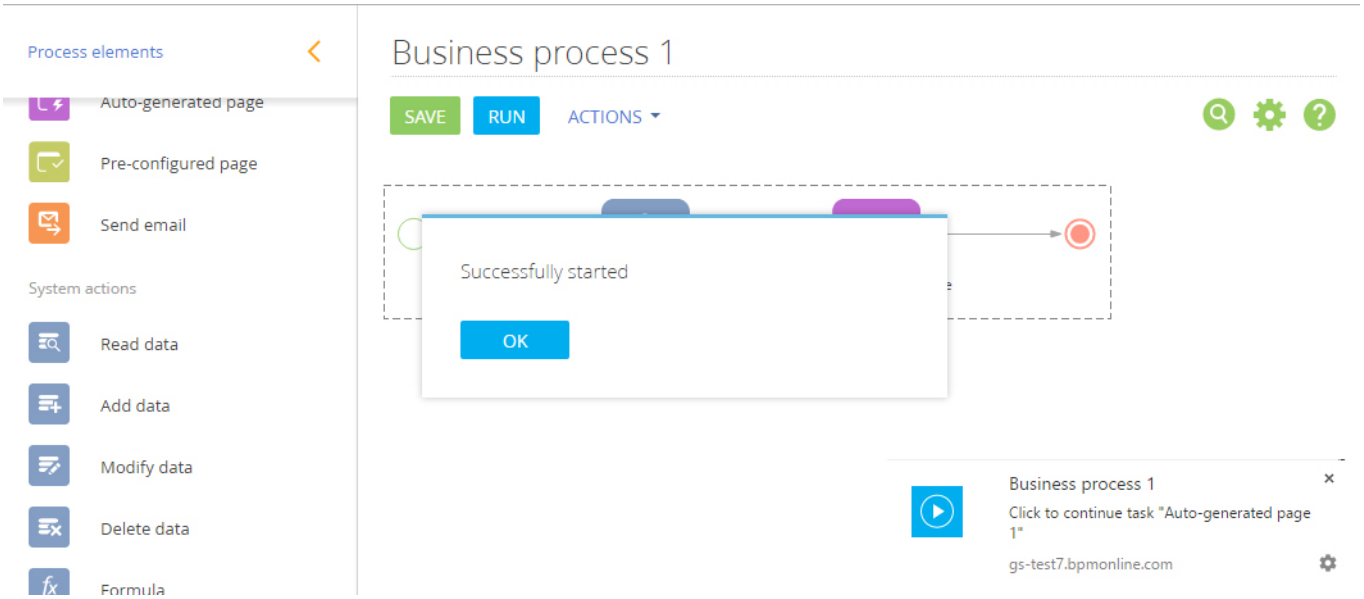
Fig. 11. The [Formula] dialog window



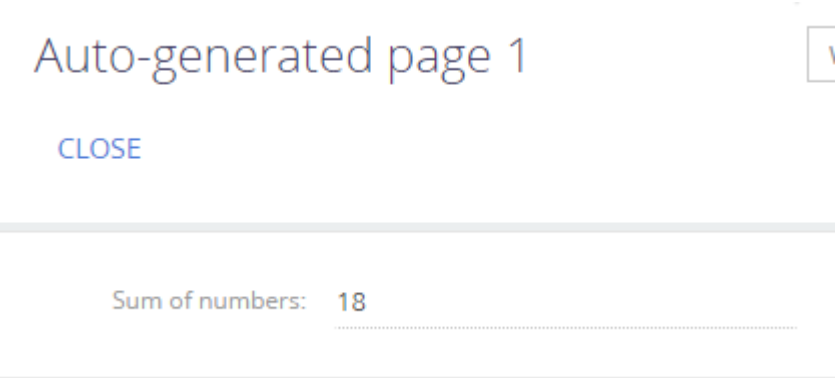
Save the formula (Fig. 11, 4) and the added auto-generated page element (Fig. 10, 3).

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.



> **Note**
>
> After changing the parameter values of the custom task, change the currently displayed page in the system before starting the business process again (go to any system section, for example). If you do not leave the auto-generated business process page, it will display the previous result.

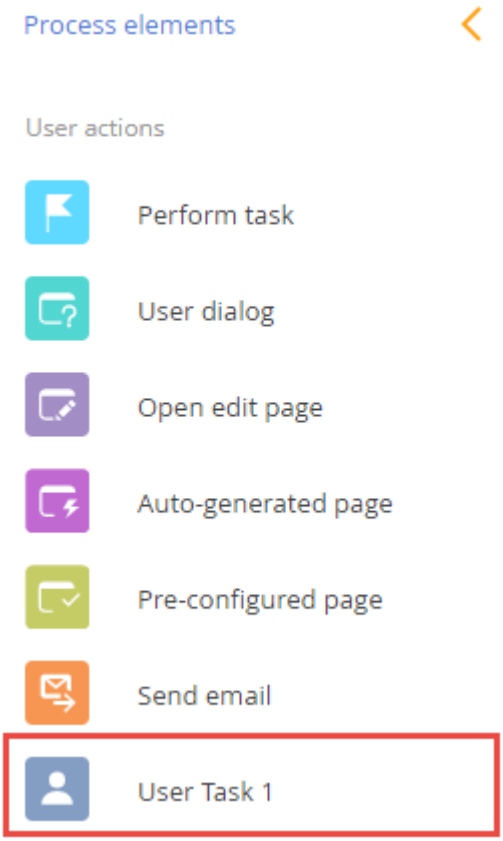
Adding process task to the [Process elements] tab

If created custom 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. 13. Custom element on the tab [User Task 1]



Typical customizations

- **How to create a mini-page**
- **Adding a mini-page to a module**
- **Adding pop-up hints**
- **How to modify sales pipeline calculations**
- **How to add extra filtering to a pipeline**
- **Adding a custom dashboard widget**
- **The Terrasoft.AlignableContainer custom element**
- **Adding a duplicate search rule**
- **Junk case custom filtering**

How to create a mini-page

Introduction

Bpm'online mini-pages were introduced in version 7.7. System users see mini-pages as an improved pop-up tool tip that contains additional section-specific functions. Mini-pages contain shortcuts to a number of actions, such as viewing the location of an account on the map, sending emails and making calls. Mini-pages are displayed when a user places their cursor over the hyperlinks that open edit pages in the [Accounts] and [Contacts] sections.

The primary purpose of mini-pages is:

- Give users the ability to get information about a record without opening the edit page.
- Give users the ability to quickly add records in sections by filling out only required fields as opposed to

filling out the entire edit page.

The structure of the mini-page view model schema is the same as the general structure of the system schema modules. The required parameters in the structure of the mini-page are:

- entitySchemaName, which contains the schema name of the object to which the mini-page will be bound,
- diff array that contains modifications.

These parameters are used to build the module view in the system UI.

Other items of the general schema structure, such as attributes, methods, mixins and messages can be used to implement the required functions. Use them to add custom controls, register messages, and implement business logic in various mini-pages. The appearance of the mini-page items can be customized.

How to add a mini-page in an existing section

To add a custom mini-page in an existing bpm'online section:

1. Add the view model schema for the card in the custom package. Set the following required properties for the created schema: [Title], [Name], [Package] and [Parent object]. Select "BaseMiniPage" as the parent object.
2. Use an SQL query to modify the SysModuleEdit system table in the bpm'online database.
3. Add necessary data and functions in the schema source code. Specify schema name of the object to which the mini-page will be bound in the entitySchemaName item and make at least one modification in the diff array.
4. Apply styles to the mini-page.

> Attention!
>
> To bind the created mini-page to the objects of a certain section, add the unique Id of the mini-page to the [Unique identifier of mini page schema] (MiniPageSchemaUId) column of these objects. Currently, the only way to do this is by modifying the SysModuleEdit system table in the bpm'online database using an SQL query.
>
> Please double-check any custom SQL queries made to the bpm'online database. Executing an invalid SQL query may damage existing data and disrupt system operation.

You can add a mini-page to any system object. Below is an example of how to create a mini-page for the [Knowledge Base] (KnowledgeBase) object.

Case description

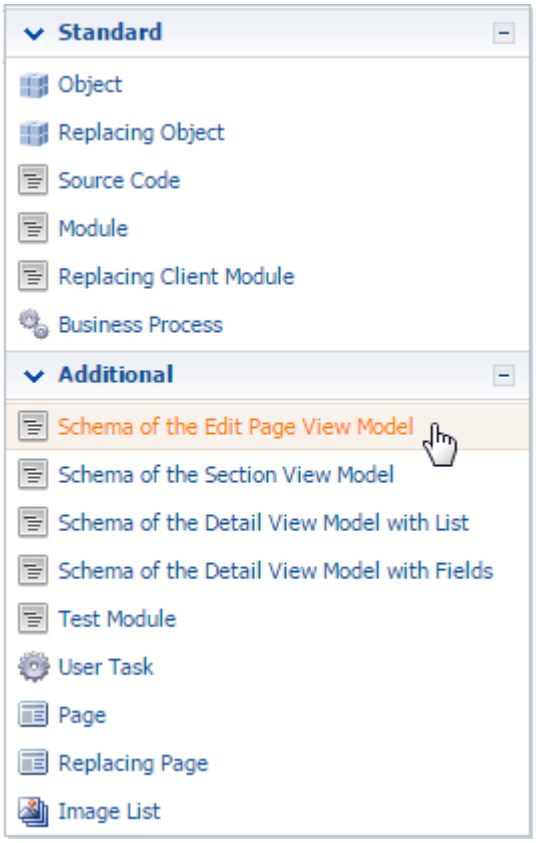
Create a custom mini-page for the [Knowledge base] section. The mini-page will be used to view a set of fields and download attachments of knowledge base articles.

Case implementation

1. Creating a model view schema of the mini-page

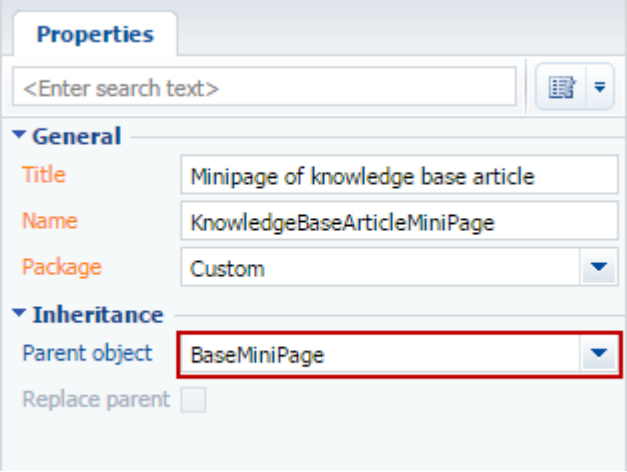
In the setup mode, go to the [Configuration] section and select the [Add] > [

Fig. 1. Adding card view schema



Fill out the properties of the mini-page view schema as shown on fig. 2.

Fig. 2. Fill out the mini-page properties.



2. Registering a mini-page in the database.

Execute the following SQL query to make changes in the database:

```
DECLARE
    -- Name of the view schema of the created mini-page.
    @ClientUnitSchemaName NVARCHAR(100) = 'KnowledgeBaseArticleMiniPage',
    -- Name of the object schema to which the mini-page is bound.
    @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, a unique Id for the mini-page is added to the MiniPageSchemaUId field of the SysModuleEdit table (fig. 3).

Fig. 3. Unique Id value of mini-page in the SysModuleEdit database 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 fields of the primary object

The mini-page code structure is identical to that of the common edit page. Before adding fields, specify the object schema, as you would for any common edit page. In this case the object schema name is "KnowledgeBase".

Adding fields to the mini-page is similar to adding fields to an edit page: all necessary changes are added to the diff array.
The only difference that results from the limited space of a mini-page, is the need to hide field titles and specify the display mode.

```
"values": {
            "labelConfig": {
                   // Hiding field title
                   "visible": false
            },
            // Specifying display mode in mini-page.
            "isMiniPageModelItem": true
}
```

A base mini-page contains the following items:

- MiniPage — Terrasoft.GridLayout — card field.
- HeaderContainer — Terrasoft.Container — card title (is placed in the first row of card field).

Add the [Name] field

On this stage, the mini-page is already functional and the remaining steps are optional.

Source code of the mini-page view model schema:

```
define("KnowledgeBaseArticleMiniPage", [], function() {
    return {
        entitySchemaName: "KnowledgeBase",
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        messages: {},
        mixins: {},
        attributes: {},
        methods: {},
        diff: /**SCHEMA_DIFF*/[
            {
```

```
                "operation": "insert",
                "name": "Name",
                "parentName": "HeaderContainer",
                "propertyName": "items",
                "index": 0,
                "values": {
                    "labelConfig": {
                        "visible": false
                    },
                    "isMiniPageModelItem": true
                }
            },
            {
                "operation": "insert",
                "name": "Notes",
                "parentName": "MiniPage",
                "propertyName": "items",
                "values": {
                    "labelConfig": {
                        "visible": false
                    },
                    "isMiniPageModelItem": true,
                    "layout": {
                        "column": 0,
                        "row": 1,
                        "colSpan": 24
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

4. How to add a functional button to a mini-page

One of the case requirements is the ability to download knowledge base article attachments using mini-pages.

Working with additional data is enabled via the mechanism that displays it as the drop-down list of a preconfigured button. To add a button for selecting attachments of a knowledge base article:

1. Add the button description to the diff array – the [FilesButton] item.
2. Add the attribute that connects primary and additional records – the [Article] virtual button.
3. Add methods for working with the drop-down list of the file selection button:
    - override the init() method;
    - override the EntityInitialized() method;
    - set the [Article] attribute value using the setArticleInfo() method;
    - get data about the attachments of the current knowledge base article using the getFiles(callback, scope) method;
    - populate the drop-down list collection of the file select button using the initFilesMenu(files) method;
    - initiate file loading and adding to the drop-down list of the file select button using the fillFilesExtendedMenuData() method;
    - initiate file download

5. How to apply style to a mini-page

Create the KnowledgeBaseArticleMiniPageCss module on the LESS tab and specify a style.

```
div[data-item-marker='KnowledgeBaseArticleMiniPageContainer'] > div {
    width: 34em;
}
```

In the designer of the mini-page schema, specify the dependency of the style module and add loading of this module in the source code.

Example of complete code of a mini-page:

```
define("KnowledgeBaseArticleMiniPage",
["terrasoft", "KnowledgeBaseFile", "ConfigurationConstants",
"css!KnowledgeBaseArticleMiniPageCss"],
    function(Terrasoft, KnowledgeBaseFile, ConfigurationConstants) {
        return {
            entitySchemaName: "KnowledgeBase",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            messages: {},
            mixins: {},
            attributes: {
                Article: {
                    type: Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                    referenceSchemaName: "KnowledgeBase"
                }
            },
            methods: {
                /**
                * Initializes drop-down list collection of the file select button.
                */
                init: function() {
                    this.callParent(arguments);
                    this.initExtendedMenuButtonCollections("File", ["Article"],
this.close);
                },
                /**
                * Initializes value of the attribute that binds primary and
additional records.
                * Fills out drop-down list collection of the file select button.
                */
                onEntityInitialized: function() {
                    this.callParent(arguments);
                    this.setArticleInfo();
                    this.fillFilesExtendedMenuData();
                },
                /**
                * Initializes loading of the files and adds them to the drop-down
list of the file select button.
                */
                fillFilesExtendedMenuData: function() {
                    this.getFiles(this.initFilesMenu, this);
                },
                /**
                * Sets value of the attribute that binds primary and additional
records.
                */
                setArticleInfo: function() {
                    this.set("Article", {
                        value: this.get(this.primaryColumnName),
                        displayValue: this.get(this.primaryDisplayColumnName)
                    });
                },
                /**
                * Gets information about files of the current knowledge base article.
                */
                getFiles: function(callback, scope) {
                    var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
                        rootSchema: KnowledgeBaseFile
```

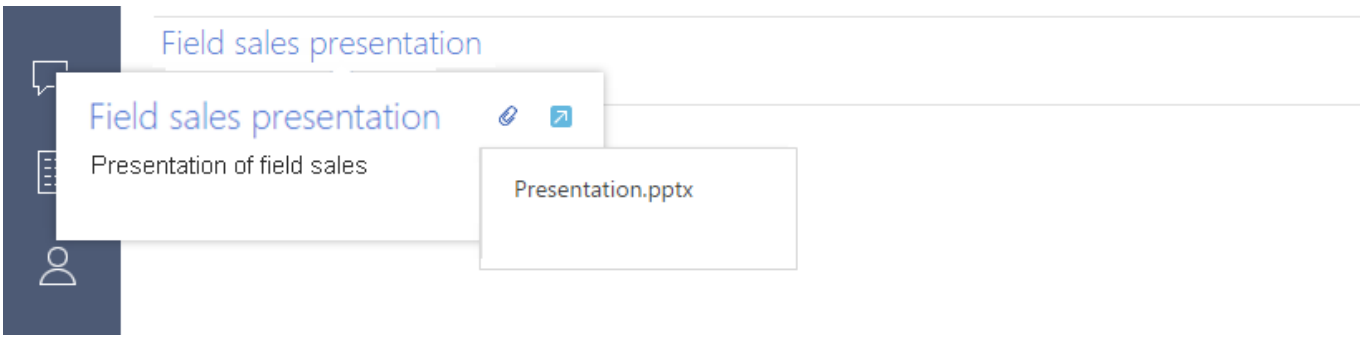
```
                    });
                    esq.addColumn("Name");
                    var articleFilter =
this.Terrasoft.createColumnFilterWithParameter(
                        this.Terrasoft.ComparisonType.EQUAL, "KnowledgeBase",
this.get(this.primaryColumnName));
                    var typeFilter = 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);
                },
                /**
                * Fills out the drop-down list collection of the file select 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 downloading of the selected file.
                */
                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": false
                        },
                        "isMiniPageModelItem": true
                    }
```

```
                },
                {
                    "operation": "insert",
                    "name": "Notes",
                    "parentName": "MiniPage",
                    "propertyName": "items",
                    "values": {
                        "labelConfig": {
                            "visible": false
                        },
                        "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": {
                            // The image must first be added to the mini-page
resources.
                            "bindTo": "Resources.Images.FilesImage"
                        },
                        // Drop-down list setup.
                        "extendedMenu": {
                            // Drop-down list item name.
                            "Name": "File",
                            // Name of the mini-page attribute that binds the primary
and additional records
                            "PropertyName": "Article",
                            // Button click handler setup.
                            "Click": {
                                "bindTo": "fillFilesExtendedMenuData"
                            }
                        }
                    },
                    "index": 1
                }
            ]/**SCHEMA_DIFF*/
        };
    });
```

6. How to save a mini-page schema

After saving the schema and updating the [Knowledge base] section page, the mini-page with the list of attachments will be displayed If a user hovers their cursor over the knowledge base article title (fig. 4).

Fig. 4. Functional mini-page



Adding a mini-page to a module

General information

When adding object mini-pages to the system you need to connect them to system modules. Arbitrary modules enable you to create links for a certain object. The connection of a mini-page displayed when you hover the cursor over a link enables you to obtain information about the object without going to the section of this object.

In the basic application version an object mini-page is connected to the following modules:

- telephony in the communication panel
- email in the communication panel
- the notification center in the communication panel
- the [Feed] section in the communication panel
- the chart-list in the dashboards section

Case description

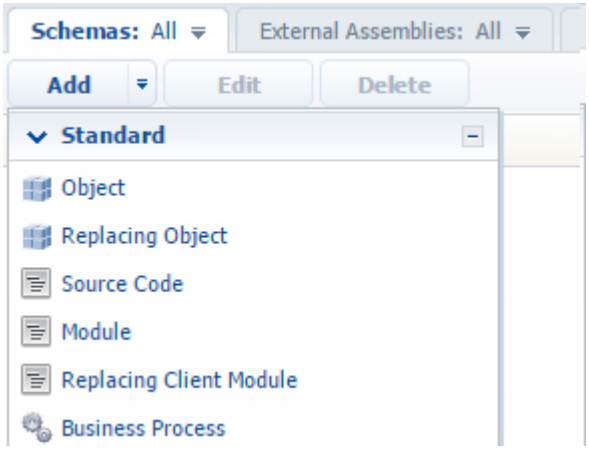
Display the current user in the upper right corner of the application next to the user profile icon. Open the mini-page by hovering the cursor over the current user profile.

Case implementation algorithm

1. Create a module

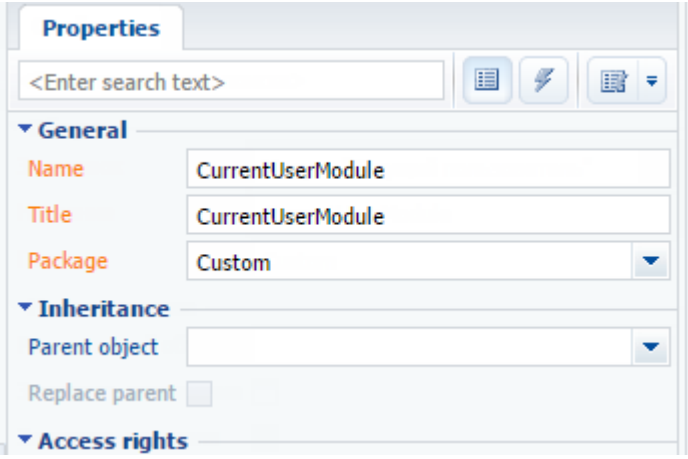
Open the system designer. Go to the [Configuration] section, and on the [Schema] tab, execute the [Add] > [Standard] > [Module] menu command (Fig. 1).

Fig. 1. Adding a module



For the generated module, you need to set the "CurrentUserModule" for the [Name] property, and the "Module "Current user" for the [Title] property (Fig. 2).

Fig. 2. Module properties



2. Create a view and module view model

To create a view model in the *CurrentUserModule* module you must implement a class inherited from *Terrasoft.BaseViewModel*. You must connect the *Terrasoft.MiniPageUtilities* utility to the *mixins* property that will enable you to use the mini-page call methods

```
// Defining the CurrentUserViewModel class.
Ext.define("Terrasoft.configuration.CurrentUserViewModel", {
    // Parent class name.
    extend: "Terrasoft.BaseViewModel",
    // Short class name.
    alternateClassName: "Terrasoft.CurrentUserViewModel",
    // The mixins used.
    mixins: {
        MiniPageUtilitiesMixin: "Terrasoft.MiniPageUtilities"
    }
});
```

To create a view, you must implement a class inherited from *Terrasoft.BaseModule*.

```
// Defining the CurrentUserModel class.
Ext.define("Terrasoft.configuration.CurrentUserModule", {
    // Short class name.
    alternateClassName: "Terrasoft.CurrentUserModule",
    // 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,

    ...
});
```

In this class you need to extend the *init* and render methods of the *Terrasoft.BaseModule* base class. The *Init* method initializes the view module model, and the *render* method connects the view model with the view in the container that is passed in the *renderTo* parameter. To create a view model use the *getViewModel* method. The link to the created view model is stored in the *viewModel* property.

```
// Creates module view model.
getViewModel: function() {
    return Ext.create("Terrasoft.CurrentUserViewModel");
```

```
},
// Module initialization.
init: function() {
    this.viewModel = this.getViewModel();
},
// Displays module view.
render: function(renderTo) {
    // Receiving view object.
    var view = this.getView();
    // Connecting a view with a view model.
    view.bind(this.viewModel);
    // Displaying a view in the renderTo element.
    view.render(renderTo);
}
```

You must then determine the getView method for it to be displayed further. The view must display the full name of the current user with the hyperlink to the contact edit page. When you build a hyperlink, you must define an event handler for the mouse hover.

```
// Creates module views.
getView: function() {
    // Getting current user contact..
    var currentUser = Terrasoft.SysValue.CURRENT_USER_CONTACT;
    // View — the Terrasoft.Hyperlink class instance.
    return Ext.create("Terrasoft.Hyperlink", {
        // filling in the link caption with a contact name.
        "caption": currentUser.displayValue,
        // Mouse over link event handler.
        "linkMouseOver": {"bindTo": "linkMouseOver"},
        // Property containing additional object parameters.
        "tag": {
            // Current user ID.
            "recordId": currentUser.value,
            // Object schema name.
            "referenceSchemaName": "Contact"
        }
    });
}
```

The following is the full module source code.

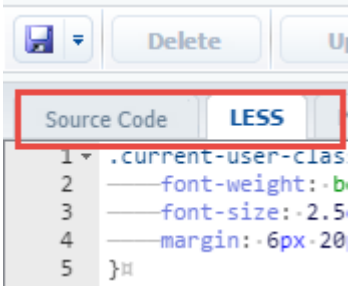
```
// Module definition.
define("CurrentUserModule", ["MiniPageUtilities"], function() {
    // Defining the CurrentUserViewModel class.
    Ext.define("Terrasoft.configuration.CurrentUserViewModel", {
        // Parent class name.
        extend: "Terrasoft.BaseViewModel",
        // Short class name.
        alternateClassName: "Terrasoft.CurrentUserViewModel",
        // The mixins used.
        mixins: {
            MiniPageUtilitiesMixin: "Terrasoft.MiniPageUtilities"
        }
    });
    // Defining the CurrentUserModel class.
    Ext.define("Terrasoft.configuration.CurrentUserModule", {
        // Short class name.
        alternateClassName: "Terrasoft.CurrentUserModule",
        // 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() {
            // Getting current user contact..
            var currentUser = Terrasoft.SysValue.CURRENT_USER_CONTACT;
            // View — the Terrasoft.Hyperlink class instance.
            return Ext.create("Terrasoft.Hyperlink", {
                // filling in the link caption with a contact name.
                "caption": currentUser.displayValue,
                // Mouse over link event handler.
                "linkMouseOver": {"bindTo": "linkMouseOver"},
                // Property containing additional object parameters.
                "tag": {
                    // Current user ID.
                    "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 module view.
        render: function(renderTo) {
            // Receiving view object.
            var view = this.getView();
            // Connecting a view with a view model.
            view.bind(this.viewModel);
            // Displaying a view in the renderTo element.
            view.render(renderTo);
        }
    });
    return Terrasoft.CurrentUserModule;
});
```

For a more expressive display of hyperlinks, you can add styles to the created module. To do this, go to the LESS tab in the module designer (Fig. 3) and add the following source code.

```
.current-user-class a {
    font-weight: bold;
    font-size: 2.5em;
    margin: 6px 20px;
}

.current-user-class a:hover {
    text-decoration: none;
}
```

Fig. 3. The LESS module designer tab



Save the created module.

3. Create a view display container

For links to be displayed in the user profile, you need to place the container in the upper right corner of the application and load the view of the created module. To do this, create a replacing client module that will extend the functionality of the *MainHeaderSchema* schema implemented in the *NUI* package. The creation of a replacing client module is described in the "**Creating a custom client module schema**" article.

To display a view in the source code of the replaced schema, you need to create a container. For this you need to use the *diff* property. For the container to be displayed in the upper right corner of the page, set the *RightHeaderContainer* as the parent of the created container. Next, you need to override the onRender method that is invoked after the view, to which you need to load the created module that has been displayed.

The following is the full replaced module source code.

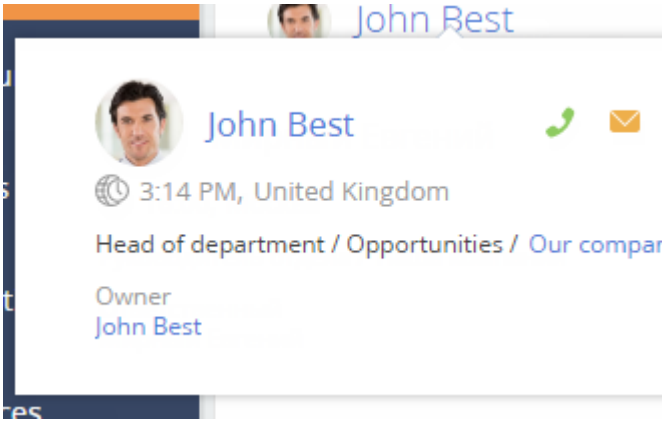
```
// Module definition.
define("MainHeaderSchema", [], function() {
    return {
        methods: {
            // Executes actions after the view has been dispalyed.
            onRender: function() {
                // Parent method calling
                this.callParent(arguments);
                // Current user module loading.
                this.loadCurrentUserModule();
            },
            // Loads current user module.
            loadCurrentUserModule: function() {
                // Get container to which the module is loaded..
                var currentUserContainer = this.Ext.getCmp("current-user-container");
                // Container existence check.
                if (currentUserContainer && currentUserContainer.rendered) {
                    // Loading module to container.
                    this.sandbox.loadModule("CurrentUserModule", {
                        // Container name.
                        renderTo: "current-user-container"
                    });
                }
            }
        },
        diff: [
            {
                // Element insertion operation.
                "operation": "insert",
                // Element name.
                "name": "CurrentUserContainer",
                // Parent container name.
                "parentName": "RightHeaderContainer",
                // Property name.
                "propertyName": "items",
                // Element value.
                "values": {
```

```
                        // Container ID.
                        "id": "current-user-container",
                        // Element type.
                        "itemType": Terrasoft.ViewItemType.CONTAINER,
                        // Container classes.
                        "wrapClass": ["current-user-class"],
                        // Container elements.
                        "items": []
                    }
                }
            ]
        };
});
```

Save the created module.

To apply the changes, you must update the application page and clean the cache. The full name of the current user with the hyperlink to the contact edit page will be displayed in the upper right corner of the application. When you hover the cursor over a hyperlink, the mini-page will appear with data about the current user (Fig. 4).

Fig. 4. Contact mini-page



Adding pop-up hints

As of version 7.7.0 bpm'online supports tooltips. This article describes tooltip types that can be added to the system elements and methods for adding them.

All tooltips can be divided into 3 main groups:

1 A field tooltip (the field title is marked with a small green triangle)

. The tooltip appears when you hover the mouse cursor over the triangle or click the field title.

2 A tooltip for other control elements (buttons, completeness indicators, images). The tooltip appears when you hover the mouse cursor over a control element.

3 The information button . The tooltip appears when you hover over the information button.

General algorithm for adding tooltips for standard controls:

1. Create the replacement schema of a page or section
2. Add tooltip text to the collection of schema localizable strings.
3. Describe the necessary modifications to schema elements in the diff array.

Let's consider creating different types of tooltips with specific examples.

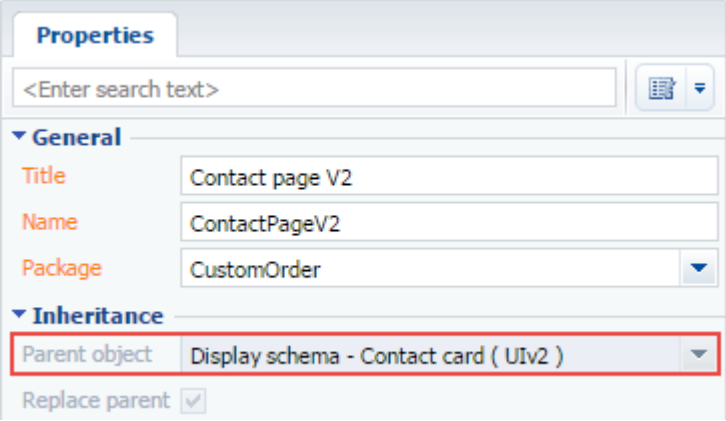
Example 1. Adding a tooltip to the [Account] field of the contact edit page

1 In a custom package, create a replacement contact edit page

You need to create a replacement client module, and set the [Contact mini-page display schema] as a parent object (Fig. 1).

The process of creating a replacement page is described in the "**Creating a custom client module schema**" article.

Fig. 1 Product replacement edit page properties

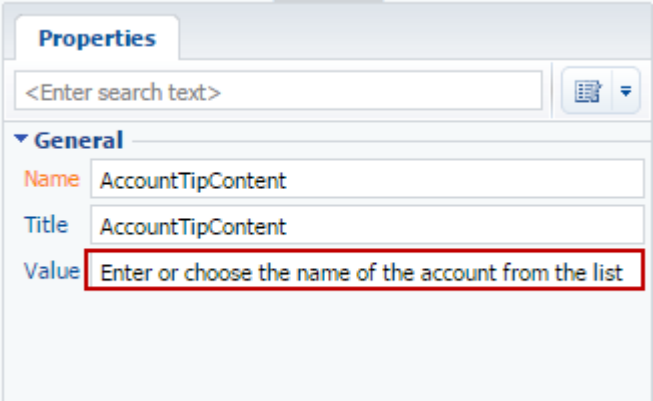


2 Add tooltip text to the collection of localizable strings of the replacement edit page schema

To do this, right-click the [LocalizableStrings] node structure and select [Add].

Fill in the properties of the created string, as shown in Fig. 2.

Fig. 2 Custom localizable string properties



3 Add field configuration object to the diff array

To add a tooltip to the field, you need to add the "tip" property to the "values" property field, which, in turn, should contain a "content" property. The value of the "content" property will serve as a tooltip text.

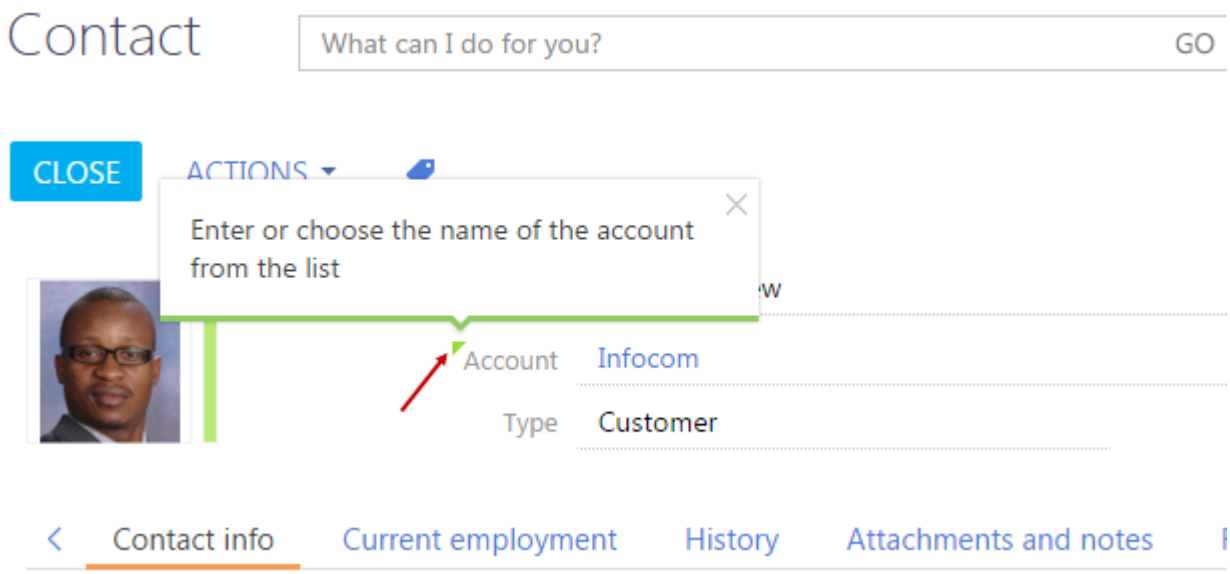
Here is the source code of the contact replacement edit page schema.

```
define("ContactPageV2", [],
function () {
    return {
        //Edit page object schema name
        entitySchemaName: "Contact",
        //Tooltip display setup
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding a tooltip to a field.
```

```
            {
                // Indicates that the operation changing the existing item is being
performed.
                "operation": "merge",
                "name": "Account",
                "parentName": "Header",
                "propertyName": "items",
                "values": {
                    //
                    "tip": {
                        // Tooltip text.
                        "content": { "bindTo": "Resources.Strings.AccountTipContent"
},

                        // Tooltip dispaly mode.
                        // By default, WIDE - width of the green stripe,
                        // which is displayed in the tooltip.
                        "displayMode": Terrasoft.controls.TipEnums.displayMode.WIDE
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

After the schema is saved on the contact edit page, the [Account] field will be displayed with a tooltip (Fig. 3).

Fig. 3 Result demonstration



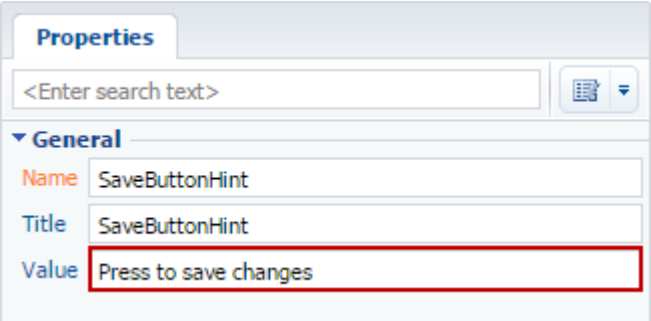
Example 2. Adding a tooltip to the [Save] button of the contact edit page

1 In a custom package, create a replacement contact edit page

2 Add tooltip text to the collection of localizable strings of the replacement edit page schema

Fill in the properties of the created string, as shown in Fig. 4.

Fig. 4 Custom localizable string properties



3 Add button configuration object to the diff array

There are several ways to add a tooltip to a control element.

Method 1

Add the "hint" property to the "values" control element property, which, in turn, should contain the tooltip text.

Here is the source code of the contact replacement edit page schema when adding tips using the first method.

```
define("ContactPageV2", [],
function () {
    return {
        //Edit page object schema name
        entitySchemaName: "Contact",
        //Tooltip display setup
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding a button to a field.
            {
                // Indicates that the operation changing the existing button is being
performed.
                "operation": "merge",
                "parentName": "LeftContainer",
                "propertyName": "items",
                "name": "SaveButton",
                "values": {
                    // Button tooltip
                    "hint": { "bindTo": "Resources.Strings.SaveButtonHint" }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

Method 2

Add the "tips" array to the "values" control element property. Then, using the "insert" operation, add the tooltip configuration object to the "tips" array. In the "values" property of this object, specify the "content" property — the text of the tooltip. Using this method, you can customize a tooltip i.e., change the display style, bind the tooltip to any event of the view model, add control elements, etc.

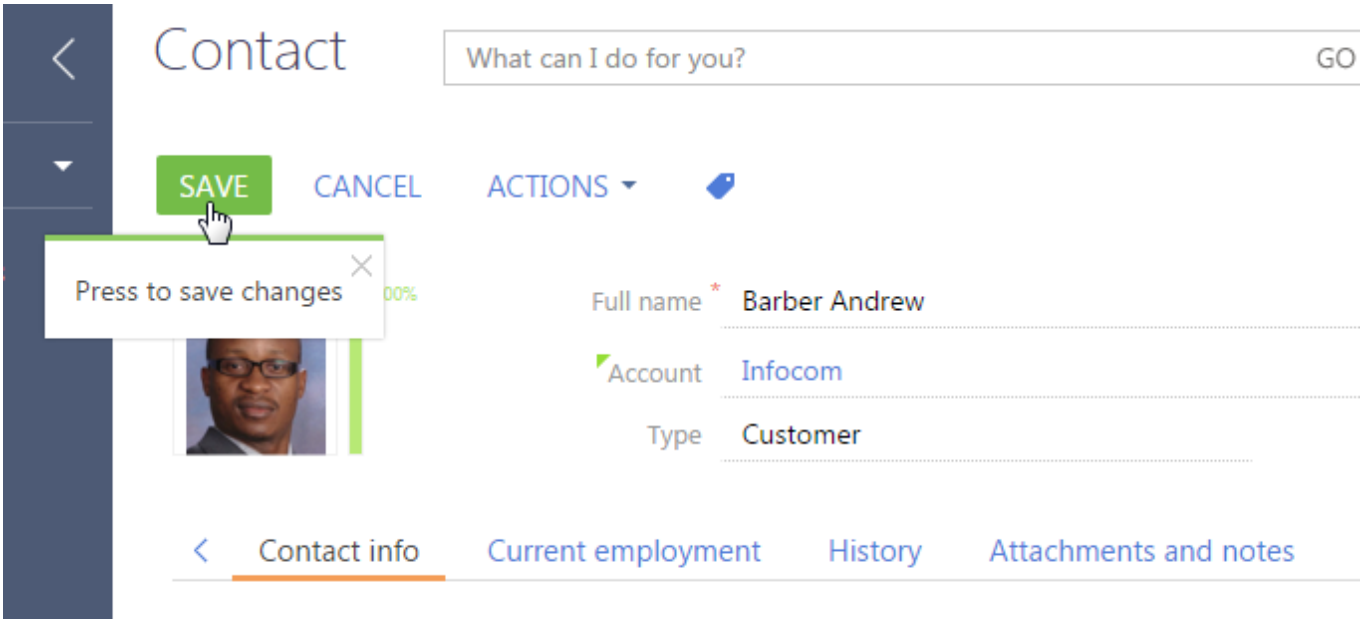
> Important!
>
> This method works for itemType:
>
> - Terrasoft.ViewItemType.BUTTON,
> - Terrasoft.ViewItemType.LABEL, Terrasoft.ViewItemType.COLOR_BUTTON,
> - Terrasoft.ViewItemType.HYPERLINK,
> - Terrasoft.ViewItemType.INFORMATION_BUTTON,
> - and for those items in which the specified property is "generator".

Here is the source code of the contact replacement edit page schema when adding tips using the second method.

```
define("ContactPageV2", [],
function () {
    return {
        //Edit page object schema name
        entitySchemaName: "Contact",
        //Tooltip display setup
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding a button to a field.
            {
                // Indicates that the operation changing the existing button is being
performed.
                "operation": "merge",
                "parentName": "LeftContainer",
                "propertyName": "items",
                "name": "SaveButton",
                "values": {
                    // Button tooltip array.
                    "tips": []
                }
            },
            // Simple tooltip configuration element
            {
                // Indicates that the operation adding a new element is being
performed.
                "operation": "insert",
                "parentName": "SaveButton",
                "propertyName": "tips",
                "name": "CustomShowedTip",
                "values": {
                    // Tooltip text.
                    "content": {"bindTo": "Resources.Strings.SaveButtonHint"}
                    // Here you can optionally configure other options
                    // of tooltip display and operation.
                }
            },
        ]/**SCHEMA_DIFF*/
    };
});
```

After the schema is saved on the contact edit page, the [Save] button will be displayed with a tooltip (Fig. 5).

Fig. 5 Result demonstration



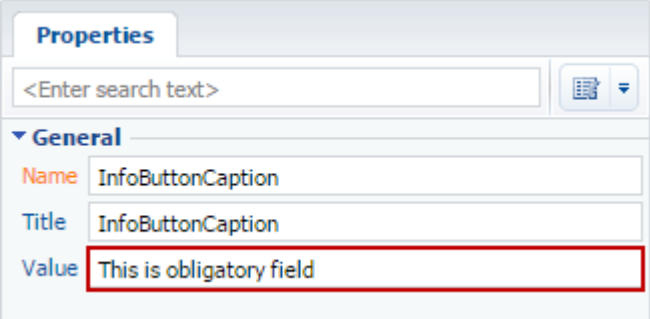
Example 3. Adding a tooltip to the [Full name] field of the contact edit page

1 In a custom package, create a replacement contact edit page

2 Add tooltip text to the collection of localizable strings of the replacement edit page schema

Fill in the properties of the created string, as shown in Fig. 6.

Fig. 6 Custom localizable string properties



3 Add button configuration object to the diff array

To add the information button to the page, you need to add a new item to the diff array with the **Terrasoft.ViewItemType.INFORMATION_BUTTON** type and the "**content**" property. The value of the "content" property will serve as a tooltip text.

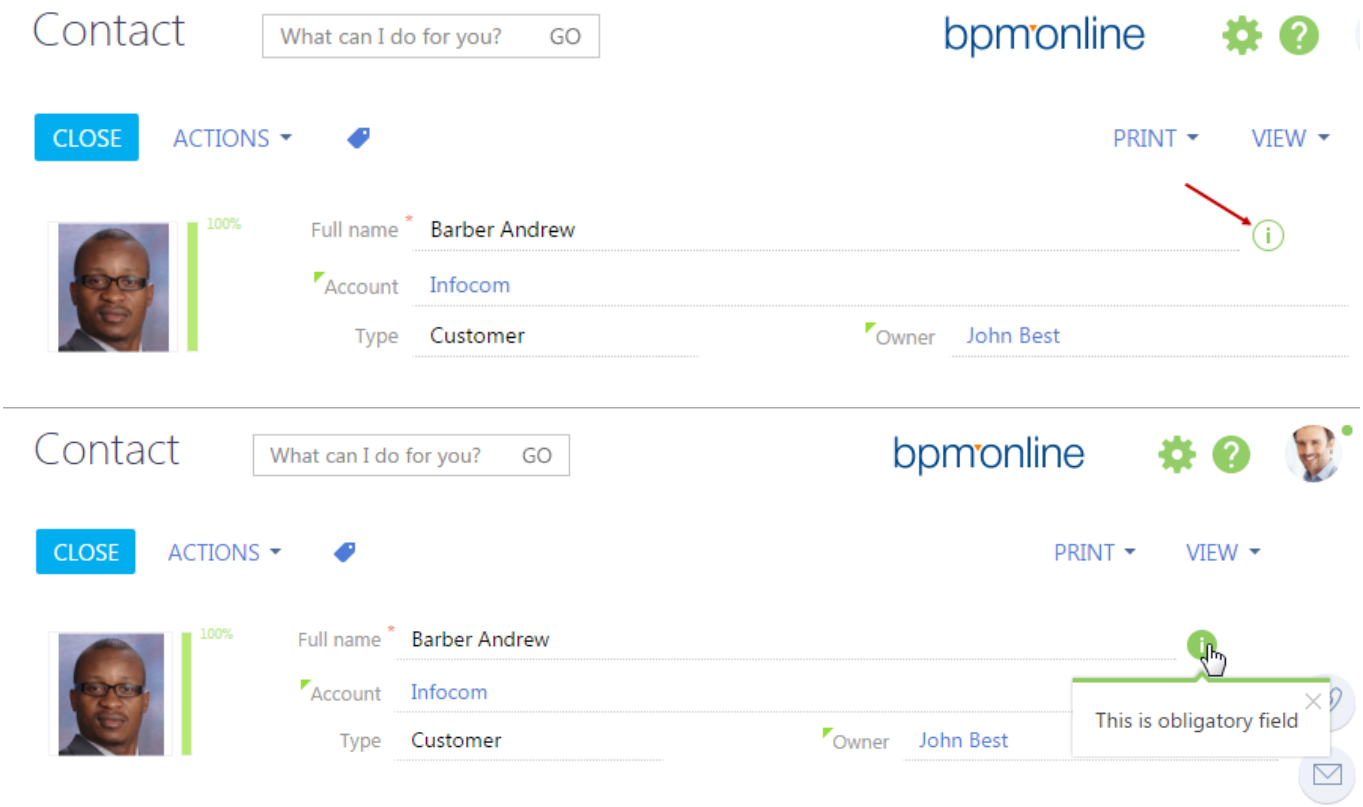
Here is the source code of the contact replacement edit page schema.

```
define("ContactPageV2", [],
function () {
    return {
        //Edit page object schema name
        entitySchemaName: "Contact",
        //Tooltip display setup
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding a button to a field.
            {
                // Indicates that the operation changing the existing button is being
performed.
                "operation": "merge",
```

```
                "parentName": "Header",
                "propertyName": "items",
                "name": "Name",
                "values": {
                    "layout": { "column": 4, "row": 0, "colSpan": 18 }
                }
            },
            {
                // Indicates that the operation adding a new element is being
performed.
                "operation": "insert",
                "parentName": "Header",
                "propertyName": "items",
                "name": "SimpleInfoButton",
                "values": {
                    "layout": { "column": 22, "row": 0, "colSpan": 1 },
                    // Indicates the control element type - "Information button"
                    "itemType": Terrasoft.ViewItemType.INFORMATION_BUTTON,
                    // Information button tooltip text
                    "content": { "bindTo": "Resources.Strings.InfoButtonCaption" }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

After the schema is saved on the contact edit page, the [Account] field will be displayed with a tooltip (Fig. 7).

Fig. 7 Result demonstration



Adding a web-resource link to a tooltip

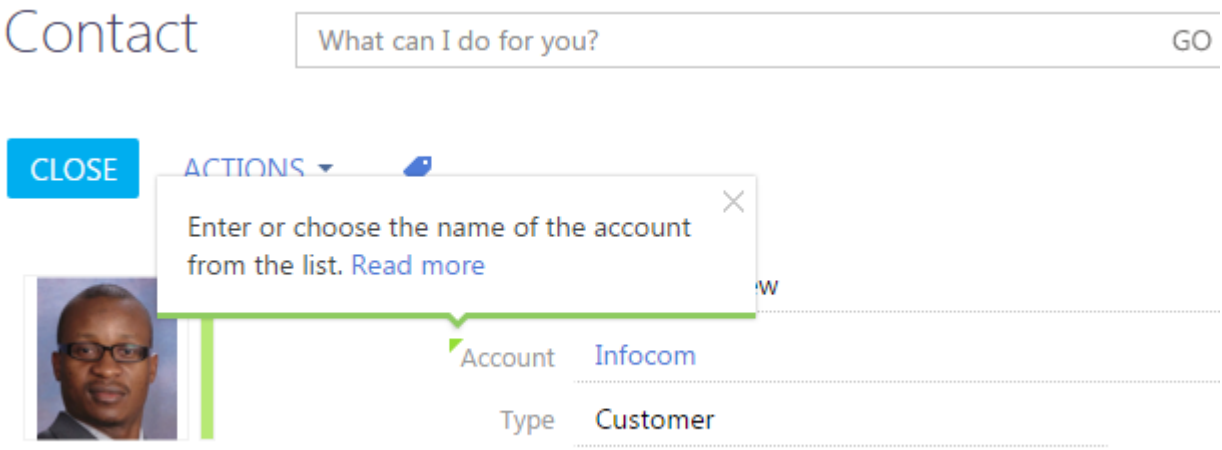
You can add web-resource links and context-sensitive help to tooltips. For this you need to add html-code link directly to a localizable string of the tooltip text.

Example of adding a direct link to a web-resource:

Read more

As a result the tooltip will look like this:

Fig. 8 Tooltip with a link example



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.

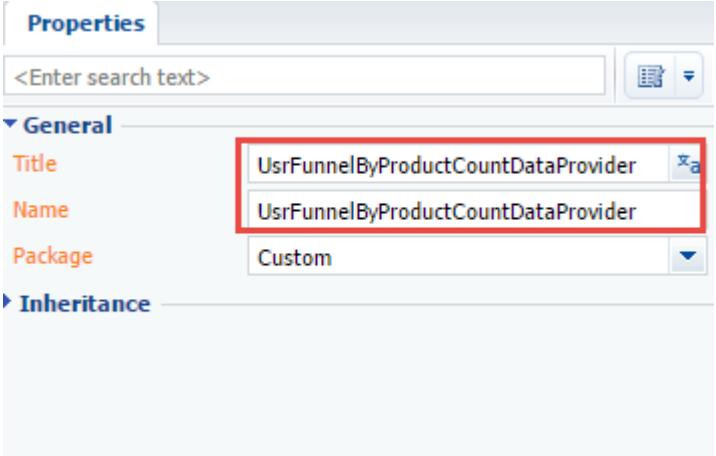
Case implementation algorithm

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* (рис. 1).

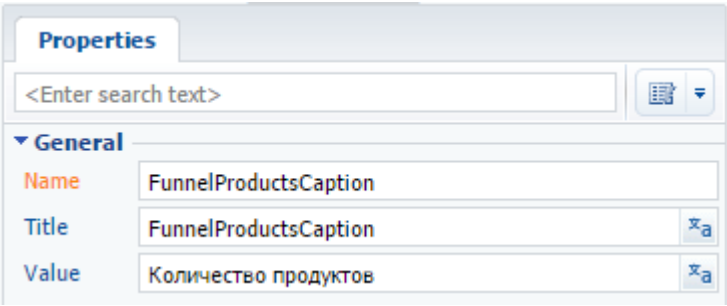
Fig. 1. Calculation provider module properties



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.
```

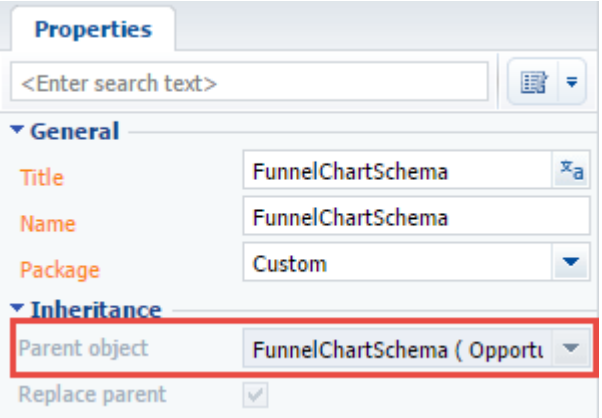
```
            // Sets the column number of products for data sampling
            addQueryColumns: function(entitySchemaQuery) {
                // Parent method calling
                this.callParent(arguments);
                // Adds the number of products column to the sample
                entitySchemaQuery.addAggregationSchemaColumn("
[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.

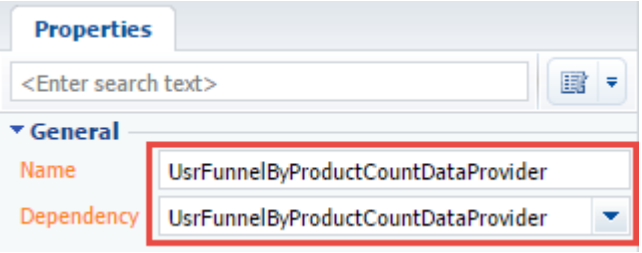
To do this, create a replacing client module and specify *FunnelChartSchema* as a parent (Fig. 3).

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



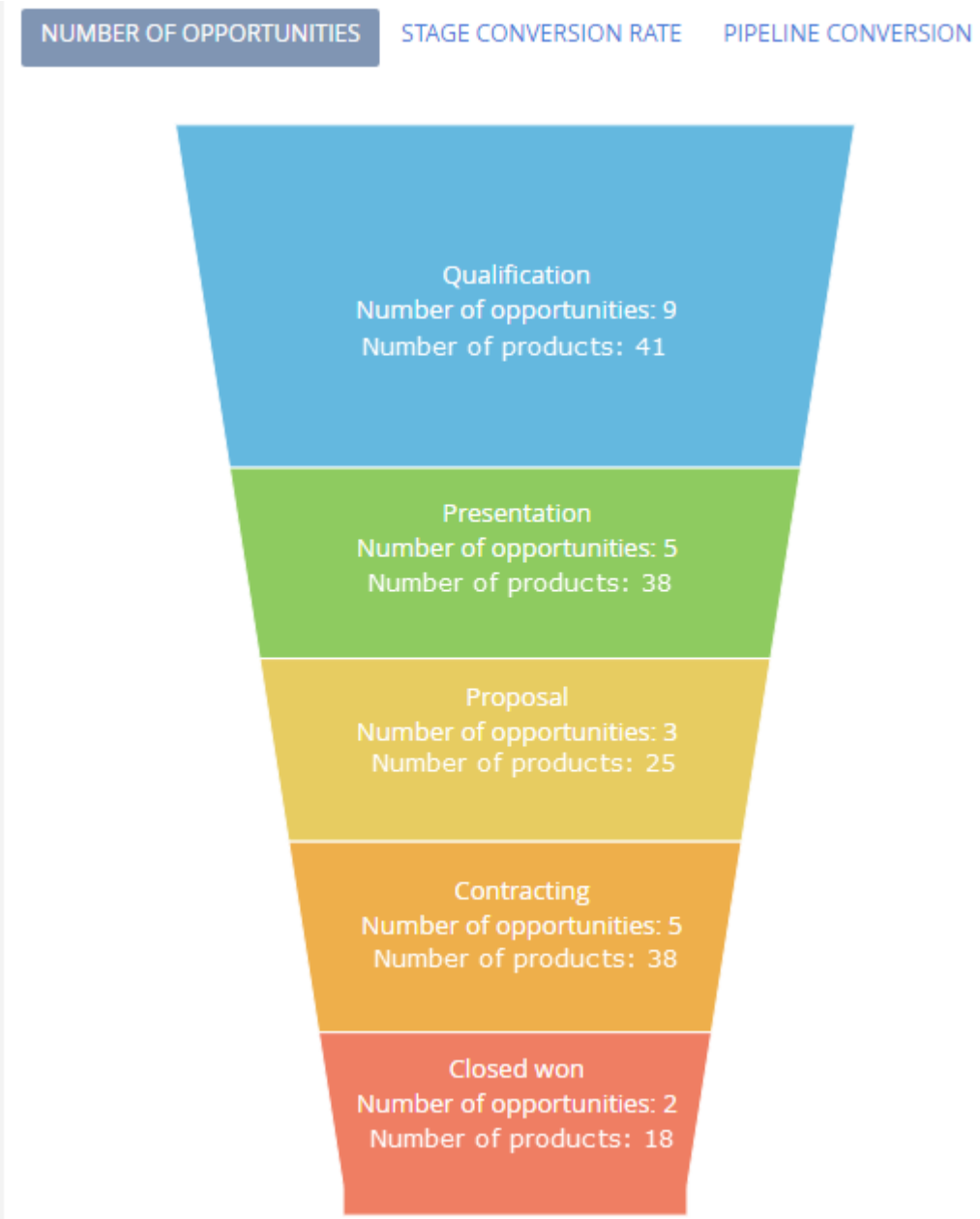
5. Specify the new calculation provider in the sales pipeline replacing schema

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 add extra filtering to a pipeline

To connect additional calculation filtering to a pipeline, you must implement the following sequence of actions.

The sequence of actions to change the calculations in a pipeline:

1. Create a new class inherited from the calculation provider in which you want to implement the required filtering logic.
2. Create a replacing FunnelChartSchema client schema and use a new calculation class in it.

Case description

You must add the "by number" filter to the sales pipeline calculation to select only those sales that have account in the [Client] field.

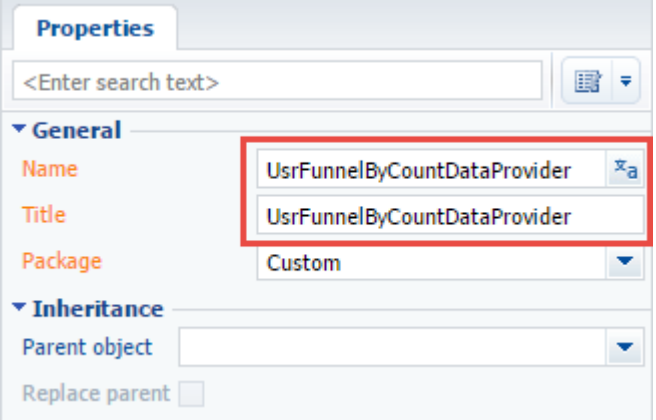
Case implementation algorithm

1. Create a new module in the custom package

Create a new calculation provider module in the custom package. The calculation provider is a class responsible for sampling, filtering and processing of the data for the pipeline chart.

Set *UsrFunnelByCountDataProvider* as the name and title for the generated module (Fig. 1).

Fig. 1. Calculation provider module properties



2. Define a new provider class and set the filter logic

To add the "by number" filter to the calculation, you need to inherit the created class from the *FunnelByCountDataProvider* class and override the *getFunnelFixedFilters* method.

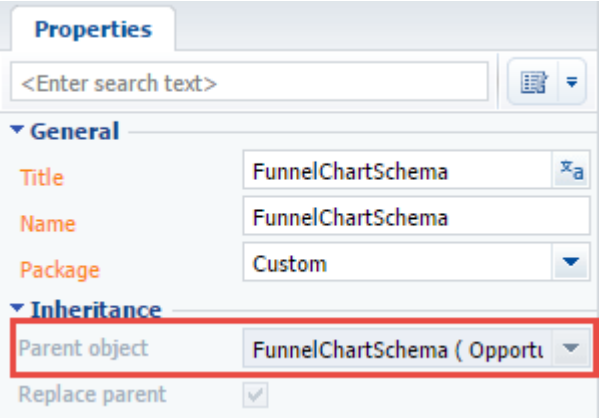
```
define("UsrFunnelByCountDataProvider", ["ext-base",
    "terrasoft", "UsrFunnelByCountDataProviderResources",
    "FunnelByCountDataProvider"],
    function(Ext, Terrasoft, resources) {
        // Defining new calculation provider.
        Ext.define("Terrasoft.configuration.UsrFunnelByCountDataProvider", {
            // Inheriting provider 'Number of opportunities'.
            extend: "Terrasoft.FunnelByCountDataProvider",
            // New provider schort name.
            alternateClassName: "Terrasoft.UsrFunnelByCountDataProvider",
            // Expanding method of the base FunnelByCountDataProvider module.
            // gets filters for selection.
            getFunnelFixedFilters: function() {
                // Calling parent method.
                var esqFiltersGroup = this.callParent(arguments);
                // Adds a filter that specifies that an account is specified by the
client in the opportunity.
                esqFiltersGroup.addItem(
                    Terrasoft.createColumnIsNotNullFilter("Account"));
                return esqFiltersGroup;
            }
        });
    });
```

Save the module.

3. Implement a module for the pipeline chart in the custom package.

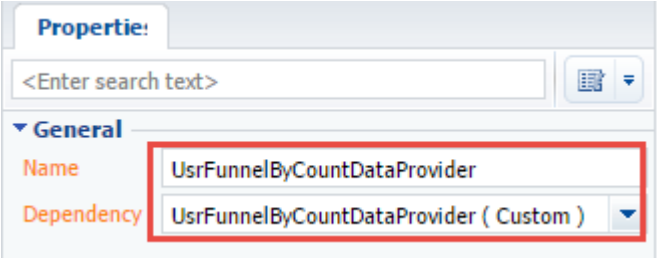
To ensure that the new provider module is used in the calculations, you need to create a replacing client module and override the method of forming the sales pipeline calculation provider. First, we need to create a replacing client module, name it *FunnelChartSchema* and set it as the parent object. (Fig. 2).

Fig. 2. Replacing module properties.



You also need to add a new calculation module to the *Dependencies* section by specifying its name in the [Dependency] and [Title] as *UsrFunnelByCountDataProvider* (Fig. 4).

Fig. 3. Pipeline dependency schema properties



4. Specify the new calculation provider in the replacing pipeline schema

To do this, replace the calculation provider class in the replacing schema to a new one.

```
define("FunnelChartSchema", ["UsrFunnelByCountDataProvider"], function() {
    return {
        entitySchemaName: "Opportunity",
        methods: {
            getProvidersCollectionConfig: function() {
                // Calls the parent method that returns the providers array.
                var config = this.callParent();
                // Searches for data provider for the "by number" filter.
                var byCount = Terrasoft.findItem(config, {tag:
"byNumberConversion"});
                // Replaces the new class.
                byCount.item.className = "Terrasoft.UsrFunnelByCountDataProvider";
                return config;
            }
        }
    };
});
```

After the schema has been saved, the new pipeline calculation module will be used in the pipeline and will display only those opportunities that have account in the [Client] field.

Adding a custom dashboard widget

General information

Dashboard widgets (analytic elements) are used for section data analysis. Go to the [Dashboards] view of a 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.
4. Add messages that are used.
5. 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

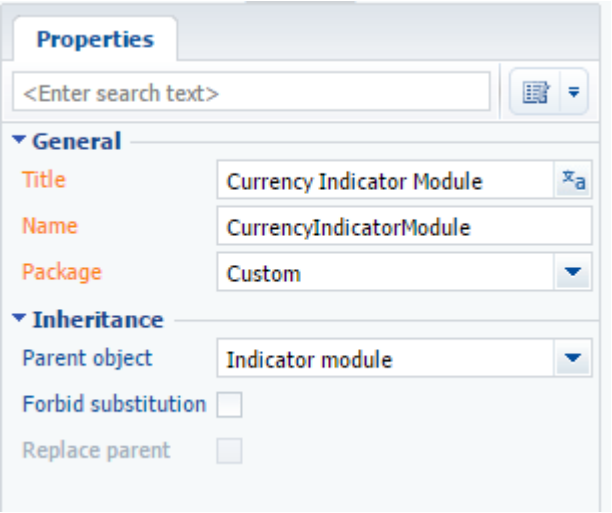
Create custom widget that shows currency exchange rate.

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 new module, you need to set the parent [Indicator module] object defined in the *NUI* package. You also need to set the "CurrencyIndicatorModule" value for the [Name] property and the "Currency Indicator Module" for the [Title] property (Fig. 1).

Fig. 1. Currency indicator module properties



2. Add the source code

To implement the necessary functionality of currency exchange rates, add the following source code to the new module.

```
define("CurrencyIndicatorModule", ["CurrencyIndicatorModuleResources",
"IndicatorModule"], function() {

    // Class that generates the configuration of the currency indicator module view.
    Ext.define("Terrasoft.configuration.CurrencyIndicatorViewConfig", {
        extend: "Terrasoft.BaseModel",
        alternateClassName: "Terrasoft.CurrencyIndicatorViewConfig",
```

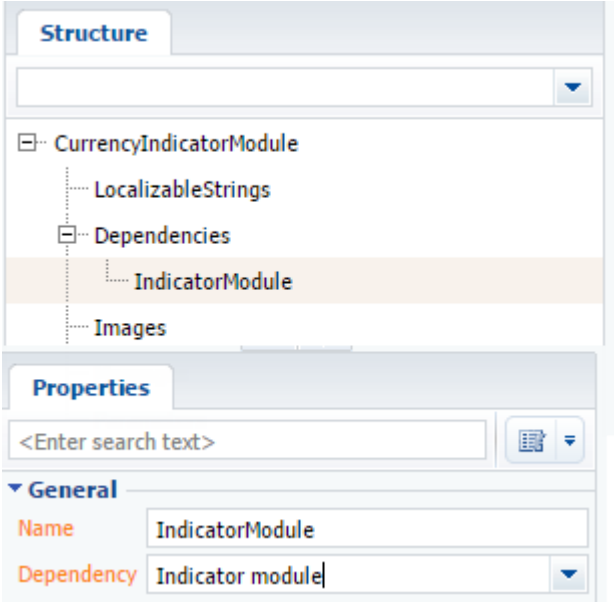
```
        // Generates the configuration of the currency indicator module view.
        generate: function(config) {
            var style = config.style || "";
            var fontStyle = config.fontStyle || "";
            var wrapClassName = Ext.String.format("{0}", style);
            var id = Terrasoft.Component.generateId();
            // The returned configuration view object.
            var result = {
                "name": id,
                "itemType": Terrasoft.ViewItemType.CONTAINER,
                "classes": {wrapClassName: [wrapClassName, "indicator-module-
wraper"]},
                "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 = 8;
            }
            if (currencyName === "EUR") {
                result = 10;
            }
            if (currencyName === "RUB") {
                result = 0.25;
            }
            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 dependencies

You need to add dependencies on the inherited indicator module to the new module.

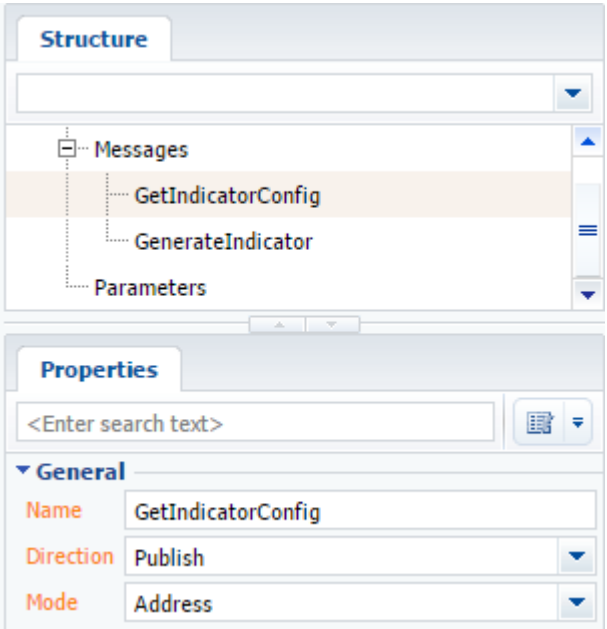
Fig. 2. The established dependence on the indicator module



4. Add messages that are used.

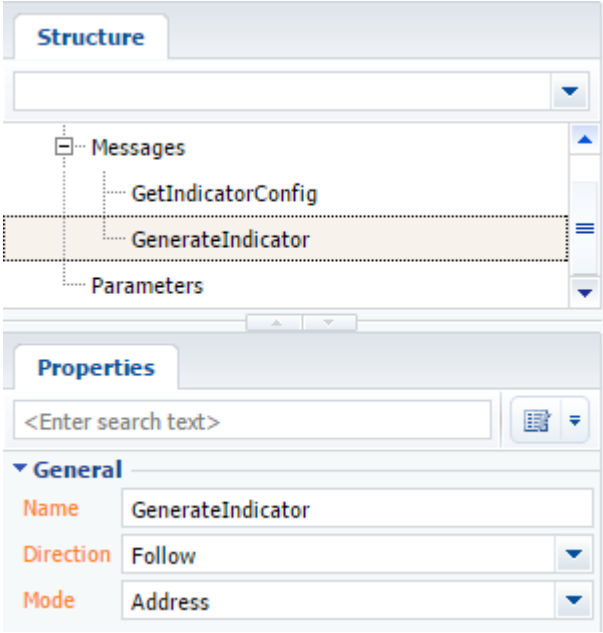
To make an indicator configuration object you need to add a *GetIndicatorConfig* address message to the module; then add the "Publish" direction for it (Fig. 3).

Fig. 3. The *GetIndicatorConfig* message



To generate the indicator view you need to add a *GetIndicator* address message to the module and to add the "Follow' direction for it (Fig. 4).

Fig. 4. *GetIndicator* message

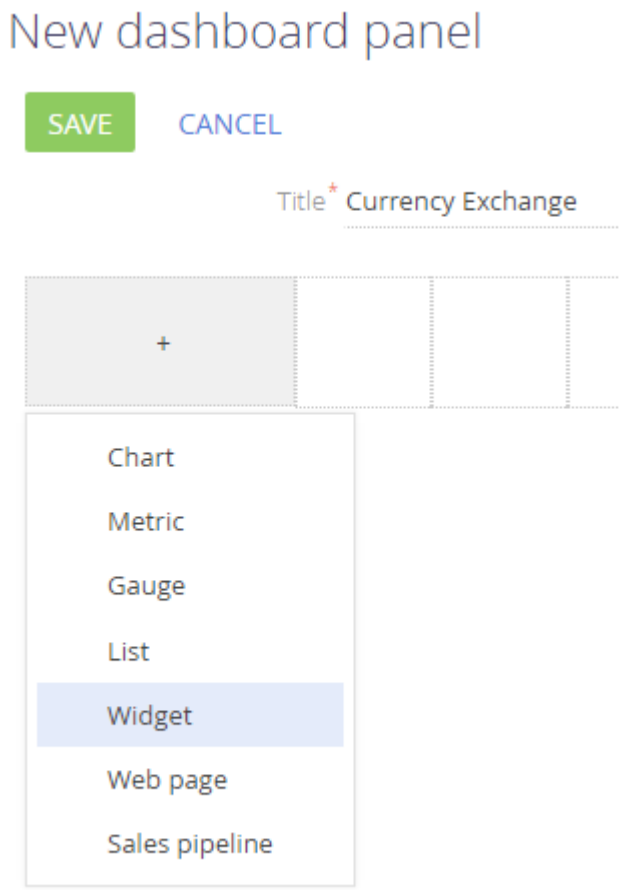


Save the new module.

5. Add the widget to the dashboard panel and set its parameters

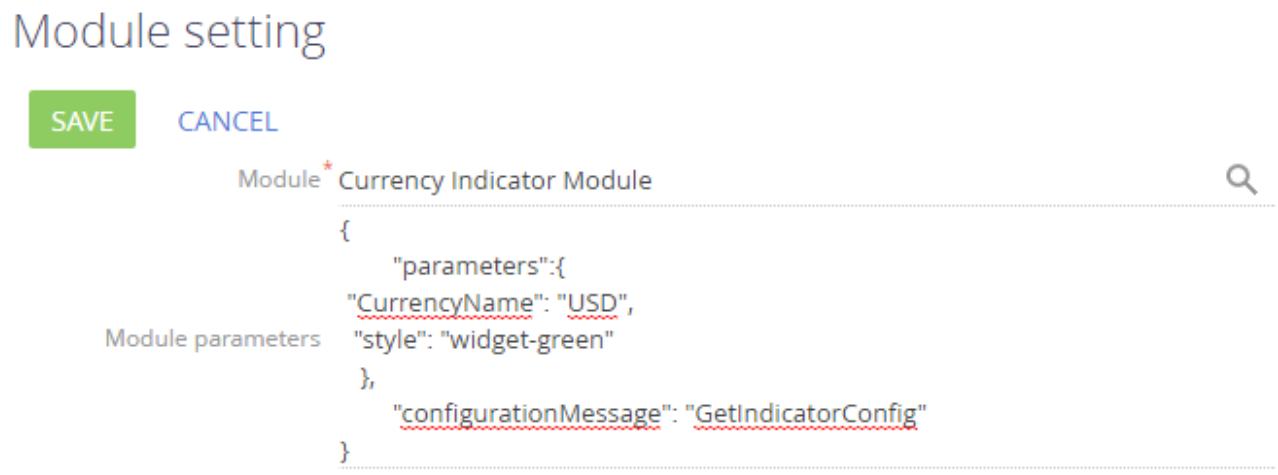
To display the widget, add it to the dashboard panel (Fig. 5).

Fig. 5. Adding the widget to the dashboard panel



In addition, you need to set the parameters of the module bound to the widget (Fig. 6).

Fig. 6. Configuration of the added widget module



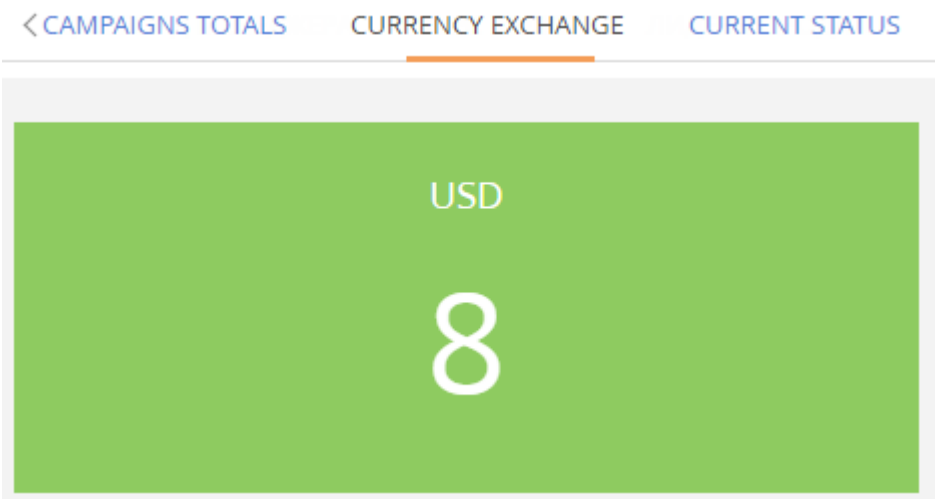
To bind the module to the added widget, add the "Currency Indicator Module" value in the [Module] field and add the configuration JSON object with the required parameters to the [Module parameters] field.

```
{
    "parameters": {
        "CurrencyName": "USD",
        "style": "widget-green"
    },
    "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.

After saving the created widget and refreshing the page, the custom widget will be displayed on the dashboards panel (Fig. 7).

Fig. 7. Currency exchange rate widget



The Terrasoft.AlignableContainer custom element

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

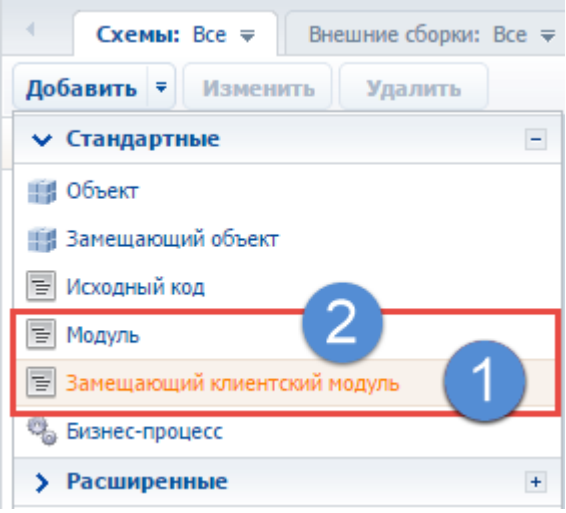
When you click on a photo in the [Contacts] section, display an enlarged image in the center of the screen with a background. When hovering over a photo, a larger version of the contact image relative to the photo container should be displayed.

Case implementation algorithm

1. Create a replacing client module

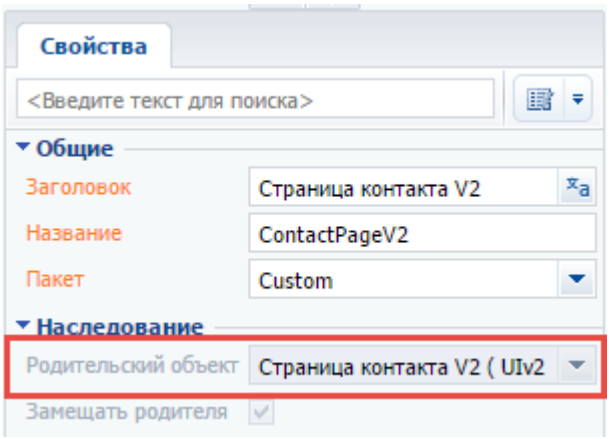
Go to the [Configuration] section, click [Add] and select [Replacing client module] (Fig. 1, 1) to create a replacing client module.

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"
```

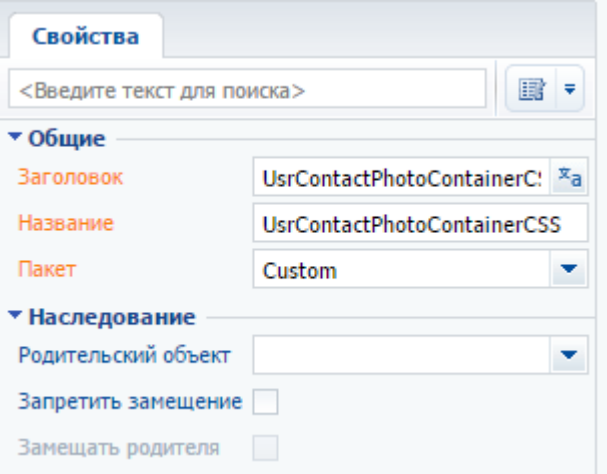
```
                    },
                    // Method-handler for pressing the button to close the element.
                    "click": {"bindTo": "close"}
                }
            },
            {
                // Inserting an element.
                "operation": "insert",
                // 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).

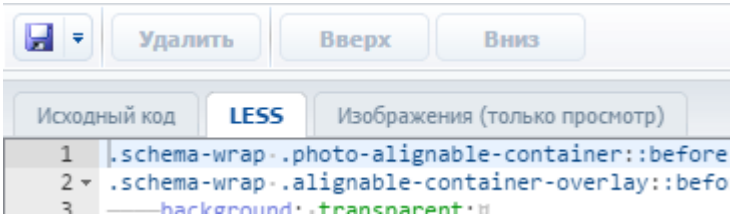
In module properties, set the title and header to "UsrContactPhotoContainerCSS" (Fig. 3).

Fig. 3. Module properties



Module styles are defined on the LESS tab (Fig. 4).

Fig. 4. LESS module tab

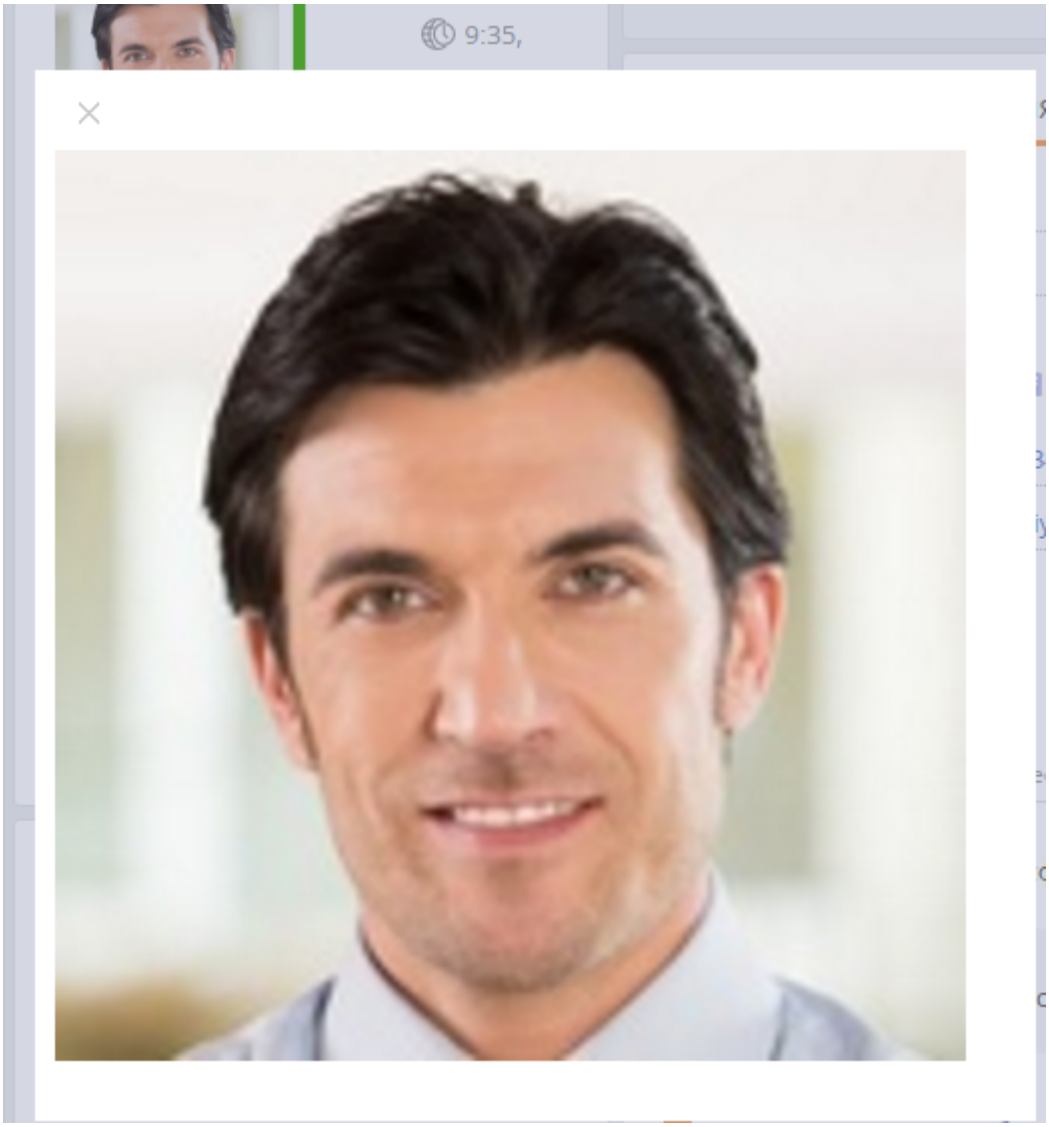


Add the following CSS selectors for the contact photo to display correctly in the screen center:

```
.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.
```

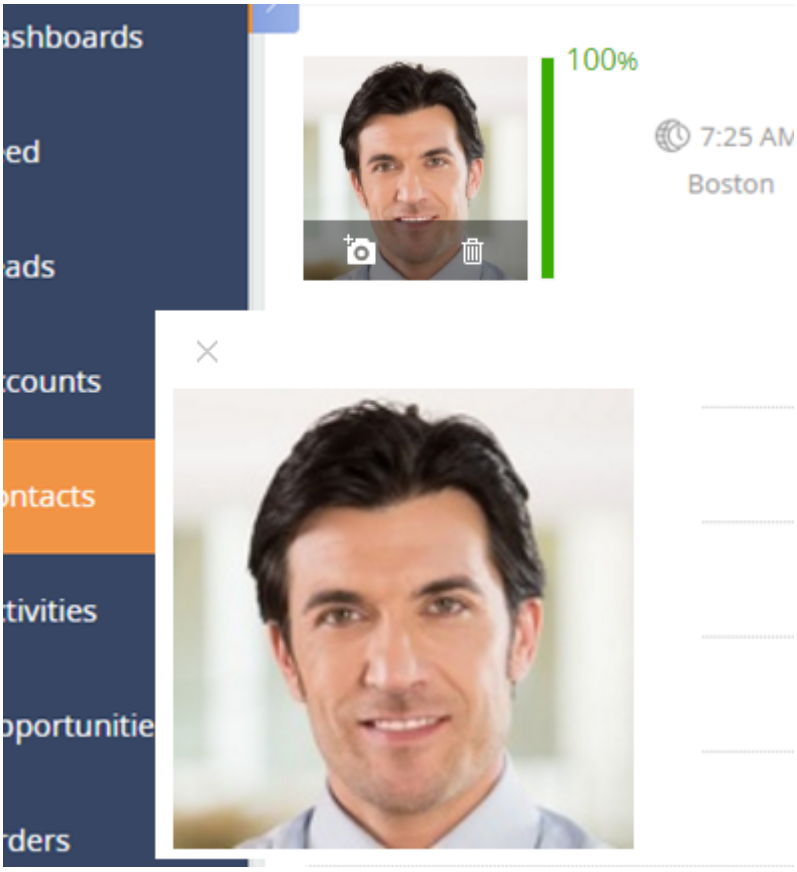
```
            "imageSrc": {"bindTo": "getContactImage"}
        }
    },...
]/**SCHEMA_DIFF*/
```

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

The deduplication uses the rules that are created in the form of stored procedures. When executed, each rule fills the *ContactDuplicateSearchResult* table with the list of duplicates. The search results are grouped by rules and displayed on the duplicate search page. More information about this functionality can be found in the "Finding and merging duplicates" article in the user guide.

The algorithm for adding a custom rule of the duplicates search

1. If necessary, add a column to the object schema, which will be used to search for duplicates.
2. Add the stored search procedure in the application database.
3. Register the stored procedure as a new rule.

The example of adding a custom rule of the duplicates search

Case description

During the deduplication process, contacts that have the same values in the [Taxpayer ID] column should be counted as duplicates and displayed in the search results.

Case implementation algorithm

1. Adding a field, value of which will be searched for duplicates

The [Taxpayer ID] field does not exist on the contact edit page by default and you need to create it using the section wizard. You need to add a field with "Taxpayer ID" value in the [Title] property and "UsrInn" value in the [Name in DB] property to the page of the [Contacts] section. The wizard will add the required column to the object automatically. More information about how to add a field to the page can be found in the "How to set up the page fields" article.

2. Adding the stored search procedure in the application database.

To search for duplicates in a specific section, add the corresponding stored procedure to the database. To add a stored procedure for searching duplicates by the [Taxpayer ID] field, you need to execute the following SQL query:

```
-- Check for the existence of a stored procedure named tsp_FindContactDuplicateByInn.
IF NOT OBJECT_ID('[dbo].[tsp_FindContactDuplicateByInn]') IS NULL
BEGIN
    -- Deleting a stored procedure.
    DROP PROCEDURE [dbo].[tsp_FindContactDuplicateByInn];
END;
GO
-- Creating stored procedure.
CREATE PROCEDURE [dbo].[tsp_FindContactDuplicateByInn] (
    -- This table parameter is transmitted only if a new contact is saved.
    -- Contains data of a new contact.
    --In the case of starting a global search for duplicates, the passed parameter
does not contain data.
        @parsedConfig CreatingObjectInfo READONLY,
    -- The unique id of the user who launched the duplicate search.
    @sysAdminUnit UNIQUEIDENTIFIER,
    -- The ID of the current rule from the table [ContactDuplicateSearchResult].
    -- This ID is created after the registration of the rule in the system.
    @ruleId UNIQUEIDENTIFIER
)
AS
BEGIN
    -- Getting the number of records from the received table to define if a global
duplicates search is launched
        DECLARE @parsedConfigRowsCount INT = (SELECT COUNT(*) FROM @parsedConfig);
    --  Create a temporary table with contacts data for searching.
    CREATE TABLE #searchContact (
        [UsrInn] INT,
        [SortDate] DATETIME
    );
    -- In case of the global search, the temporary table is populated with data.
    IF @parsedConfigRowsCount = 0
    BEGIN
        -- Adding data to the temporary table to find duplicates.
        INSERT INTO #searchContact ([UsrInn], [SortDate])
```

```
        -- Request a selection of contacts data.
        SELECT
            -- Select of the Taxpayer ID dates of the contact modification.
            [UsrInn],
            MAX([ModifiedOn])
        FROM [Contact]
        -- The grouping by fields is added for the possibility of using the check for
quantity.
        GROUP BY [UsrInn]
        -- The table is filled only if there is more than one contact.
        HAVING COUNT(*) > 1;
    END;

    -- Filling out the results table.
    INSERT INTO [ContactDuplicateSearchResult] ([ContactId], [GroupId], [RuleId],
[SysAdminUnitId])
    SELECT
        -- Contact duplicate ID.
        [vr].[Id],
        -- Forming a group number.
        DENSE_RANK() OVER (ORDER BY [vr].[SortDate] DESC, [vr].[UsrInn]),
        -- Rule ID.
        @ruleId RuleId,
        -- The ID of the user who started the duplicate search process.
        @sysAdminUnit
    FROM (
        -- A subquery that fills out the duplicates table.
        SELECT
            -- Contact ID.
            [v].[Id],
            --Contact Taxpayer ID.
            [v].[UsrInn],
            -- Sorting date.
            [r].[SortDate]
        -- The data source tables.
        FROM [Contact] [v], #searchContact r
        -- The rule that determines that the contacts are duplicates.
        WHERE [v].[UsrInn] = [r].[UsrInn]
        -- Grouping the search result.
        GROUP BY [v].[UsrInn], [r].[SortDate], [v].[Id]
    ) [vr];
END;
GO
```

3.

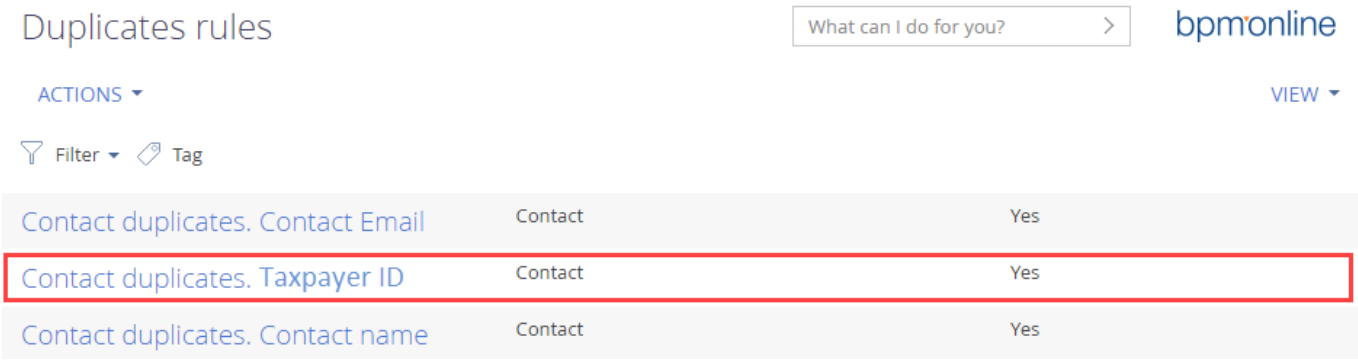
Registration of the stored procedure as a new rule

To register the stored procedure as a new rule of the duplicates search, add a corresponding record to the *DuplicatesRule* table. To do this, execute the following SQL query:

```
-- A variable that stores the value of the UId column of the Contact schema.
DECLARE @ContactUId UNIQUEIDENTIFIER;

-- Gets the value of the column UId 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', 'Contacts duplicates.Taxpayer
ID');
```

To apply the changes you need to refresh the section page and empty the browser cache. After that, a new rule will appear in the rules search list (Fig.1).

Fig. 1. Duplicates search rule by [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*.

Analytics

- **How to create own macros for a MS Word printable**

How to create own macros for a MS Word printable

We recommend familiarization with the setup of MS Word printables detailed in the article The MS Word printables setup.

In most cases, a printable can be set up using standard tools of the BPMonline MS Word Report Designer. To implement special tasks for printables setup, macros must be used. A macro for the printable setup is a class implementing the ***IExpressionConverter*** interface.

To call the created macro from the printable template, mark it with the ***ExpressionConverterAttribute*** attribute, with the indication of the macro name, for example:

```
[ExpressionConverterAttribute("CurrentUser")]
```

The ***Evaluate(object value, string arguments = "")***interface method must be implemented in the class. The method accepts the field value from the printable template as an argument and returns the value of the string type; the latter will replace this field in the finished printable.

Algorithm of creating a custom macros for a printable:

1. Create a printable and add the [ID] column to the list of the printable columns. This column will be an incoming parameter for the macros.
2. Add the source code module to the custom package and describe the inheritor class of the IExpressionConverter interface. The class must be marked with the ExpressionConverterAttribute attribute with the macro name and the Evaluate(object value, string arguments = "") method must be implemented in it.
3. Add a tag with the macro name in the [#MacrosName#] format to the [ID] column in the printable template.

4. Publish the source code module and upload the updated template to the printable.

An example of macro creation for the printable is provided below.

Case description

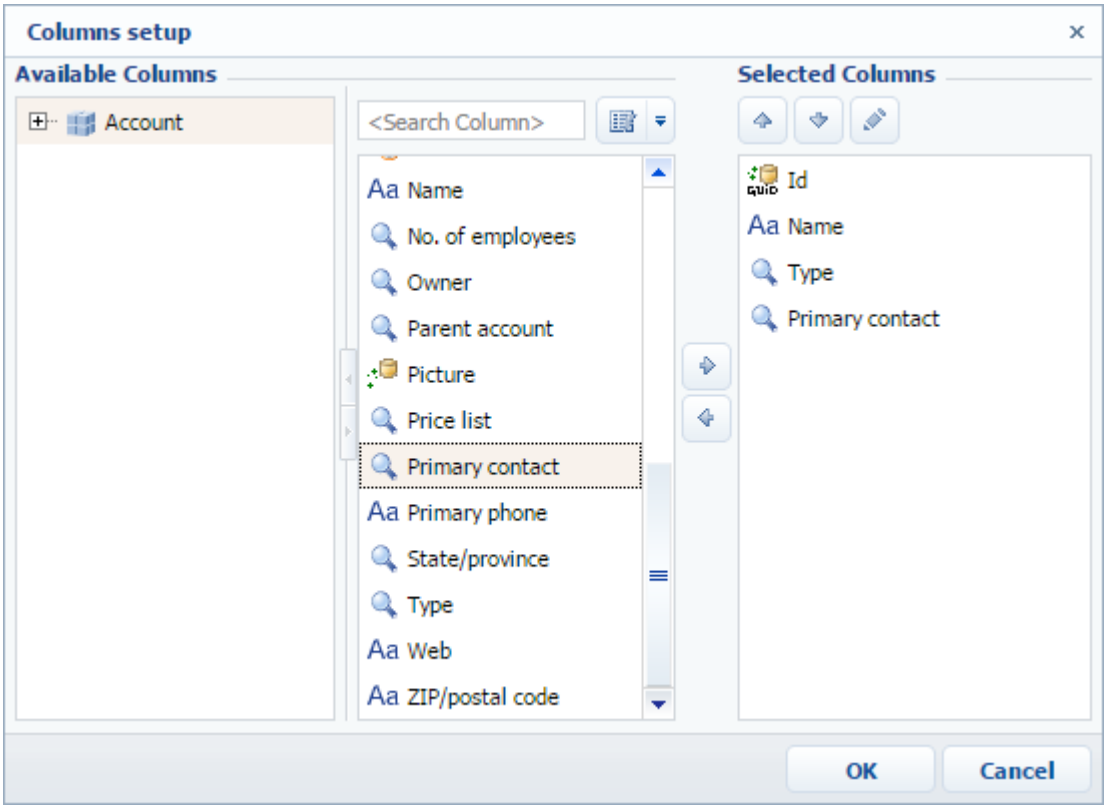
Create the [Account summary] printable for the edit page of the [Accounts] section. This printable will contain such common fields as [Name], [Type] and [Primary Contact], and also the [Additional Information] field where the annual turnover will be displayed for accounts of the [Customer] type and the number of employees will be displayed for accounts of the [Partner] type. In addition, the printable must contain information on when it was generated and and the name of the user contact who generated it.

Case implementation algorithm

1. Create the [Account Summary] printable for the edit page of the [Accounts] section

Add [Id], [Name], [Type], and [Primary Contact] columns to the list of printable columns (Fig. 1).

Fig. 1. – List of printable columns



The printable is detailed in the article [The MS Word printables setup](#).

2. Download the printable template and allocate fields on its page

3. Add three modules of the source code to the custom package

The macros code for receiving additional information depending on the type of the account is as follows:

```
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;

    // Attribure with macros name [AccountInfoByType].
    [ExpressionConverterAttribute("AccountInfoByType")]
    // The class must implement the IExpressionConverter nterface.
    class AccountInfoByTypeConveter : IExpressionConverter
    {
        private UserConnection _userConnection;

        // Implementation of 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 for receiving additional information depending on the type of the
account.
        // The account Id is used as the incoming parameter.
        private string getAccountInfo(Guid accountId)
        {
            try
            {
                /// Creating an object of the EntitySchemaQuery class with the
[Account] core schema/
                EntitySchemaQuery esq = new
EntitySchemaQuery(_userConnection.EntitySchemaManager, "Account");
                // Adding the [Name] column from the [Type] lookup field to the
schema.
                var columnType = esq.AddColumn("Type.Name").Name;
                // Adding the [Name] column [EmployeesNumber] lookup field to the
schema.
                var columnNumber = esq.AddColumn("EmployeesNumber.Name").Name;
                // Adding the [Name] column [AnnualRevenue] lookup field to the
schema.
                var columnRevenue = esq.AddColumn("AnnualRevenue.Name").Name;
                // Records are filtered by the account Id.
                var accountFilter =
esq.CreateFilterWithParameters(FilterComparisonType.Equal, "Id", accounted);
                esq.Filters.Add(accountFilter);
                // Receiving the entities collection.
```

```
                EntityCollection entities = esq.GetEntityCollection(_userConnection);
                // If the collection contains items, the method returns
                // corresponding information depending on the account type.
                if (entities.Count > 0)
                {
                    Entity entity = entities[0];
                    var type = entity.GetTypedColumnValue<string>(columnType);
                    switch (type)
                    {
                        case "Client":
                            return String.Format("Annual turnover {0} mln. roubles",
                                entity.GetTypedColumnValue<string>(columnRevenue));
                        case "Partner":
                            return String.Format("Number of employees {0} persons",
                                entity.GetTypedColumnValue<string>(columnNumber));
                        default:
                            return String.Empty;
                    }
                }
                return String.Empty;
            }
            catch (Exception err)
            {
                throw err;
            }
        }
    }
}
```

Below is the macros 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;

    // Attribute with the [CurrentDate] macros name.
    [ExpressionConverterAttribute("CurrentDate")]
    // The class must implement the IExpressionConverter interface.
    class CurrentDateConveter : IExpressionConverter
    {
        private UserConnection _userConnection;

        // Implementation of 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;
            }
        }
    }
}
```

Below is the macros code for determining the current user (the user who generated the printable):

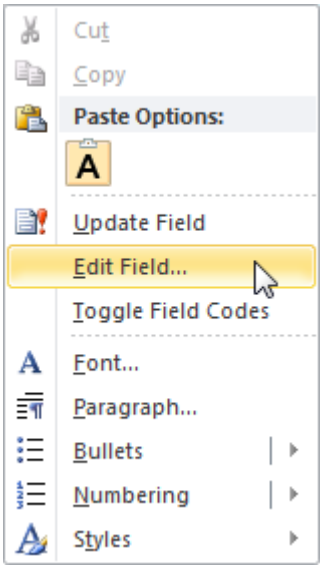
```
namespace Terrasoft.Configuration
{
    using System;
    using System.CodeDom.Compiler;
    using System.Collections.Generic;
    using System.Data;
    using System.Linq;
    using System.Runtime.Serialization;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    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;

    // Attribute with the [CurrentUser] macros name.
    [ExpressionConverterAttribute("CurrentUser")]
    // The class must implement the IExpressionConverter interface.
    class CurrentUserConveter : IExpressionConverter
    {
        private UserConnection _userConnection;
        // Implementation of 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 user contact.
                return _userConnection.CurrentUser.ContactName;
            }
            catch (Exception err)
            {
                return err.Message;
            }
        }
```

```
        }
}
```

4. Add tags with macros names to the report template fields

For this purpose, select the [Edit Field...] item in the right-click menu of the template field (Fig. 2).

Fig. 2. — Right-click menu of the template field



Add tags to the field names (Fig. 3, 4, 5).

Fig. 3. — Adding the AccountInfoByType tag for receiving additional information about the account

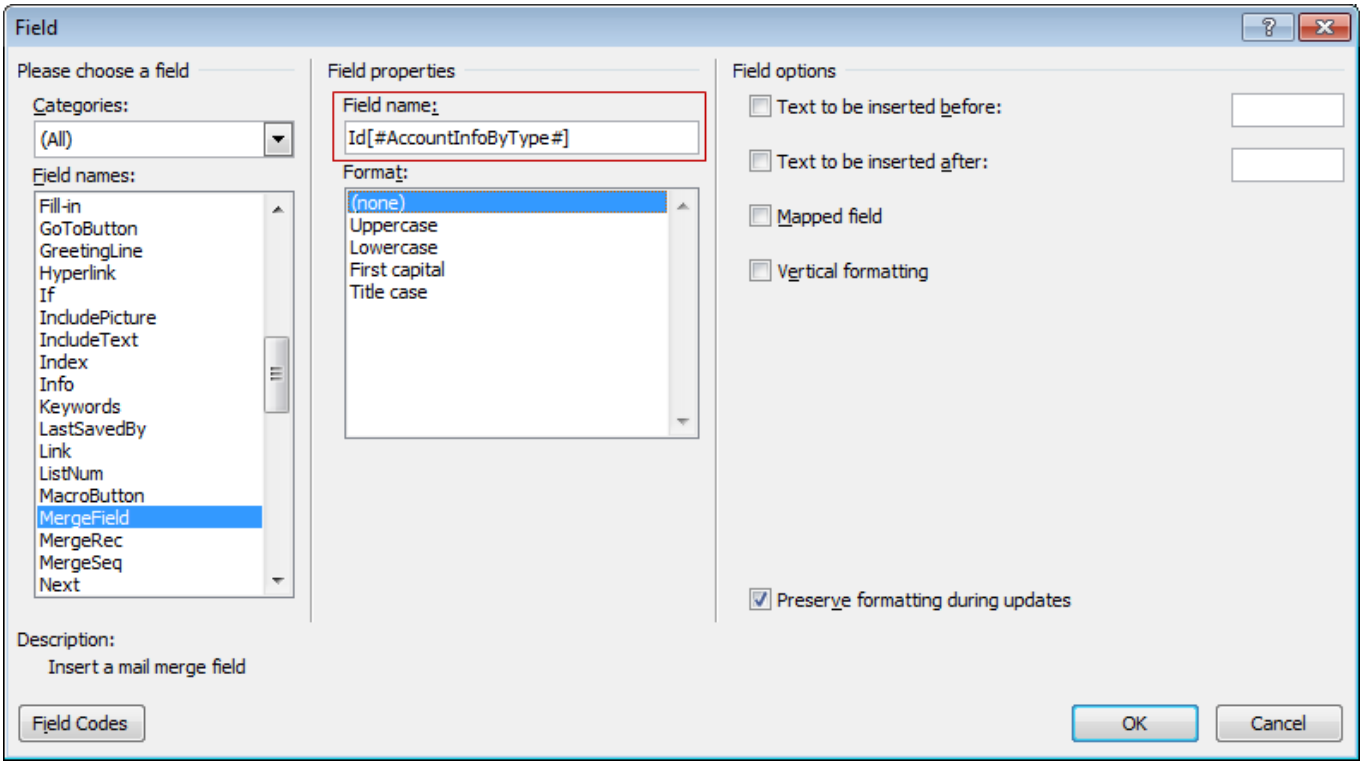


Fig. 4. — Adding the CurrentDate tag for receiving the current date

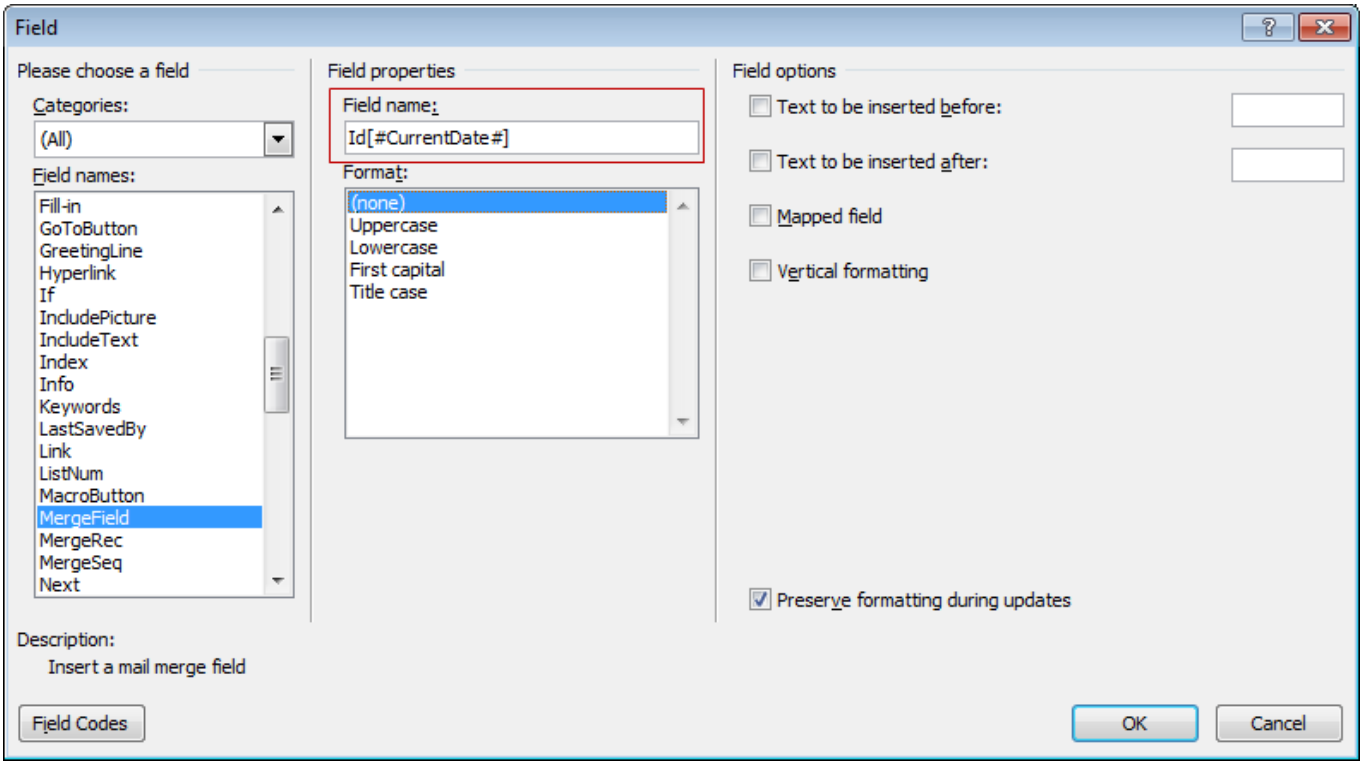
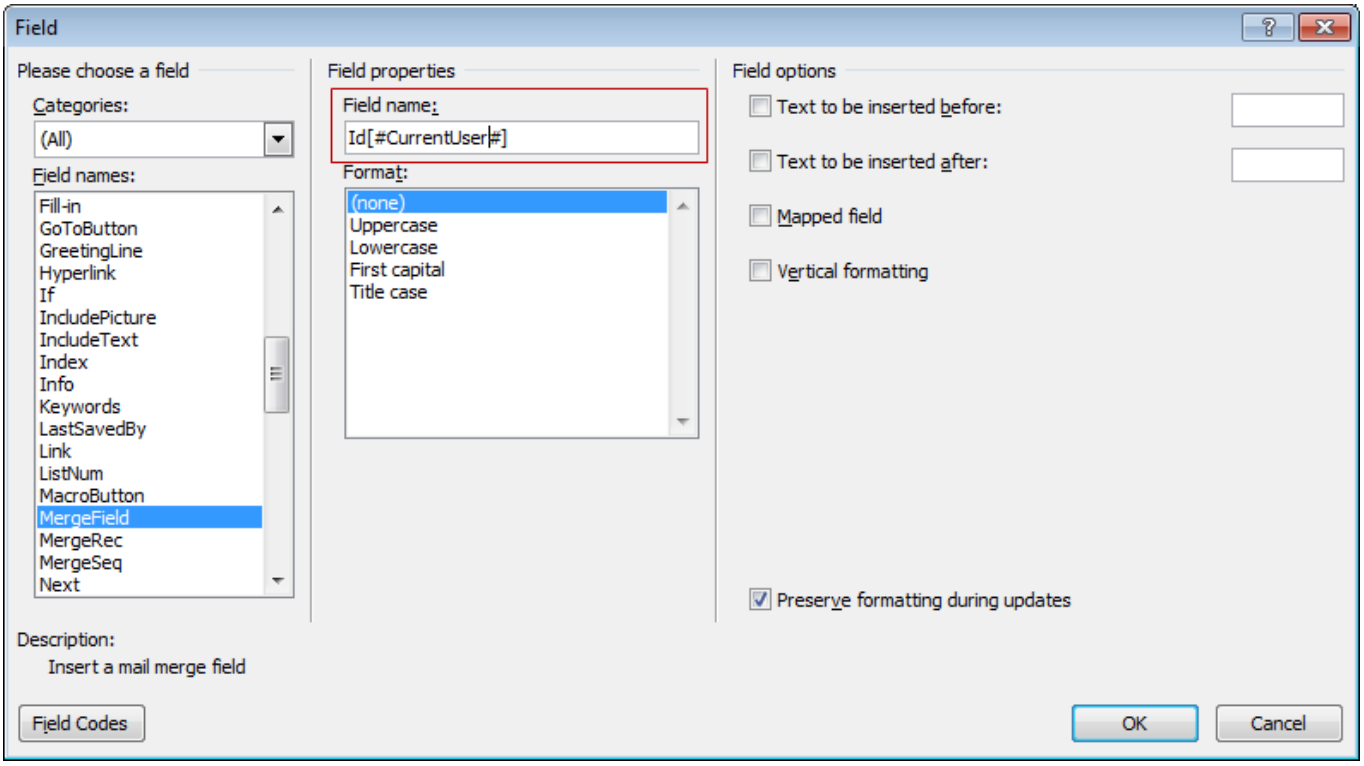


Fig. 5. — Adding the CurrentUser tag for receiving the current user contact



As a result, the template will look as shown in Fig. 6.

Fig. 6. – Printable template

Account summary

Name	«Name»
Type	«Type»
Primary contact	«Primary contact»
Additional info	«Id[#AccountInfoByType#]»

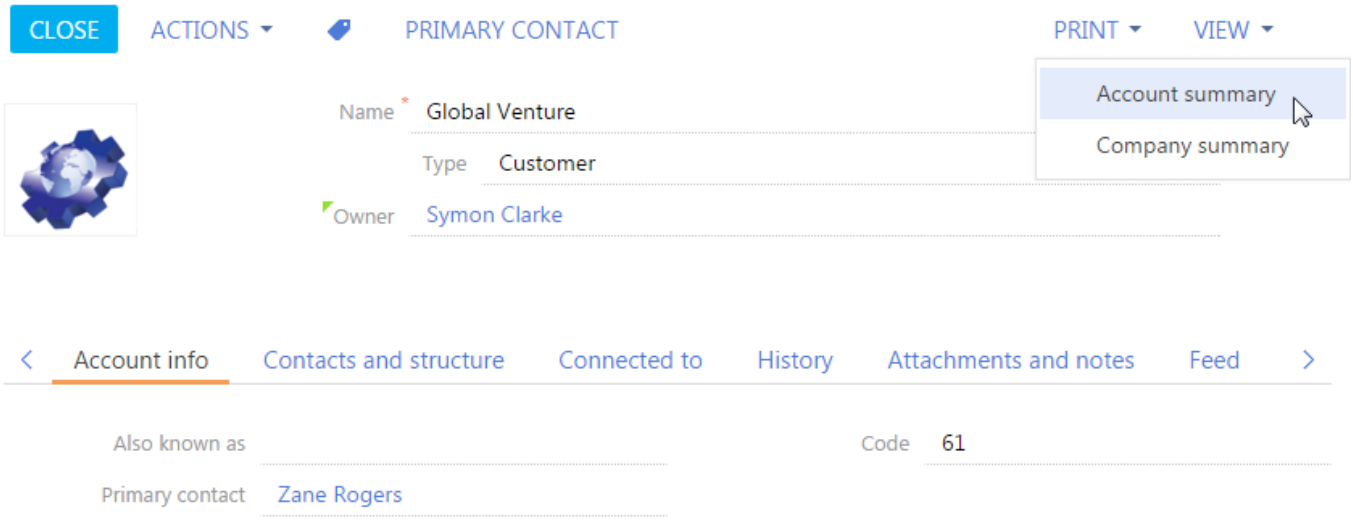
Date	«Id[#CurrentDate#]»
Created by	«Id[#CurrentUser#]»

5. Save the template and upload it to the printable

6. Publish the source code modules

When the template is uploaded and the source code modules are published on the account edit page, a new [Account summary] printable will appear in the [Print] list (Fig. 7).

Fig. 7. — New printable in the [Accounts] section



Data in the template will depend on the type of the account (Fig. 8, 9).

Fig. 8. — Printable for the account of the [Customer] type

Account summary

Name	Global Venture
Type	Customer
Primary contact	Zane Rogers
Additional info	Annual revenue 16 – 20 million

Date 5 Jan 2016
Created by John Best

Fig. 9. — Printable for the account of the [Partner] type

Account summary

Name	Feature IT
Type	Partner
Primary contact	Tony Campbell
Additional info	No. of employees 201-500

Date 5 Jan 2016
Created by John Best

Working with data

- **CRUD-operations in cofiguration**
- **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 cofiguration

- **The use of EntitySchemaQuery implementation on client**

The use of EntitySchemaQuery implementation on client

Content

- **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

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	Description		
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);
```

Table 1. — Query result getting method

Method	Description		
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	Description
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 esqContact = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
esqContact.addColumn("Name");
esqContact.addColumn("Country.Name", "CountryName");

// Creation of the first filter instance.
var esqFirstFilter = esqContact.createColumnFilterWithParameter(Terrasoft.ComparisonType.EQUAL,
"Country.Name", "Mexico");

// Creation of the second filter instance.
var esqSecondFilter = esqContact.createColumnFilterWithParameter(Terrasoft.ComparisonType.EQUAL,
"Country.Name", "USA");

// Filters will be updated by OR logical operator in query filters collection.
esqContact.filters.logicalOperation = Terrasoft.LogicalOperatorType.OR;

// Adding created filters to collection.
esqContact.filters.add("esqFirstFilter", esqFirstFilter);
esqContact.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 esqContact = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
esqContact.addColumn("Name");
esqContact.addColumn("Country.Name", "CountryName");

// Select all contacts where country is not specified.
var esqFirstFilter = esqContact.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 = esqContact.createColumnBetweenFilterWithParameters("BirthDate", dateFrom, dateTo);

// Add created filters to query collection.
esqContact.filters.add("esqFirstFilter", esqFirstFilter);
```

```
esqContact.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

- **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);
```

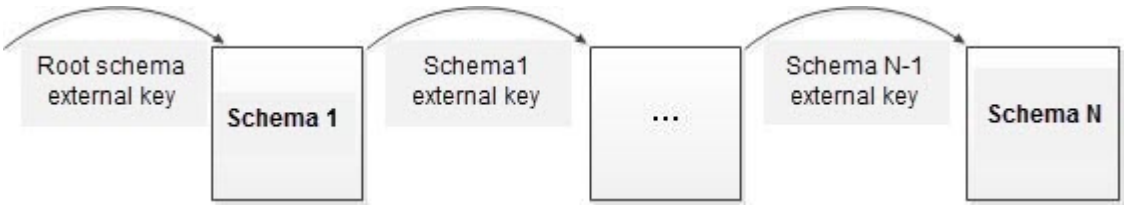
Definition of root schema. Creation of paths to columns against root schema. Examples.

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

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

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

- **How to create own configuration service**
- **How to call a configuration service**

How to create own configuration service

General

Bpm'online service model implement a basic set of web-services that can be used for integration of bpm'online with external applications and systems. Examples of system services are EntityDataService.svc that provides possibilities for data communication with bpm'online through OData protocol and also ProcessEngineService.svc that can run bpm'online business processes from external applications.

You can create your user web-services in the configuration, used for implementation of specific integration tasks.

A configuration web-service is represented by RESTful-service, implemented on the basis of WCF technology.

Configuration service creation procedure.

To create your own web-services in the configuration, take the following actions:

1. Create source code schema using C# language in the configuration.
2. Add links to namespaces such as System.ServiceModel, System.ServiceModel.Web, System.ServiceModel.Activation in the 'usings' block.
3. Create service class in the Terrasoft.Configuration namespace, create service class and place it with the [ServiceContract] and [AspNetCompatibilityRequirement] attributes with parameters (see implementation example).
4. Add realization of service methods to created class. Each service method should be marked with the [OperationContract] and [WebInvoke] attributes with parameters (see implementation example).
5. If necessary, you can implement additional classes, i.e. objective data types that will be processed by your service. Each class should be marked with the [DataContract] attribute and class fields should be marked with the [DataMember] attribute.
6. Execute publication of source code schema.

After publication of schema, the created web-service will be available for calling of configuration schemas from a source code and also external applications. For more details about calling of configuration web-services see the **How to call a configuration service** article.

Configuration service example

This example shows the creation of a configuration service using one method. This method accepts one input parameter of string type and returns the string.

Full program web-service code is shown below:

```
// Configuration service class should be implemented in Terrasoft.Configuration name
space.
// If necessary, you can create your own workspace for a web-service but it should be
// attached to Terrasoft.Configuration .
namespace Terrasoft.Configuration.CustomConfigurationService
{
    using System;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;

    // Service class is marked with [ServiceContract] compulsory attributes and
    // [AspNetCompatibilityRequirements] with parameters.
    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class CustomConfigurationService
    {
        // Service methods are marked with compulsory attributes [OperationContract]
and
        //[WebInvoke] with parameters.
        [OperationContract]
        [WebInvoke(Method = "POST", RequestFormat = WebMessageFormat.Json, BodyStyle
= WebMessageBodyStyle.Wrapped,
        ResponseFormat = WebMessageFormat.Json)]
```

```
        public string GetTransformValue(string inputParam)
        {
            var result = inputParam + " + output string";
            return result;
        }
    }
}
```

How to call a configuration service

Calling of a service from code of configuration schemas

To call a configuration web-service from client JavaScript code, take the following actions:

1. Connect the ServiceHelper module to a module of the page, from which the service is called. This module provides a convenient interface for execution of queries to the server through Terrasoft.AjaxProvider query provider, implanted in the client core.
2. all the *callService(serviceName, serviceMethodName, callback, serviceData, scope)* method from ServiceHelper module by transferring the following parameters to it (Table 1).

Table 1. — callService() method parameters

Parameter	Description
serviceName	Is a name of a configuration service. It corresponds to the class name in C# that implements the service.
servcieMethodName	Is a name of a called method of configuration service. The method can accept input parameters and return resultant values.
callback(response)	Is a function of callback where a service response is processed. The function accepts response object as a parameter. If a called service method returns any value, you can get it in the client through the response object property. The name of the property, where the method output-value is returned, is formed by the following rule: [Service method name]+[Result]. For example, if GetSomeValue() method is called then the returnable value will be included into the response.GetSomeValueResult property.
scope	Is an execution context.

Example of service calling from configuration schema

It is required to add a button on the new contact creation page. Clicking on the button will call a configuration web-service. The results, returnable by web-service method, will be displayed in the info window.

The example uses a web-service from an example, which is described in the **How to create own configuration service** article.

Case implementation algorithm

1. Create replacing contact edit page and add a button to it

The process of adding a button to the edit page is described in the article **How to add a button to an edit page in the new record add mode**.

Add localized GetServiceInfoButtonCaption string with title name, for example "Call service" to the schema of replacing module of contact edit page.

Connect the ServiceHelper module as a dependency in the declaration of the edit page module.

2. Add button handler and call web-service method in it

Use the callService method of the ServiceHelper module to call a web-service. Transfer the name of configuration service class, name of called service method, object with initialized input parameters for service method, callback function where you have to process server results, and also execution context to the ServiceHelper module.

Source code of replacing module of edit page is shown below.

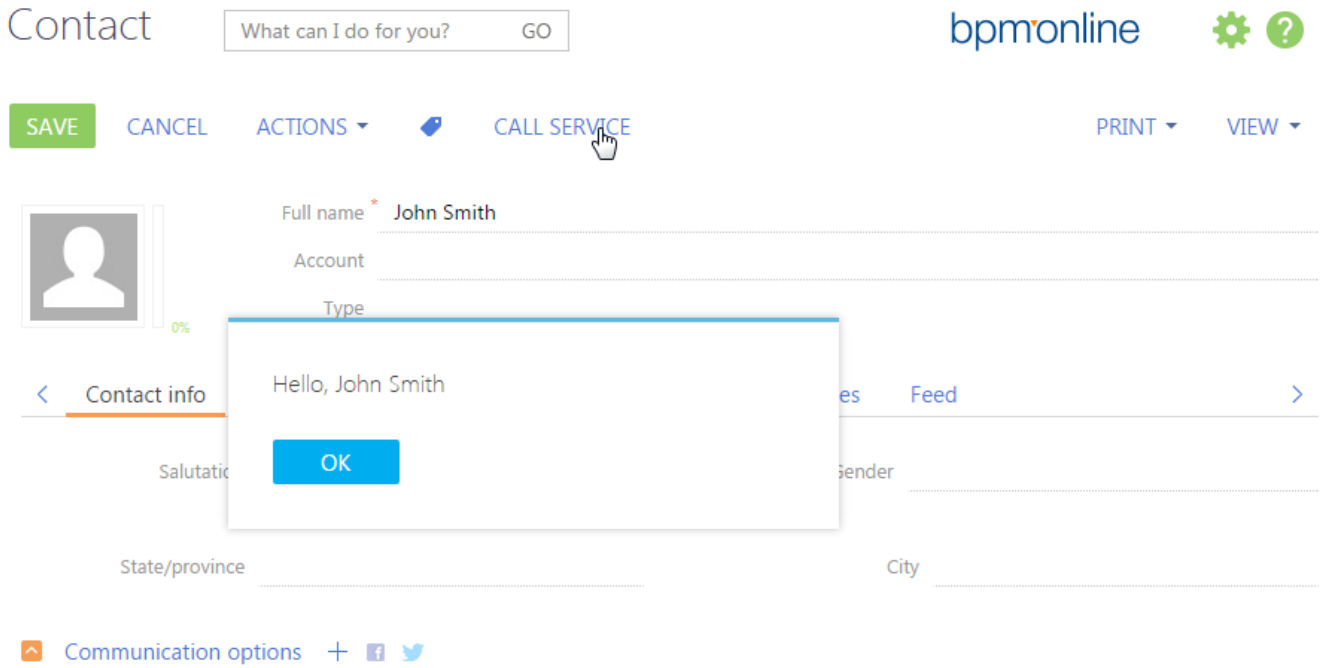
```
define("ContactPageV2", ["ServiceHelper"],
    function(ServiceHelper) {
        return {
            // Name of edit page object schema.
            entitySchemaName: "Contact",
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            // Collection of methods of edit page view model.
            methods: {
                // Method verifies whether the [Full name] field of the page is
completed.
                isContactNameSet: function() {
                    return this.get("Name") ? true : false;
                },
                // The method-handler of button clicking.
                onGetServiceInfoClick: function() {
                    var name = this.get("Name");
                    // Object that initializes input parameters for service method.
                    var serviceData = {
                        // Property name coincides with name of input parameter of
service method.
                        inputParam: name
                    };
                    // Calling of web-service and processing of results.
                    ServiceHelper.callService("CustomConfigurationService",
"GetTransformValue",
                        function(response) {
                            var result = response.GetTransformValueResult;
                            this.showInformationDialog(result);
                        }, serviceData, this);
                }
            },
            diff: /**SCHEMA_DIFF*/[
                // Metadata for adding new control item, i.e. user button, to the
page.
                {
                    //It specifies that adding item to the page is performed.
                    "operation": "insert",
                    //Metaname of parent control item, to which the button is added.
                    "parentName": "LeftContainer",
                    // It indicates that the button is added to collection of control
items of
                    // parent item (metaname of which is indicated in parentName.
                    "propertyName": "items",
                    //Metaname of added column.
                    "name": "GetServiceInfoButton",
                    //Additional properties of the field.
                    "values": {
                        //Type of added item - button.
                        itemType: Terrasoft.ViewItemType.BUTTON,
                        // Linking of column title to localizable schema string.
                        caption: {bindTo:
"Resources.Strings.GetServiceInfoButtonCaption"},
                        //Linking of method -handler of button clicking
                        click: {bindTo: "onGetServiceInfoClick"},
                        //Linking of button accessibility property.
                        enabled: {bindTo: "isContactNameSet"},
```

```
                        // Setting of filed layout.
                        "layout": {"column": 1, "row": 6, "colSpan": 2}
                    }
                }
            ]/**SCHEMA_DIFF*/
        };
    });
```

3. Save a scheme in the configuration

Case execution result is shown on Fig. 1.

Fig. 1. — Result of configuration service calling case



Reading multilingual data with EntitySchemaQuery

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.

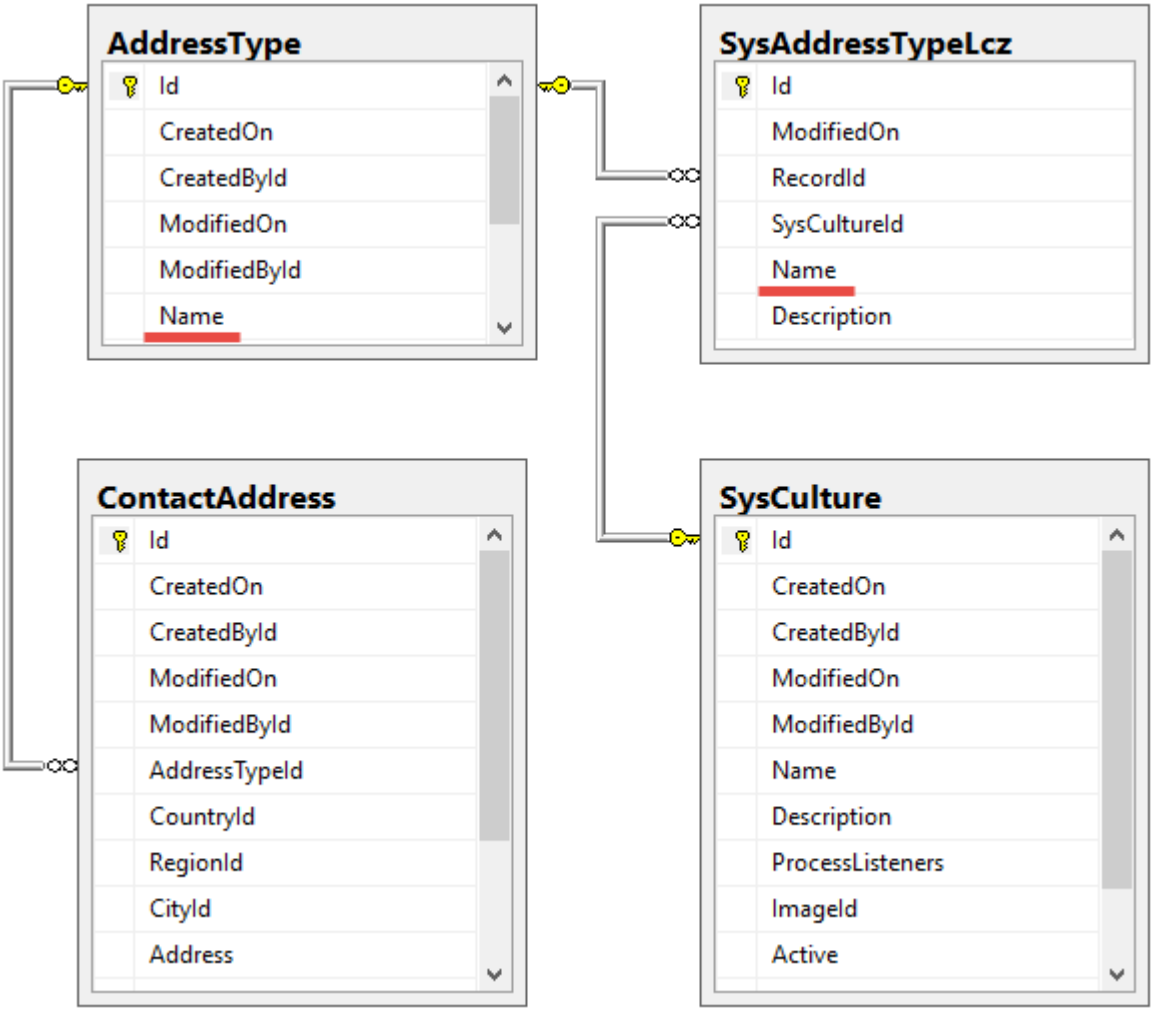
Case description

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:

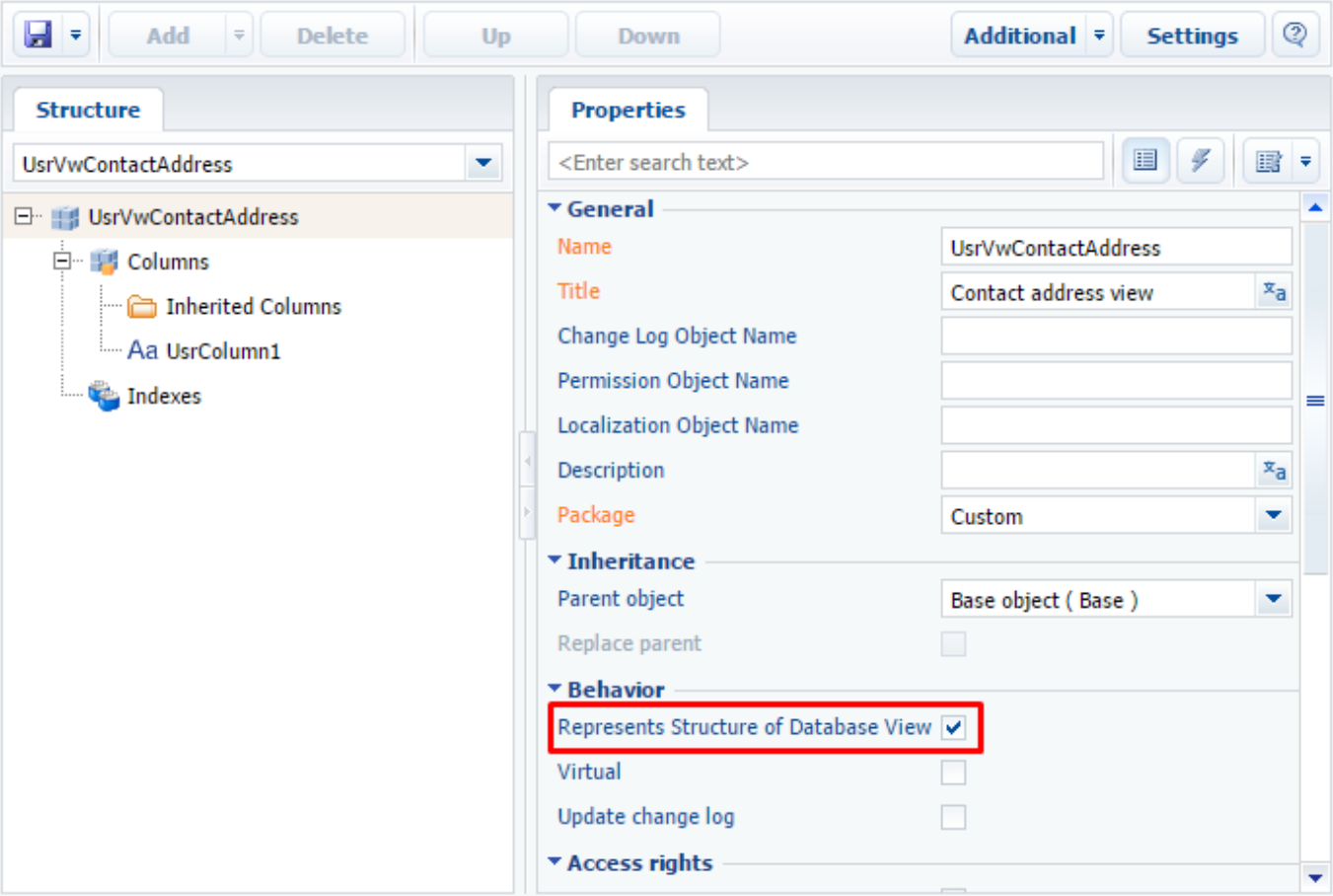
1. Create a view object schema

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:

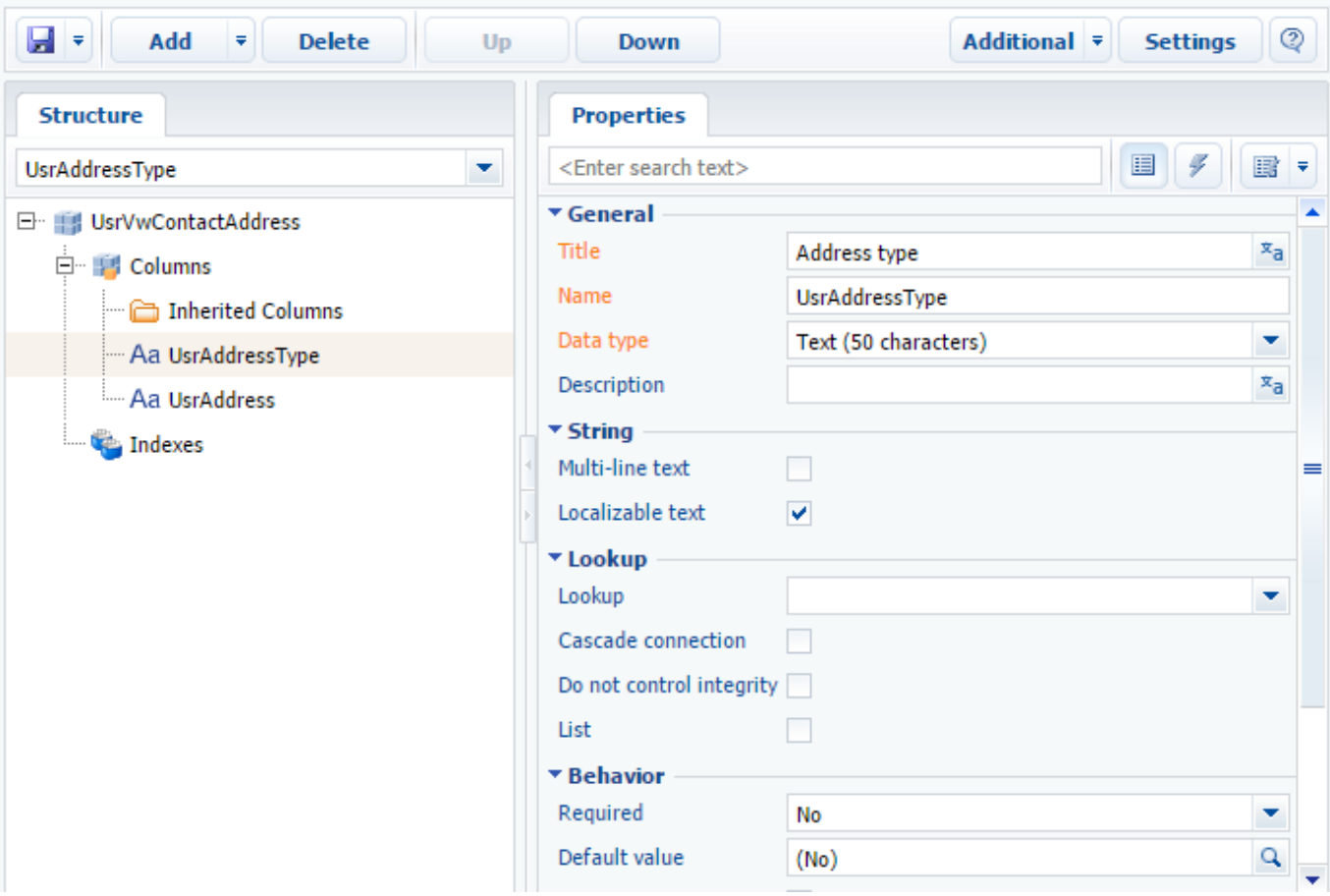
- [Name] – "UsrAdress". The *Usr* prefix corresponds to the value of the [Prefix for object name] system setting.
- [Title] – "Address".
- [Data type] – "Text (50 characters)". The string may be set to a different maximum length. The amount of memory used in the database depends on the number of characters.

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 own 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

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">
   123 6th St. Melbourne Delivery 71 Pilgrim Avenue Chevy Chase Work 514 S. Magnolia St. Delivery
 </string>
```

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

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">
  123 6th St. Melbourne Dirección de entrega 71 Pilgrim Avenue Chevy Chase Dirección de trabajo 514 S. Magnolia St. Dirección de entrega
 </string>
```

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 own 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;
```

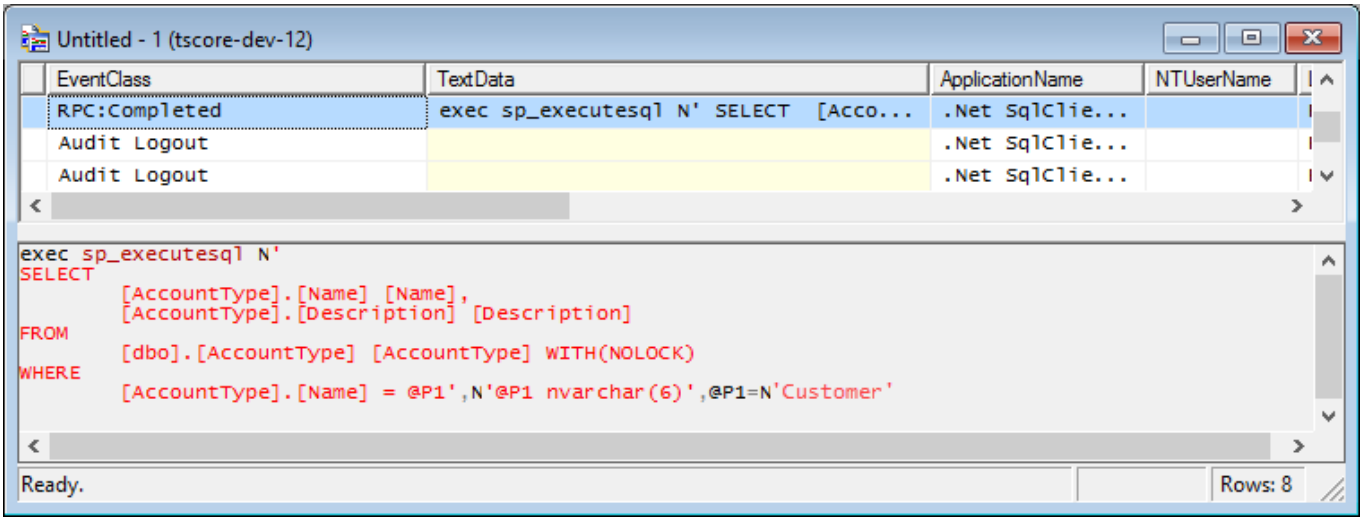
If a user who has a default language selected in the profile executes the method containing this code, the following query will be sent to the database:

```
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.

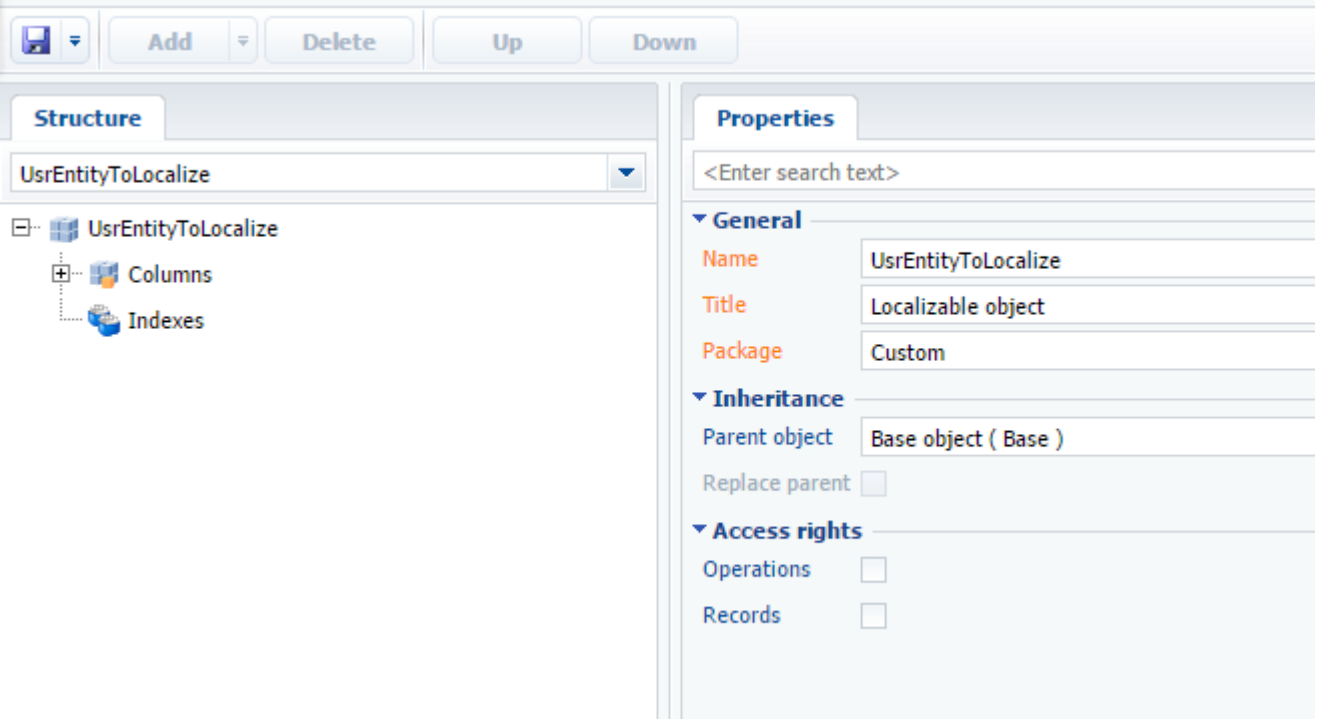
Case implementation algorithm

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

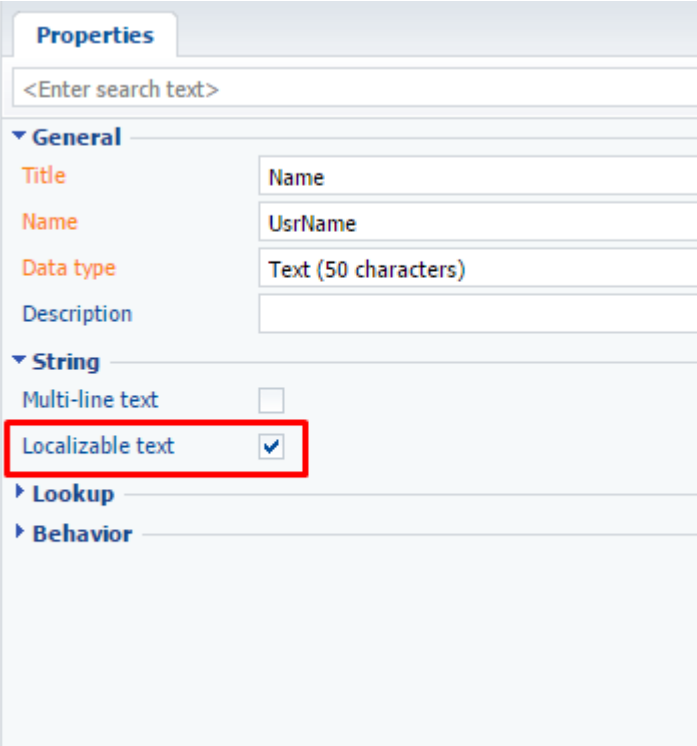
Add the column to the created schema with the following properties:

- [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()) {
            //...
        }
    }
}
```

Mobile application

- **Mobile application debugging**
- **How to add a custom section to a mobile application**

- **Working with editable lists in the mobile app**
- **Adding the [Attachments] detail to the section in mobile application**
- **Adding a custom widget to the mobile application**

Mobile application debugging

Introduction

During the process of developing custom solutions for the bpm'online mobile application, you need to repeatedly perform *application debug*.

Mobile application is not an **application of a hybrid type** (mobile web application in the native shell) and you can debug it via Google Chrome Developer tools in the Mobile device mode. More information about debugging the application code via browser development tools can be found in the "**Client code debugging**" article.

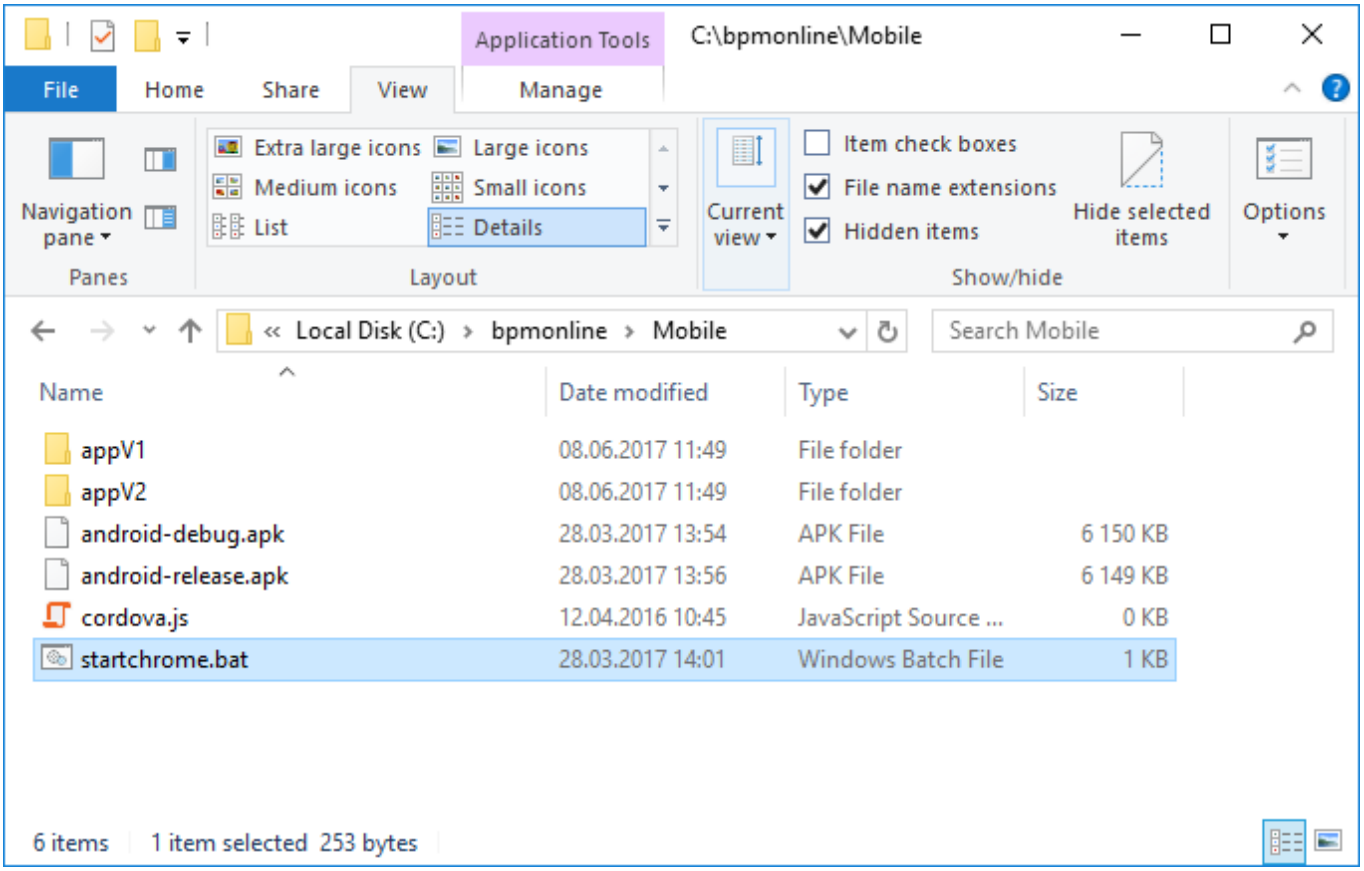
To launch a mobile application in the debug mode:

1. Get the files necessary for debugging the mobile application.
2. Launch the *startchrome.bat*.
3. Enter the debug mode for mobile devices in the Google Chrome.
4. Make the necessary settings and synchronize the mobile application bpm'online.

Getting the necessary files

Contact the support team to get the files for debugging a mobile application. Support team will provide an archive with corresponding files. Extract archive to any folder (for example, *C:\bpmonline\Mobile*) (Fig. 1).

Fig. 1. The contents of the unpacked archive



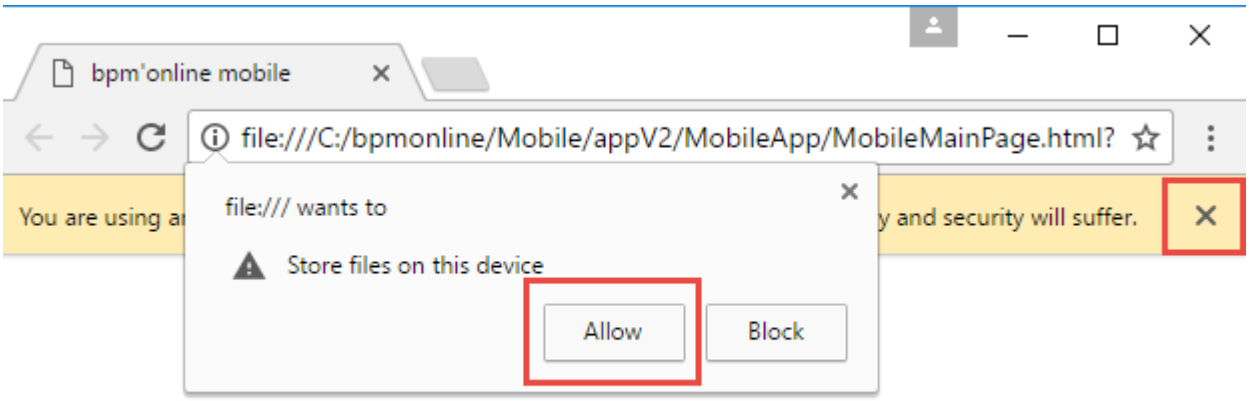
Launch of the startchrome.bat

> ⚠ ATTENTION
>
> Close Google Chrome application before you launch *startchrome.bat*.

The *startchrome.bat* is located in the root folder of unpacked archive. The Google Chrome will launch after executing the startchrome.bat.

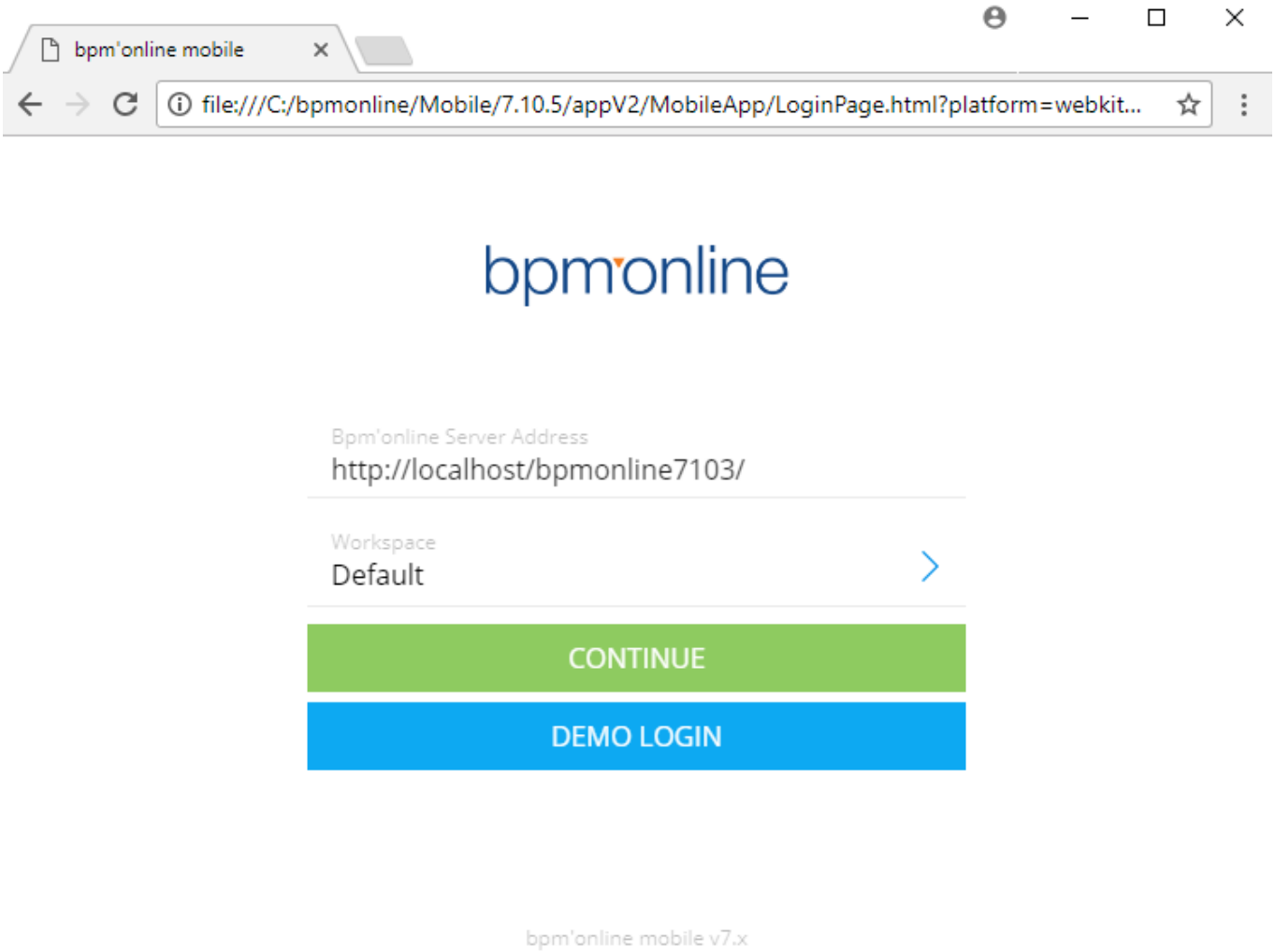
When you first start the browser with the startchrome.bat, an information window will appear, warning you about saving the files to the file system (Fig. 2). Allow the saving of the files. Close the warning about the *--disable-web-security* unsupported file (Fig. 2).

Fig. 2. Information window with warning



The execution of startchrome.bat will launch the Google Chrome with the settings page of bpm'online mobile application (Fig. 3).

Fig. 3. Mobile application settings page



Switching to mobile application debugging mode

To access the developer tools in Google Chrome, press *F12* key or *Ctrl* + *Shift* + *I* keys. You can debug the local version of the mobile application in the browser. More information about debugging application code via browser development tools can be found in the "**Client code debugging**" article.

> NOTE
>
> After switching to mobile device display mode, refresh the page by pressing F5 key.

Configuring and synchronization of the mobile application

At first login to the mobile application, you need to enter the HTTP address of the bpm'online application on the settings page. To do this, you need to start debug process and click the [Next] button (Fig. 5). After that, enter user name and password (Fig. 6).

Fig. 5. Settings page of the local mobile application

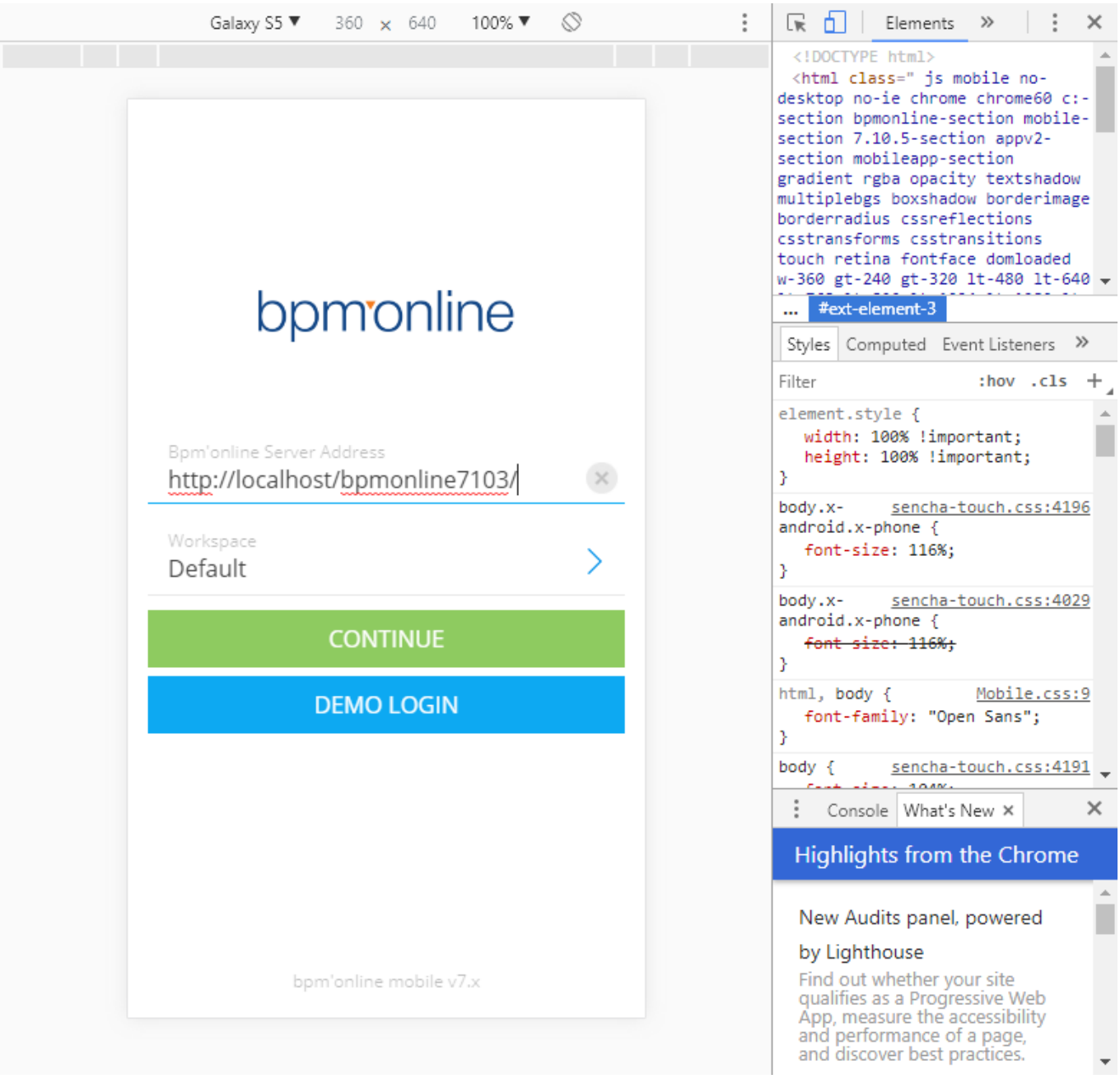
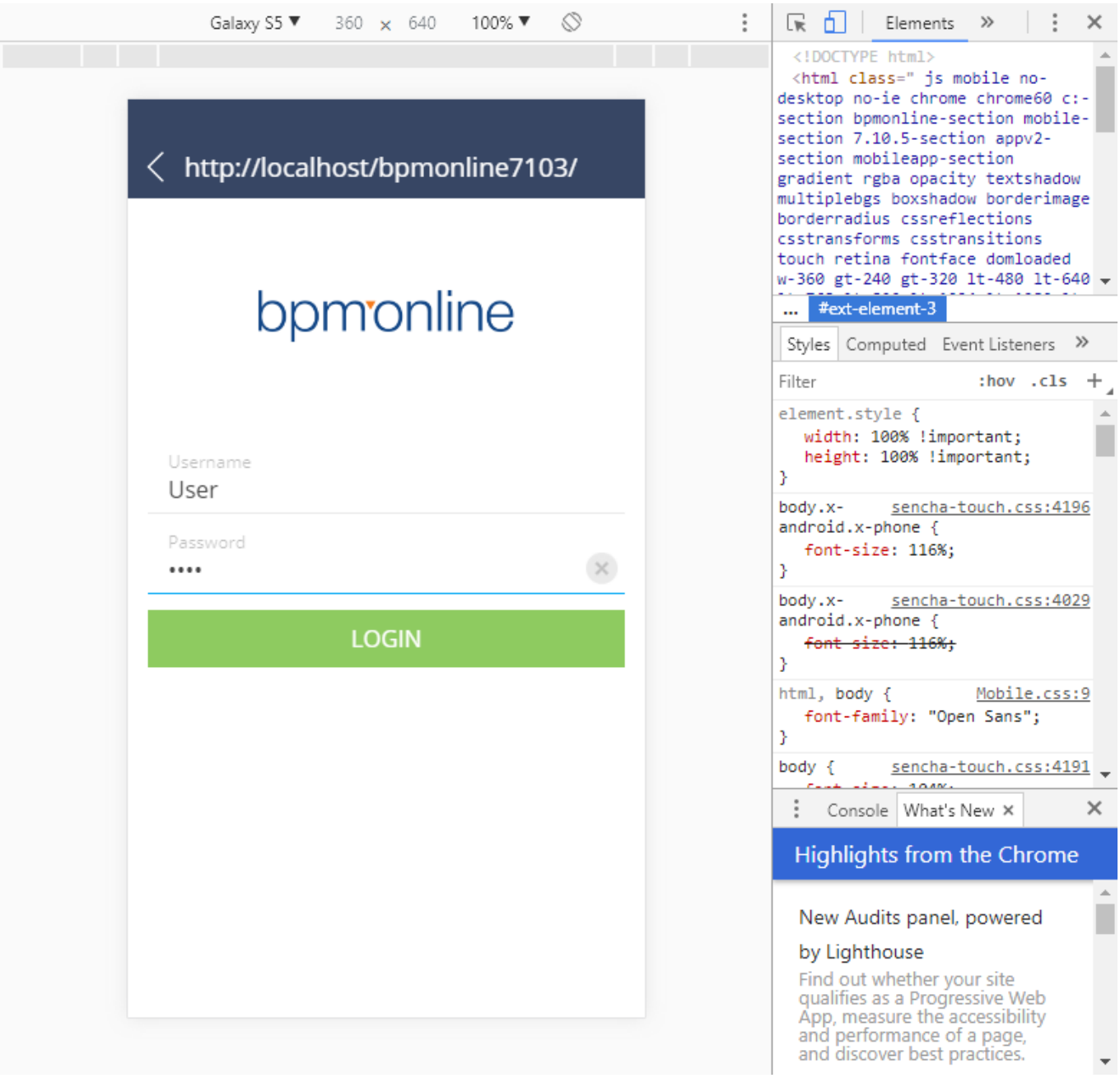


Fig. 6. Login page



After configuration and login, the local version of mobile application will behave as an application installed to the mobile device. The native functions of the mobile device (for example, working with the camera, downloading files, etc.) will not be supported. More information about working with mobile device in bpm'online can be found in the bpm'online mobile documentation.

How to add a custom section to a mobile application

Introduction

One of the most common development tasks is adding a custom section to the mobile bpm'online application. Use the mobile application wizard for this task. Using the wizard, not only can you add an existing section to the necessary workplace, but also pre–configure the display of the list, the edit page and the details of the added section.

The result of the mobile application wizard will be a set of client modules automatically created by the system added to the current user package:

- *The mobile application manifest schema* is a schema that extends the

MobileApplicationManifestDefaultWorkplace manifest of the *Mobile* package that describes the base version of the mobile application. This module is the main one and, in addition to other data, contains an array of names of the remaining modules in the *Models* section, or schema extensions of the model pages and action utilities. Here, you can also specify the main bpm'online application synchronization settings. For example, the content of schema model columns, which will be imported to the mobile application, and filters used to synchronize data. The name of the manifest scheme is formed using the *MobileApplicationManifest* mask. For example, for the [FieldForce] workplace, the system generates the *MobileApplicationManifestFieldForceWorkplace* manifest name, and for the [DefaultWorkplace] workplace, the name *MobileApplicationManifestDefaultWorkplace*.

NOTE

The model should be viewed as an object, e.g. the main section object.

- *List page extension schema* – implements additional settings for the base list page. Here, the user can select what will be displayed in the list of the added section of the mobile application. The name of this schema is generated by the *Mobile[Model name]GridPageSettings[Workplace name]* mask. For example, for an object named "*Book*" in the custom [Books] section of the [Field Sales] workplace, the system will generate the following list page extension schema name: *MobileBookGridPageSettingsFieldForceWorkplace*.

NOTE

Add the prefix specified in the [Prefix for object name] system setting to the name created by the wizard.

- *Edit page extension schema* – implements additional settings for the base edit page, e.g., which columns of the model can be edited in the mobile application. The name of this schema is generated by the *Mobile[Model name]RecordPageSettings[Workplace name]* mask. For example, for an object named "*Book*" in the custom [Books] section of the [Field Sales] workplace, the system will generate the following edit page extension schema name: *MobileBookRecordPageSettingsFieldForceWorkplace*.
- *Action extension schema* – implements the actions of the section. By default, the [Edit], [Copy] and [Delete] actions are implemented in the mobile application for the edit page, and only the [Add] action is added to the section list. The name of this schema is generated by the *Mobile[Model name]ActionsSettings[Workplace name]* mask. For example, for an object named "Book" in the custom [Books] section of the [Field Sales] workplace, the system will generate the following action extension schema name: *MobileBookActionsSettingsFieldForceWorkplace*.

If necessary, the user can add other modules to the package, in which the additional settings for the section or model (business rules, events, default values, etc.) will be implemented.

General sequence of adding a section to the mobile application

1. Create (if necessary) and configure the section in the main bpm'online application.
2. Add a custom section to the desired workplace of the mobile application using the mobile application wizard. Configure the display of the section, its edit pages and details.
3. Add advanced settings.

Case description

In the mobile application, add a custom [Claims] section with the following characteristics:

1. The schema of the claim object must contain (in addition to the columns inherited from the base object) the [Content] column of the (*String*) string type. In the [Content] column, the multi–line text must be stored (the contents of the claim).

2. Of all claims stored in the main application database, the mobile application must download only the claim created by the current user. In this case, you should consider the mode of the application – online or offline.

3. On the section list page of the mobile application, the user should see the claim topics and the claim creation dates. On the edit page – the title, the date of creation, and actual claim content. In this case, the creation date field must be locked for editing, and the content field must be multi–line.

NOTE

> We do not recommend downloading all section data to the mobile application, since mobile devices have limited memory resources. It is recommended to only synchronize the data related to the current user.

Case implementation algorithm

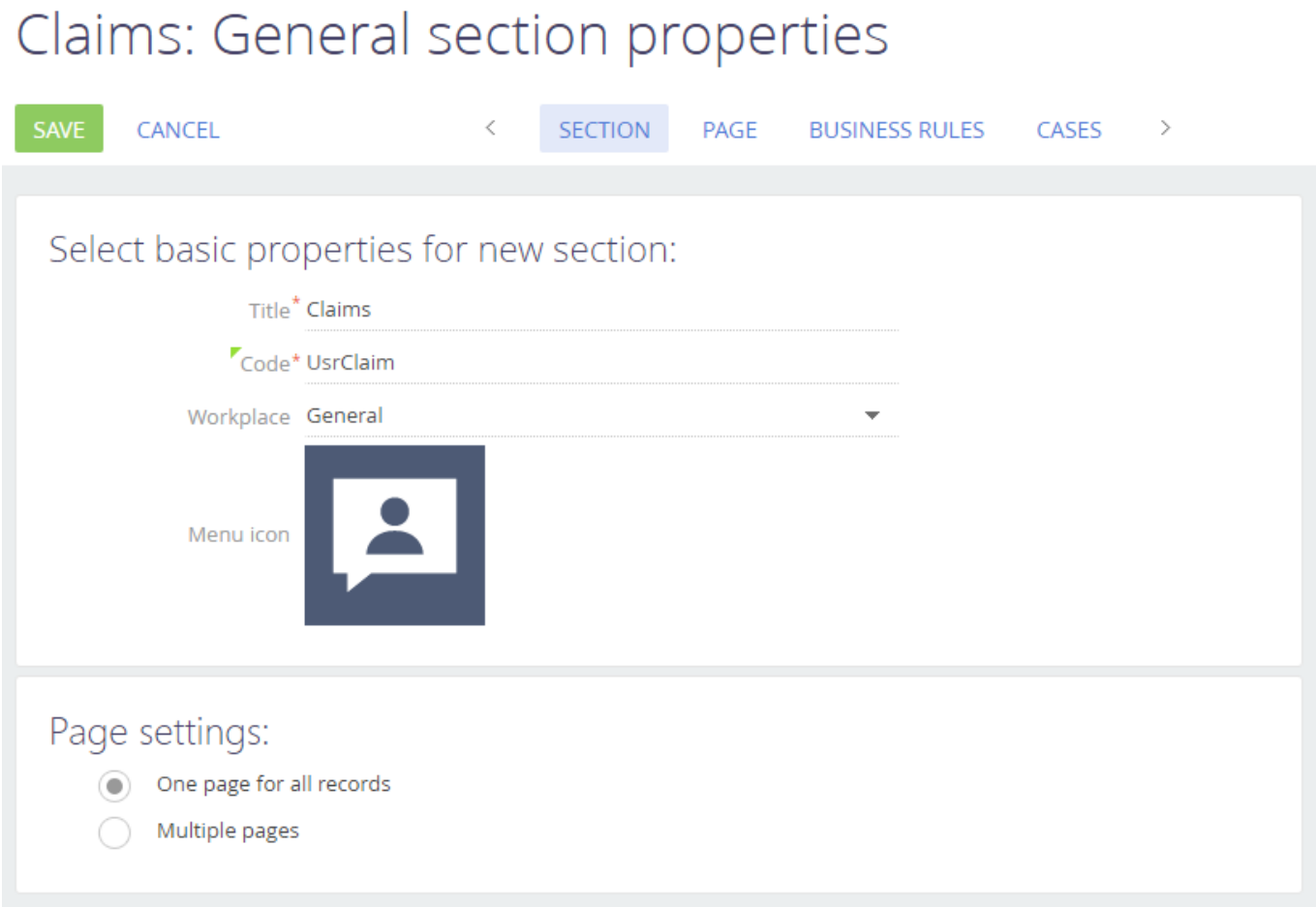
1. Create and configure a custom section in bpm'online

Use the Section Wizard to create a new custom section.

> ATTENTION
>
> If the created sections needs to be transferred to another application, do the following:
>
> - **create a custom package**
> - specify the name of the package in the [Current package] system setting
>
> Otherwise, the wizard will create schemas and data bindings in the **[Custom] package**, which will lead to problems with the transfer of the section.

In the first step ("Section") of the section wizard, in the [Code] field, enter "UsrClaim", and in the [Title] field, enter "Claims" (Fig. 1).

Fig. 1. Section properties setup



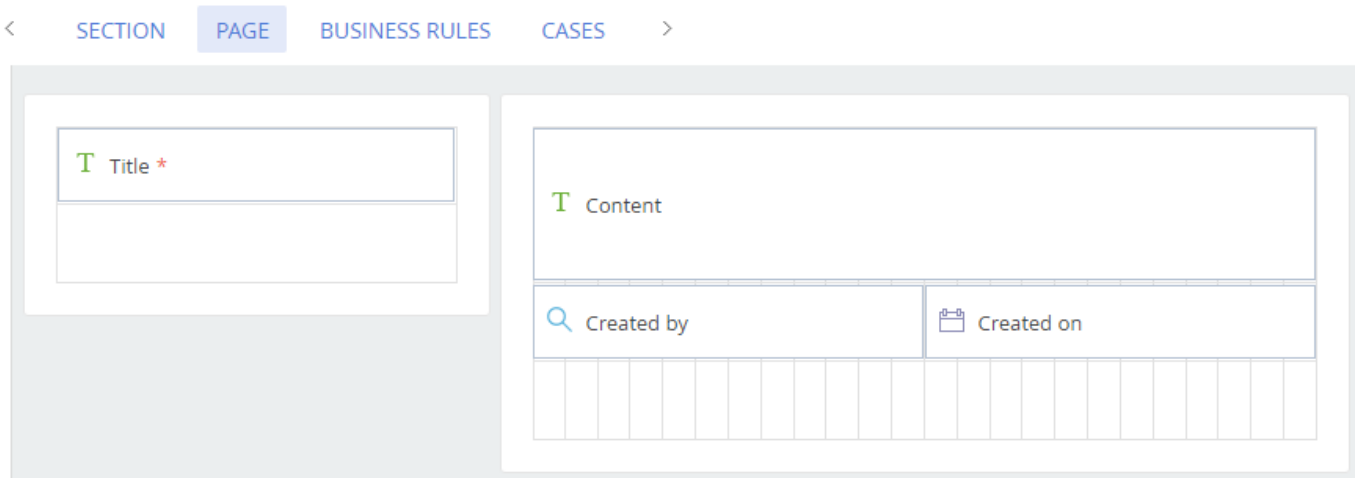
In the second step of the section wizard ("Page"), you need to rename the existing *UsrName* column and add a new one. Specify the following properties for the columns.

Title	Name in DB	Type
Title	*UsrName*	*String*

Content	*UsrContent*	*Multi-string text*

The placement of the columns on the page of the section in the wizard is shown in Fig. 2.

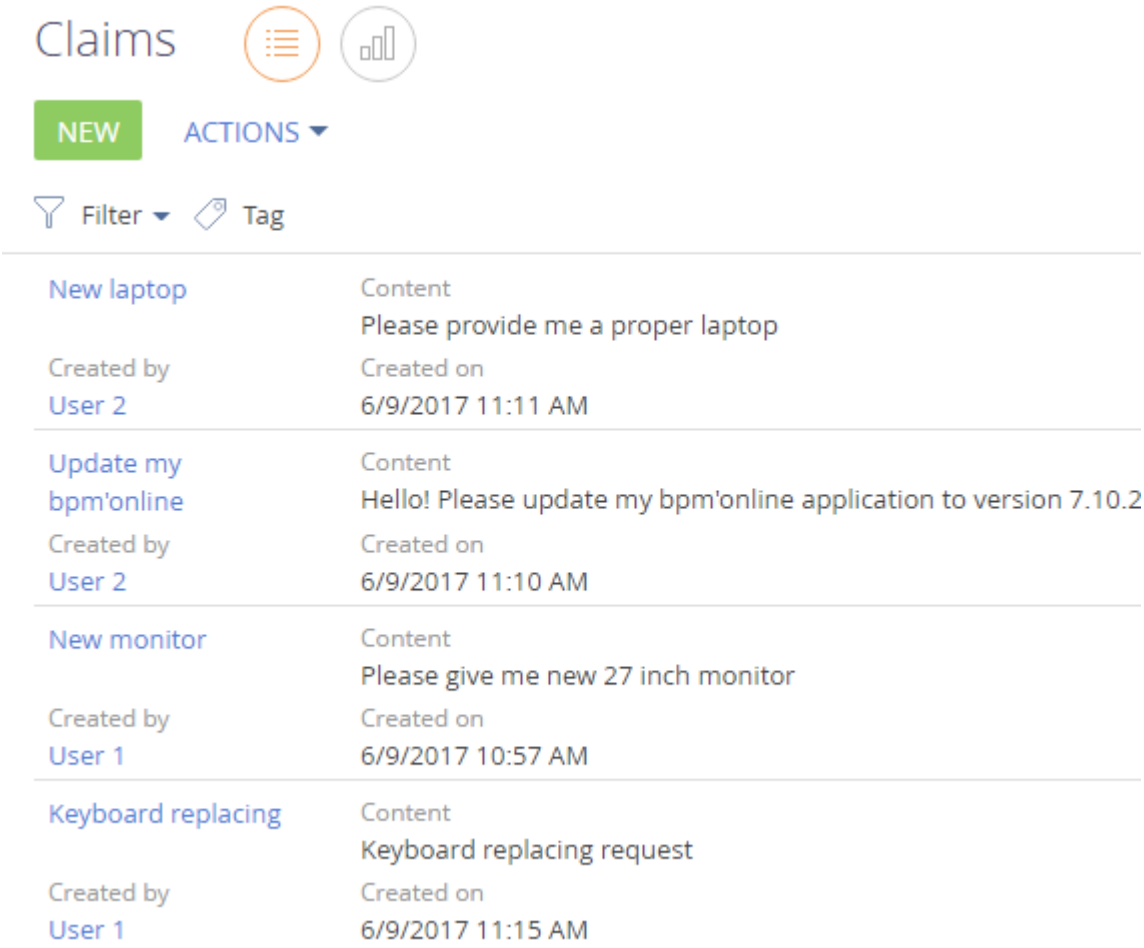
Fig. 2. The placement of the columns on the section page



After saving the configuration results, the system will automatically create all necessary diagrams of objects and pages in the current user package. Log in to the application again to make the section appear in the workplace.

To check the synchronization with the mobile application, you need to add several records on behalf of different users to the section (Fig. 3).

Fig. 3. Section records



2. Add a custom section to the desired workplace of the mobile application using the mobile application wizard

Use the mobile application wizard to add a custom [Claims] section to the mobile application. Open a workplace in the wizard (e.g. [Main workplace]).

Click [Set up sections] and then click [New] (Fig. 5).

Fig. 4. Configuring a workplace

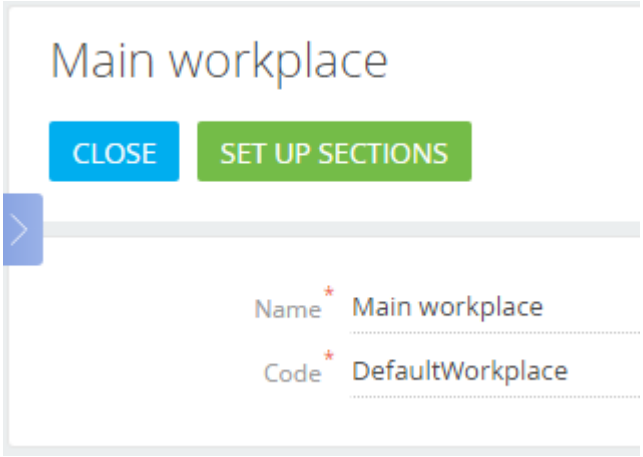
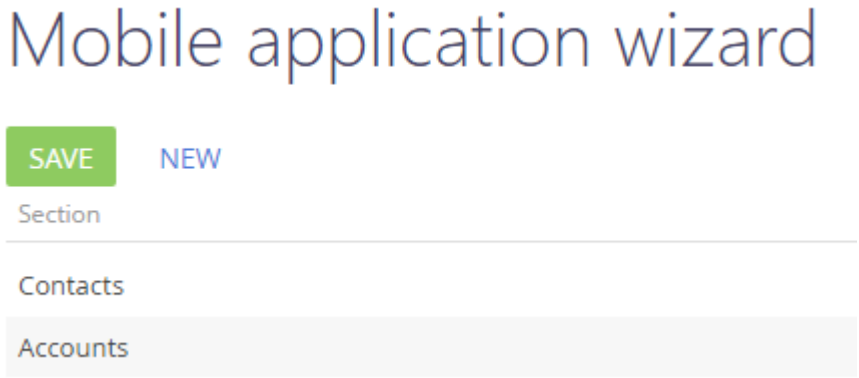
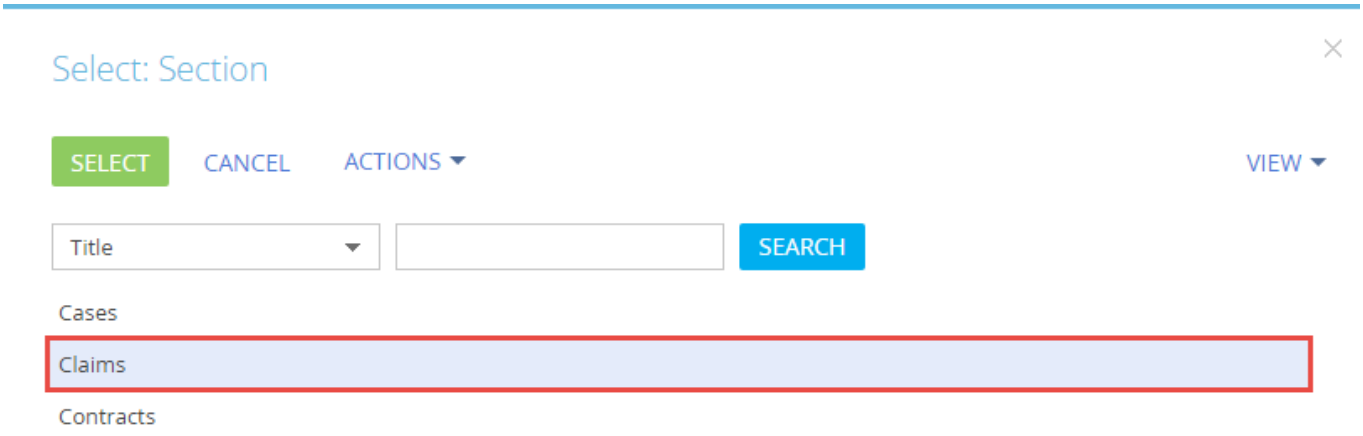


Fig. 5. Configuring sections



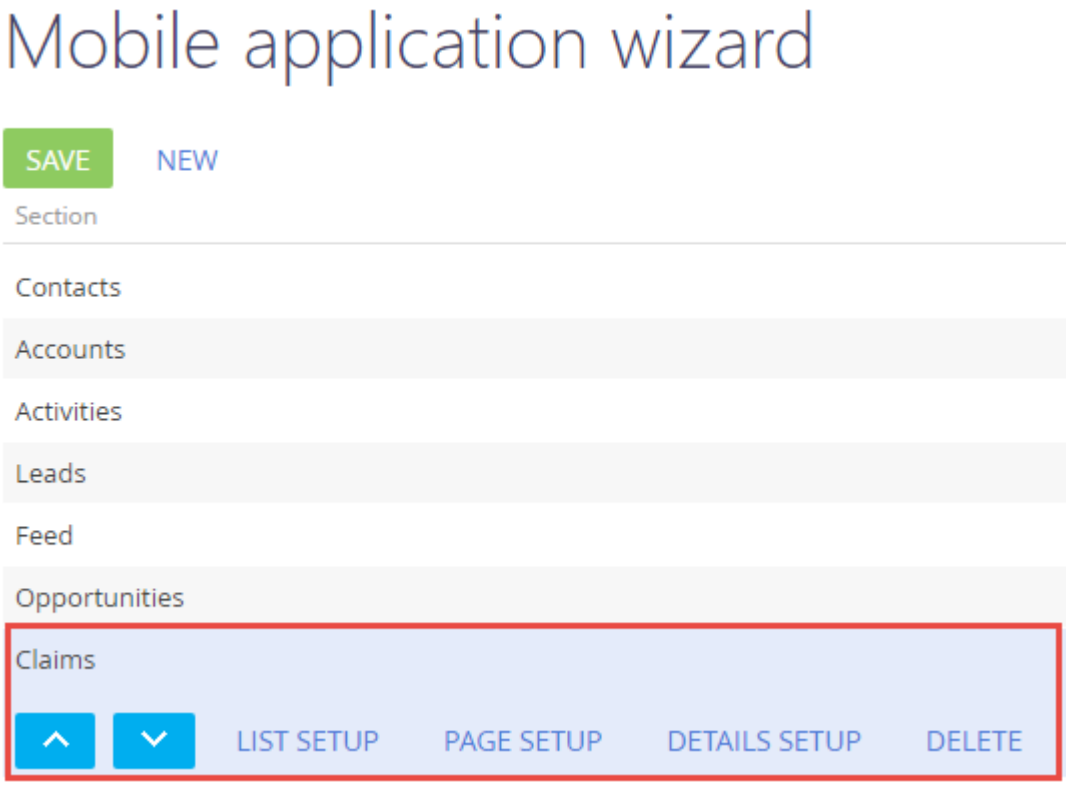
In the window that appears, select the [Claims] section from the list and then click the [Select] button (Fig. 5). Click the [Save] button (Fig. 6).

Fig. 6. Adding a section to a workplace



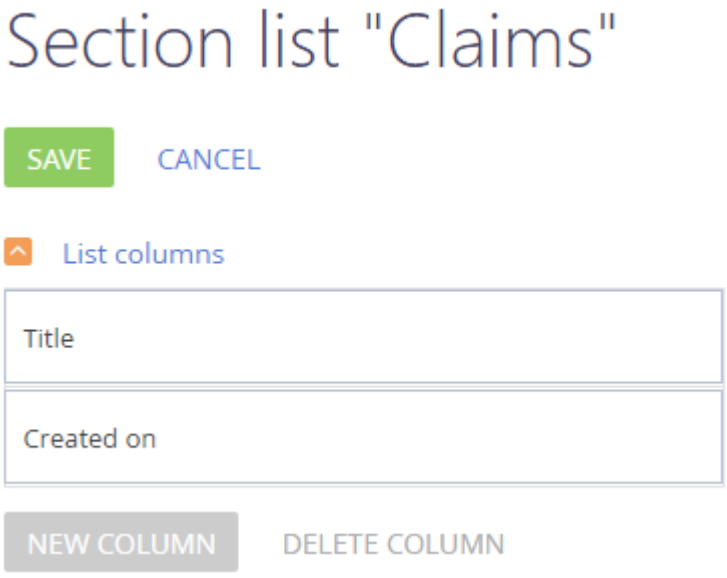
Configure the list for the selected [Claims] section. Access the list edit page by clicking [List setup].

Fig. 7. Actions in the selected record



By default, the only column displayed in the list is the [Title] column. Add the [Created on] column and save the changes.

Fig. 8. Configuring a list in the mobile application



> NOTE
>
> In the mobile application list (Fig. 8), you can add up to two columns using the wizard. By default, only two columns are displayed on the mobile application's list page - primary and secondary.

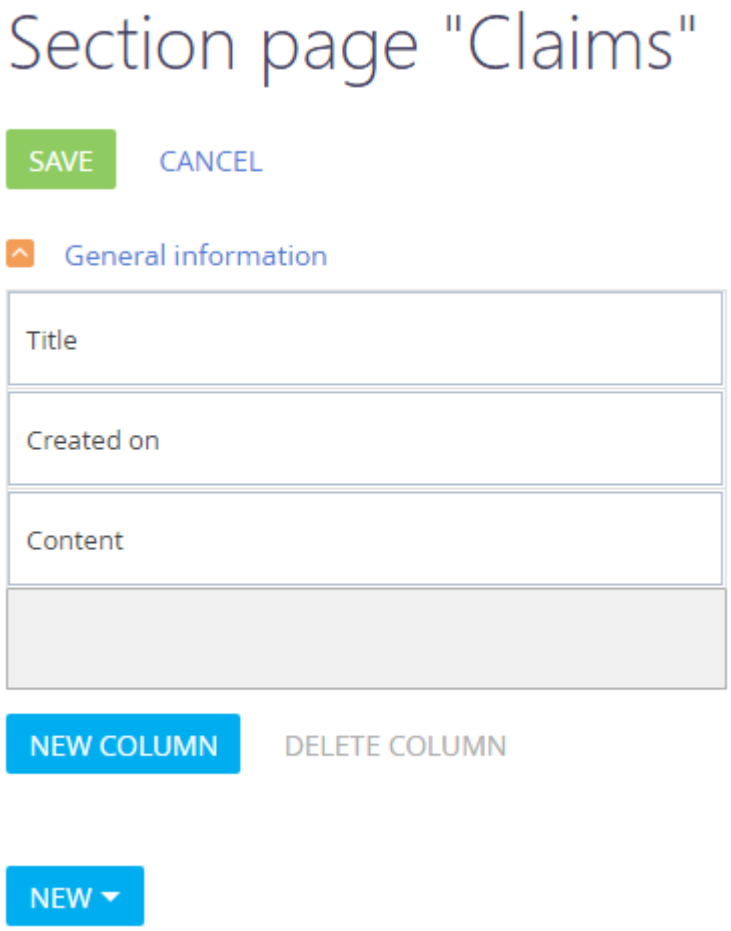
The final step of adding a custom section to the workplace of the mobile application is to set up an edit page for a claim in the [Claims] section. To set up integration:

1. After selecting the [Claims] section in the mobile application wizard, click [Page Setup] (Fig. 7).

2. Add two columns – [Created on] and [Contents] (Fig. 9).

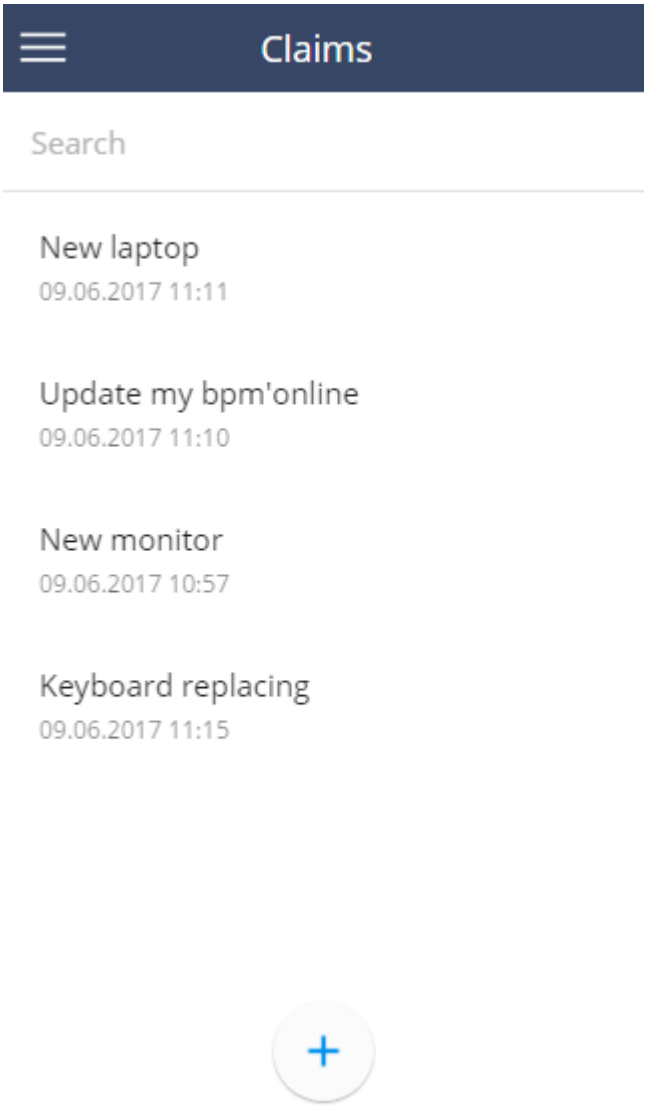
3. Save the changes.

Fig. 9. Configuring a claim edit page



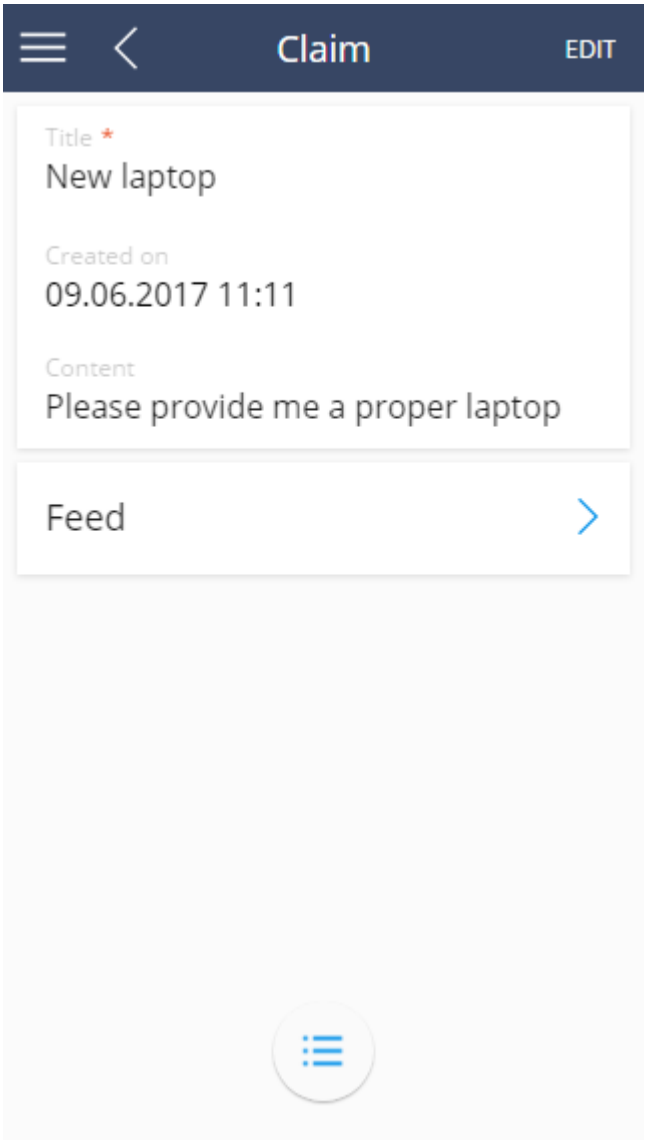
As a result, the new [Claims] section will be displayed in the mobile application.

Fig. 10. The [Claims] section in the mobile application



At this step, the [Claims] section is fully functional in the mobile application. However, the application loads all data from the main bpm’online application, regardless of the mode it is used in – online or offline. Additionally, on the claim edit page, the content field is only one-line, and the [Created on] field allows you to change the existing creation date (Fig. 11).

Fig. 11. The claim edit page in the mobile application



3. Filtering the synchronized data for the offline mode

> NOTE
>
> To switch the mobile application to offline or online mode, specify the necessary application in the [Mobile application operation mode] system setting of the main bpm'online application. The default mode for a new user is online.

In order to only download data related to the current user during offline synchronization, it is necessary to make the following changes to the manifest schema:

1. In the array of the *ModelDataImportConfig* property, change the configuration object for importing data related to the *UsrClaim* object. Add the corresponding configuration object to the *SyncFilter* property and add the *CreatedBy* column to the *SyncColumns* property array.

```
{
    "SyncOptions": {
        ...
        "ModelDataImportConfig": [
            {
                "Name": "UsrClaim",
                "SyncFilter": {
                    "property": "CreatedBy",
```

```
                    "valueIsMacros": true,
                    "value": "Terrasoft.ValueMacros.CurrentUserContact"
                },
                "SyncColumns": [
                    "UsrName",
                    "CreatedOn",
                    "UsrContent",
                    "CreatedBy"
                ]
            },
            ...
        ]
    },
    ...
}
```

2. In the array of the *ModelDataImportConfig* property, change the configuration object for importing data related to the *Contact* object. Because the *Id* column is sufficient for data filtering the *SyncColumns* property array should be left blank.

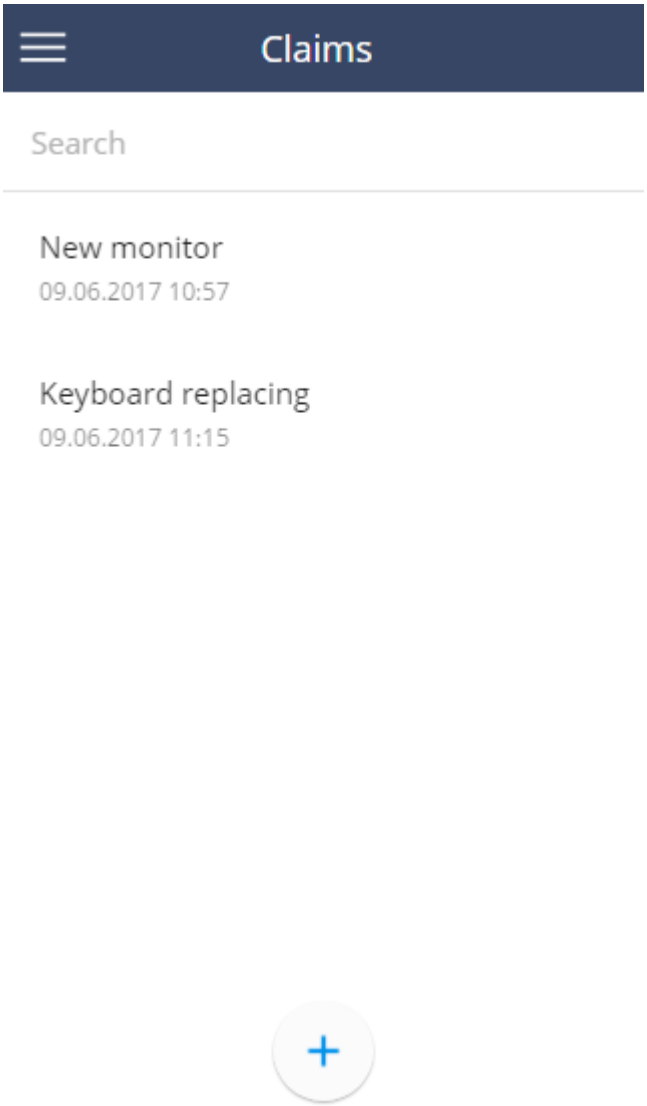
```
{
    "SyncOptions": {
        ...
        "ModelDataImportConfig": [
            ...
            {
                "Name": "Contact",
                "SyncColumns": []
            }
        ]
    },
    ...
}
```

3. In the *RequiredModels* array of the configuration object of the *UsrClaim* model, add the *Contact* model (required for filtering). It is required because the *CreatedBy* column of the *UsrClaim* object contains the contact's identifier.

```
{
    ...
    "Models": {
        "UsrClaim": {
            "RequiredModels": [
                "UsrClaim",
                "SocialMessage",
                "Contact"
            ],
        ...
    }
}
```

As a result, while the mobile application is offline, only the claims associated with the current user will be displayed on the section list (Fig. 12).

Fig. 12. User1 user claims in offline mode



4. Filtering the synchronized data for the online mode

In order to only display the records associated with the current user in the online mode, do the following:

1. Create a new module for the additional settings of the [Claims] section. It is recommended to name the module using a *[Object Name Prefix]Mobile [Unit Object Name]ModuleConfig[Workplace Name]* mask. For the [Claims] section, the recommended module name is *UsrMobileUsrClaimModuleConfigDefaultWorkplace*. The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**" article.

2. Add the following code to the created module:

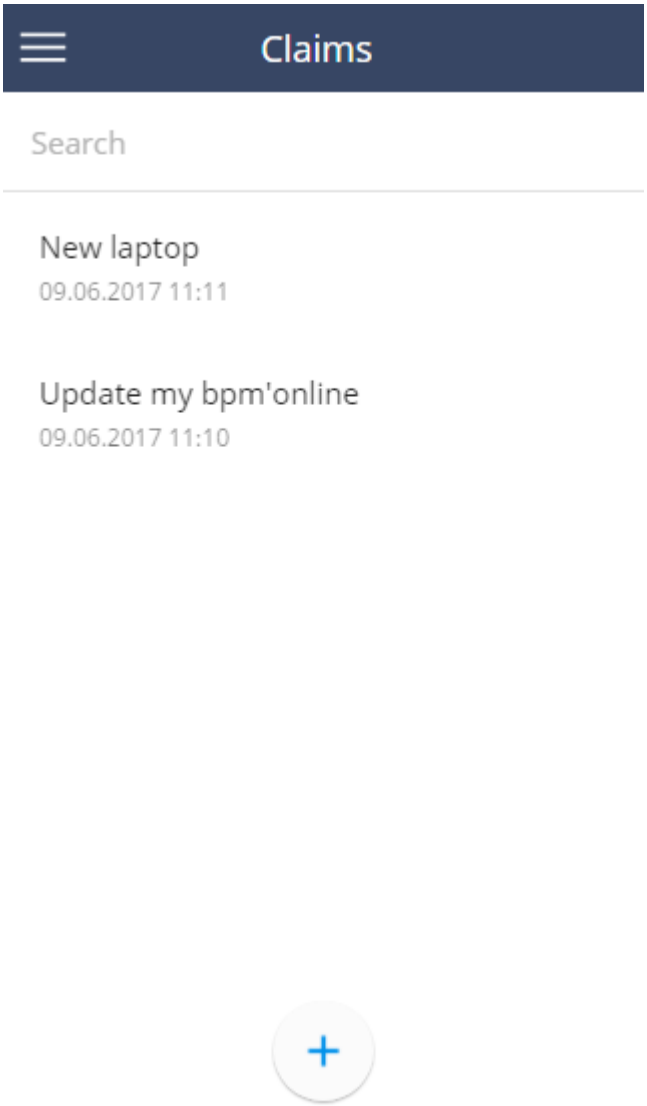
```
// Adding a filter for the UsrClaim model.
Terrasoft.sdk.Module.addFilter("UsrClaim", Ext.create("Terrasoft.Filter", {
    // Filter name.
    name: "UsrClaimCustomFilter",
    // The  model property used to perform the record comparison operation.
    property: "CreatedBy",
    // Is the comparison value a macro?
    valueIsMacros: true,
    // Value for comparison with the model property.
    value: "Terrasoft.ValueMacros.CurrentUserContact"
}));
```

3. Register the module in the manifest schema by adding its name to the *PagesExtensions* array:

```
{
    ...
    "Models": {
        "UsrClaim": {
            ...
            "PagesExtensions": [
                "UsrMobileUsrClaimActionsSettingsDefaultWorkplace",
                "UsrMobileUsrClaimGridPageSettingsDefaultWorkplace",
                "UsrMobileUsrClaimRecordPageSettingsDefaultWorkplace",
                "UsrMobileUsrClaimModuleConfigDefaultWorkplace"
            ]
        },
        ...
    }
}
```

As a result, while the mobile application is online, only the claims associated with the current user will be displayed on the section list (Fig. 13).

Fig. 13. User1 user claims in online mode



5. Add advanced settings

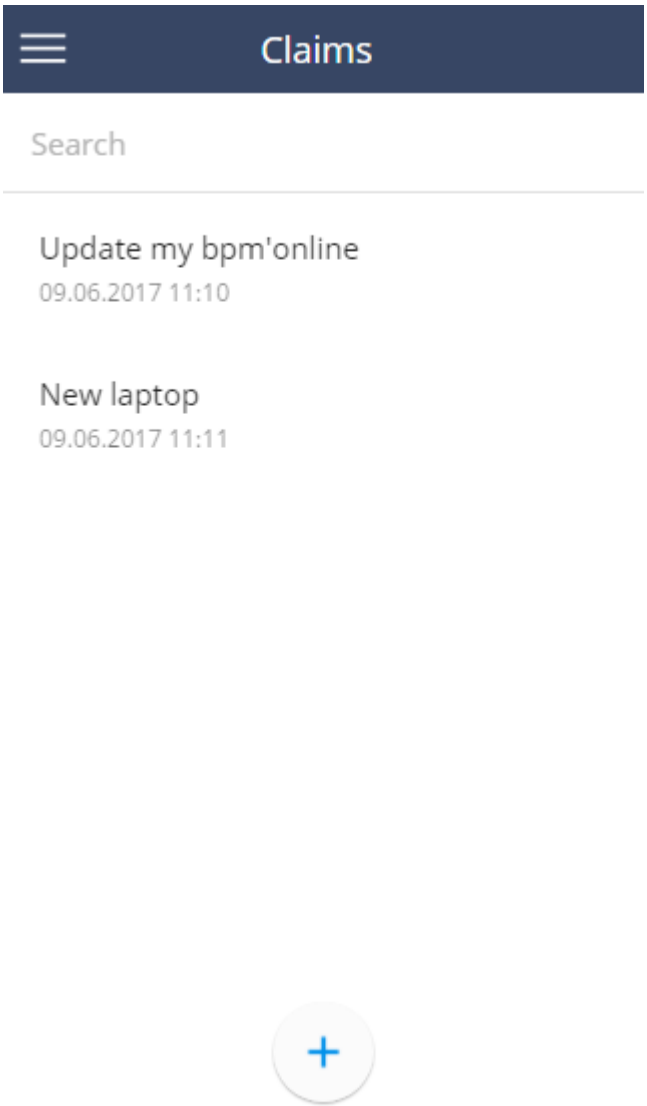
5.1 Add sorting in the section list

To sort the records of the custom section by their creation date, you need to add the following source code to the *UsrMobileUsrClaimModuleConfigDefaultWorkplace* module, created in the previous step:

```
...
// Sorting a section list.
Terrasoft.sdk.GridPage.setOrderByColumns("UsrClaim", {
    // Sorting by creation date column.
    column: "CreatedOn",
    // Ascending order.
    orderType: Terrasoft.OrderTypes.ASC
});
```

As a result, the records will be displayed in ascending order and sorted by the creation date (Fig. 14)

Fig. 14. Displaying User2 claims in ascending order and sorting by creation date



5.2 Change the edit page field to a multi-line field

On the edit page of the claim, make the [Content] input field multi-line by adding the following source code to the *UsrMobileUsrClaimModuleConfigDefaultWorkplace* module, created in step 4:

```
// Set properties for the [Content] column on the order edit page.
Terrasoft.sdk.RecordPage.configureColumn("UsrClaim", "primaryColumnSet",
"UsrContent", {
```

```
    // Specify if the field is multi-line.
    isMultiline: true
});
```

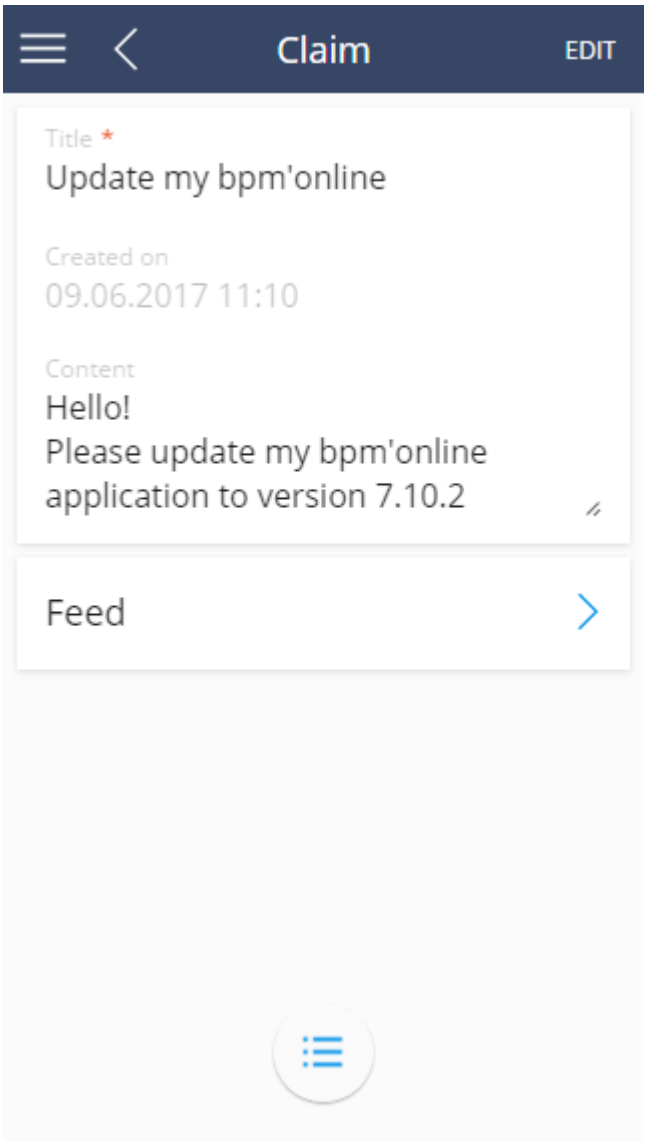
5.3 Make the field read-only

On the edit page of the claim, make the [Content] input field read-only by adding the following source code to the *UsrMobileUsrClaimModuleConfigDefaultWorkplace* module, created in step 4:

```
// Set the properties for the [Date of creation] column on the order edit page.
Terrasoft.sdk.RecordPage.configureColumn("UsrClaim", "primaryColumnSet", "CreatedOn",
{
    // Specify if the field is read-only.
    readOnly: true
});
```

As a result, the ability to change the creation date of a claim will be blocked, and the [Content] field will allow you to enter and display multi-line text (Fig. 15).

Fig. 15. A modified claim edit page



Working with editable lists in the mobile app

Introduction

Starting with version 7.9.0., bpm'online mobile application can work with editable lists. A new *Terrasoft.controls.EditList* component was created to implement this function. This component displays the edit columns, that enables editing the records in the list without opening the edit page.

To use the editable list in the section or in the detail:

1. Create the extending schema of the list page and use the *Terrasoft.view.BaseEditGridPage* view class and *Terrasoft.controller.BaseEditGridPage* controller class as parent classes.

2. In the view of the *grid* property, specify the set of editable columns using the *editColumns* property.

Use case

The example of the view and controller implementation code:

```
// Downloaded data for the page.
Terrasoft.LastLoadedPageData = {
    controllerName: "ActivitySKUGridPage.Controller",
    viewXType: "activityskugridpageview"
};

// The data store that the controller displays in the view.
Ext.define("ActivitySKUGridPage.Store", {
    extend: "Terrasoft.store.BaseStore",
    config: {
        model: "ActivitySKU",
        controller: "ActivitySKUGridPage.Controller"
    }
});

// View
Ext.define("ActivitySKUGridPage.View", {
    extend: "Terrasoft.view.BaseEditGridPage",
    xtype: "activityskugridpageview",
    config: {
        id: "ActivitySKUGridPage",
        grid: {
            editColumns: [
                {
                    name: "Balance"
                },
                {
                    name: "Presence"
                }
            ],
            store: "ActivitySKUGridPage.Store"
        }
    }
});

// View model controller
Ext.define("ActivitySKUGridPage.Controller", {
    extend: "Terrasoft.controller.BaseEditGridPage",
    statics: {
        Model: ActivitySKU
    },
    config: {
        refs: {
            view: "#ActivitySKUGridPage"
        }
```

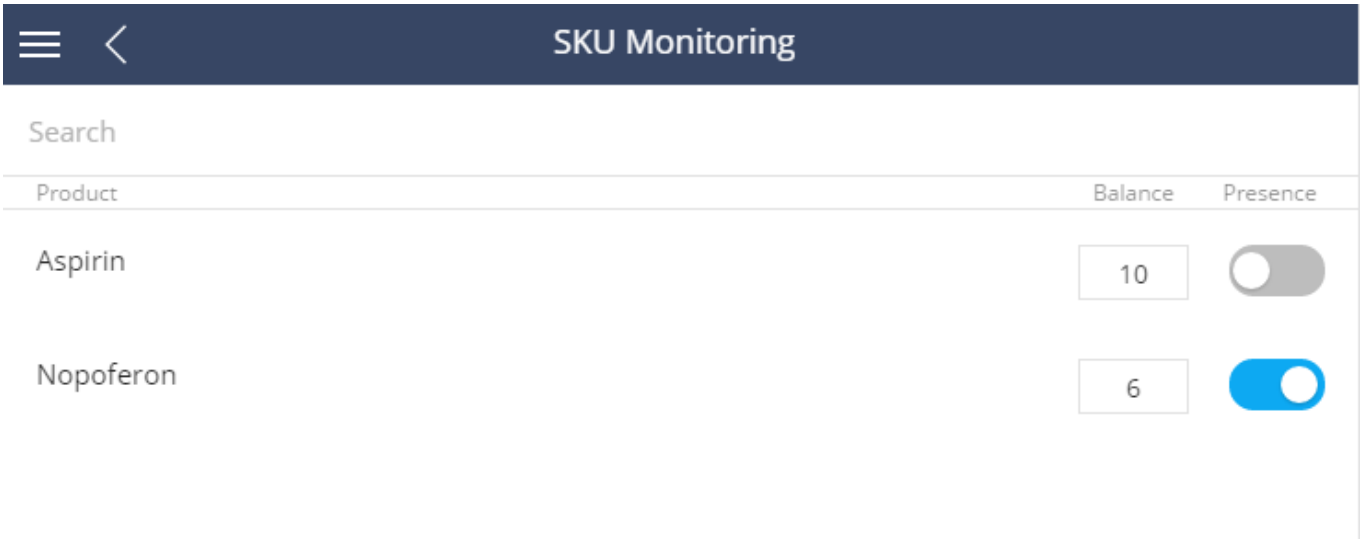
```
    }
});
```

In the configuration schema of the corresponding section or detail (usually it is a schema with the *ModuleConfig* in the name ending; see more details in the "**How to add a custom section to a mobile application**" article) the same editable columns that will be in the data fetch must be specified for the list. To do this, use the *addColumns()* method of the *Terrasoft.sdk.GridPage* class:

```
Terrasoft.sdk.GridPage.addColumns(skuModelName, ["Balance", "Presence"]);
```

As a result, the detail will look like the one shown in Fig. 1.

Fig. 1 Editable list of the [SKU Monitoring] detail



Adding the [Attachments] detail to the section in mobile application

Introduction

The [Attachments] detail is used to store files and web-links connected to corresponding section record. Detail is available in all bpm'online sections (see "How to work with attachments and notes") and in the [Opportunities] section of mobile application.

The [Attachments] detail is implemented in the mobile application via embedded detail. The embedded detail is used to facilitate the data edit process without opening the separate page.

Steps to add a detail to the record page in the section of mobile application:

1. Create section schema and define an embedded detail based on the object for the [Attachments] detail.

2. Create schemas of the view and edit pages of the section record in the mobile application.

3. Add information about new schemas to the mobile application manifest.

Case description

Add the [Attachments] detail on the edit page of the [Contacts] section of mobile application.

Case implementation algorithm

1. Create section schema and define an embedded detail there

Create a new module schema in the [Configuration] section of a custom package with the following property values:

- [Title] – "UsrMobileContactModuleConfig".
- [Name] – "UsrMobileContactModuleConfig".
- [Package] – the name of the package used for development.

Add localized strings to the created schema (Fig. 1) with properties given in the table 1.

Fig. 1. Adding of the localizable string in the module schema designer

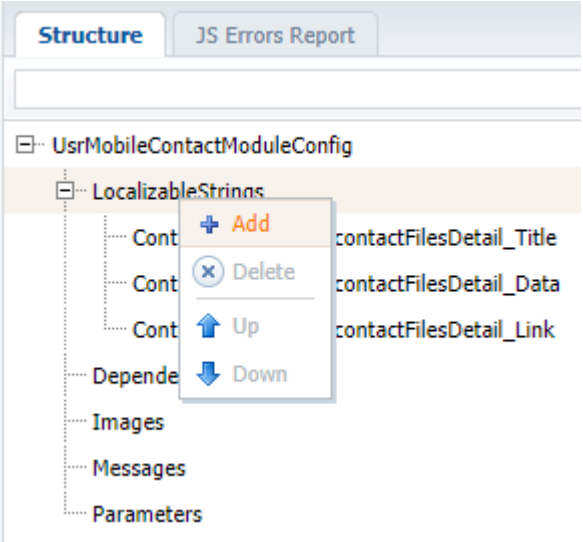


Table 1. Localizable string properties

Localizable string	Value	Description
ContactRecordPage_contactFilesDetail_Title	Attachments	Detail title.
ContactRecordPage_contactFilesDetail_Data	File	Field title for the file.
ContactRecordPage_contactFilesDetail_Link	Link	Field title for the web-link.

Add following source code on the [Source Code] tab of the schema in the module designer:

```
// Adding an embedded detail.
Terrasoft.sdk.RecordPage.addEmbeddedDetail("Contact", {
    name: "contactFilesDetail",
    position: 5,
    title: "ContactRecordPage_contactFilesDetail_Title",
    modelName: "ContactFile",
    primaryKey: "Id",
    foreignKey: "Contact",
    displaySeparator: false
},
// An array of configuration objects of the detail columns.
[
    {
        name: "Data",
        displayColumn: "Name",
        label: "ContactRecordPage_contactFilesDetail_Data",
        placeHolder: "ContactRecordPage_contactFilesDetail_Data"
    },
    {
        name: "Type",
        hidden: true
    },
    {
        name: "Name",
        label: "ContactRecordPage_contactFilesDetail_Link",
```

```
        placeHolder: "ContactRecordPage_contactFilesDetail_Link",
        viewType: Terrasoft.ViewTypes.Url
    }
]);
```

Save the schema to apply changes.

The process of calling the *addEmbeddedDetail()* method occurs here and the following values are passed as arguments:

- Name of the main section object to which the detail will be added.
- Configuration object of the detail properties.
- Array of configuration objects of the properties of detail columns.

2. Create schemas of the view and edit pages of the record

Page of the section record can be displayed in the mobile applications in two modes: in the view mode and in the edit mode. To display the detail correctly in both modes you need to add corresponding controllers and views.

To display the detail correctly in the view mode add the new module schema to the custom package with following values of the properties:

- [Title] – "UsrMobileContactPreviewPage".
- [Name] – "UsrMobileContactPreviewPage".
- [Package] – the name of the package used for development.

Add following source code on the [Source Code] tab of the schema in the module designer:

```
// Storing of the viewed pages and sections.
Terrasoft.LastLoadedPageData = {
    controllerName: "ContactPreviewPage.Controller",
    viewXType: "contactpreviewpageview"
};
// Detail view in the view mode.
Ext.define("ContactPreviewPage.View", {
    extend: "Terrasoft.view.BasePreviewPage",
    xtype: "contactpreviewpageview",
    config: {
        id: "ContactPreviewPage"
    }
});
// Detail controller in the view mode.
Ext.define("ContactPreviewPage.Controller", {
    extend: "FileAndLinksPreviewPage.Controller",
    statics: {
        Model: Contact
    },
    config: {
        refs: {
            view: "#ContactPreviewPage"
        }
    },
    fileModel: "ContactFile"
});
```

Save the schema to apply changes.

To display the detail correctly in the edit mode add the new module schema to the custom package with following values of the properties:

- [Title] – "UsrMobileContactEditPage".
- [Name] – "UsrMobileContactEditPage".
- [Package] – the name of the package used for development.

Add following source code on the [Source Code] tab of the schema in the module designer:

```
// Storing of the viewed pages and sections.
Terrasoft.LastLoadedPageData = {
    controllerName: "ContactEditPage.Controller",
    viewXType: "contacteditpageview"
};
// Detail view in the edit mode.
Ext.define("ContactEditPage.View", {
    extend: "Terrasoft.view.BaseEditPage",
    xtype: "contacteditpageview",
    config: {
        id: "ContactEditPage"
    }
});
// Detail controller in the edit mode.
Ext.define("ContactEditPage.Controller", {
    extend: "FileAndLinksEditPage.Controller",
    statics: {
        Model: Contact
    },
    config: {
        refs: {
            view: "#ContactEditPage"
        }
    },

    fileModel: "ContactFile"
}, function() {
    this.addDefaultBusinessRules();
});
```

Save the schema to apply changes.

3. Add information about new schemas to the mobile application manifest

The information about new schemas should be added to the mobile application manifest to apply the changes. For this, create the *MobileApplicationManifestDefaultWorkplace* extending schema in the custom package. Learn more about application manifest in the "**Mobile application manifest**" article. The procedure for creating a replacing client schema is covered in the "**Creating a custom client module schema**" article.

Add following source code on the [Source Code] tab of the *MobileApplicationManifestDefaultWorkplace* extending schema in the module designer:

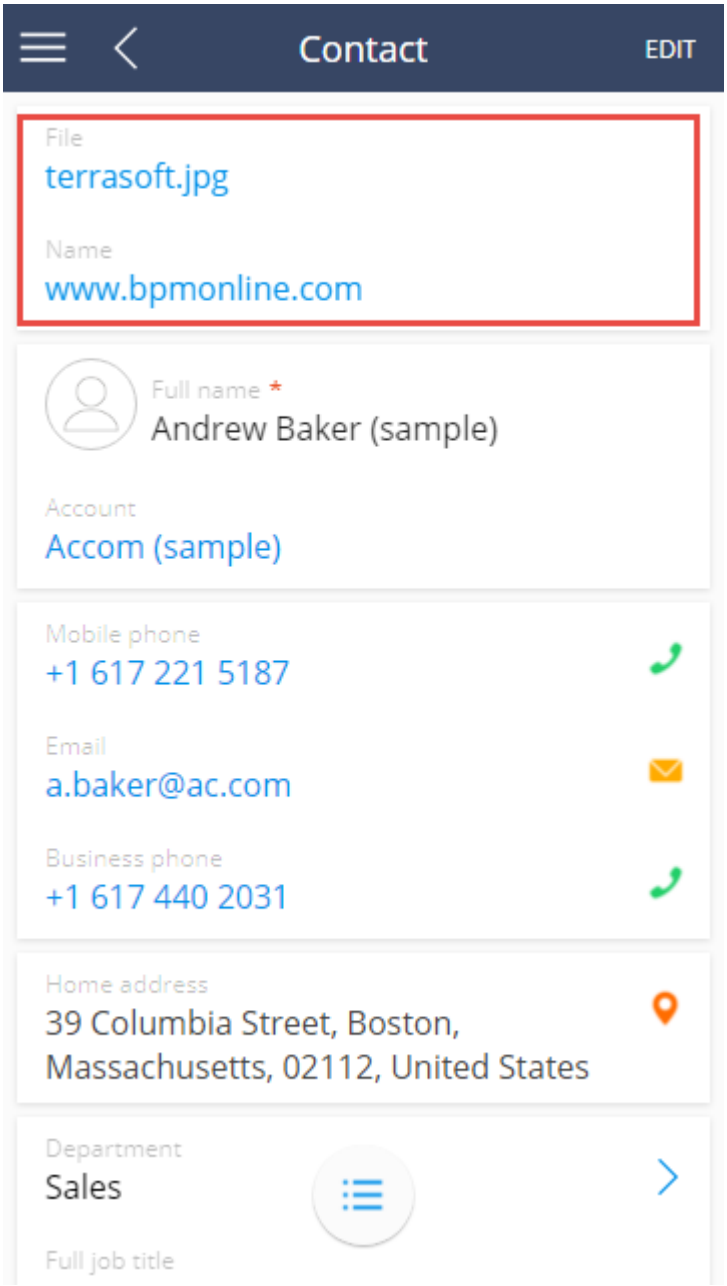
```
{
    "SyncOptions": {
        "SysSettingsImportConfig": [],
        // Configuratio of the import of the required models.
        "ModelDataImportConfig": [
            // File type.
            {
                "Name": "FileType",
                "SyncColumns": []
            },
            //The [Attachment] detail object pf the [Contacts] section.
            {
                "Name": "ContactFile",
                "SyncByParentObjectWithRights": "Contact",
                "SyncColumns": [
                    "Contact",
                    "Type",
                    "Data",
                    "Size"
                ]
```

```
            }
        ]
    },
    // Used models.
    "Models": {
        "Contact": {
            // Configuration of the view mode.
            "Preview": "UsrMobileContactPreviewPage",
            // Configuration of the edit mode.
            "Edit": "UsrMobileContactEditPage",
            "RequiredModels": [
                "ContactFile",
                "FileType"
            ],
            // Configuration of the section.
            "ModelExtensions": [
                "UsrMobileContactModuleConfig"
            ]
        }
    }
}
```

Save the schema to apply changes.

As a result, the [Attachments] detail will appear on the record page of the [Contacts] section (Fig. 2).

Fig. 2. The [Attachments] detail on the contact page



Adding a custom widget to the mobile application

Introduction

The bpm'online mobile application supports dashboards since version 7.10.3 (version 7.10.5 of the mobile application). To receive settings and data for a dashboard, use the AnalyticsService service (see "**Getting the settings and data from the [Dashboards] section**"). The following widget types are supported: "Chart", "Indicator", "List" and "Gauge".

To add a custom widget to the mobile application:

1. Implement a widget setup interface in the bpm'online application.
2. Add the instance of the implemented custom widget to the application.

3. Configure the display of the widget in the mobile application.

> ATTENTION
> This article only describes the implementation of widget display in the mobile application.

To display a custom widget in the mobile application:

1. Implement the data receiving process of a custom widget.
2. Add the implementation of displaying the widget in the mobile application.

Case description

Add a custom widget to the dashboards page of the mobile application.

Case implementation algorithm

1. Implementation of the data receiving process of a custom widget

To receive data of each custom widget type, create a class that will implement the *IDashboardItemData* interface or will be inherited from the *BaseDashboardItemData* base class. To do this, the class must be decoded by the *DashboardItemData* attribute. To implement the class, add the **[Source code]** schema to the custom package.

The class implementation to the *CustomDashboardItem* custom widget type will be as follows:

```
namespace Terrasoft.Configuration
{
    using System;
    using Newtonsoft.Json.Linq;
    using Terrasoft.Core;

    // Attribute indicating the custom widget type.
    [DashboardItemData("CustomDashboardItem")]
    public class CustomDashboardItemData : BaseDashboardItemData
    {
        // Class constructor.
        public CustomDashboardItemData(string name, JObject config, UserConnection
userConnection, int timeZoneOffset)
            : base(name, config, userConnection, timeZoneOffset)
        {

        }
        // A method that returns the required data.
        public override JObject GetJson()
        {
            JObject itemObject = base.GetJson();
            itemObject["customValue"] = DateTime.Now.ToString();
            return itemObject;
        }
    }
}
```

2. Implementation of displaying a custom type of a widget.

2.1. Add a data displaying class

To do this, create a client module in a custom package (for example, UsrMobileCustomDashboardItem). In the created module, implement a class that extends the *Terrasoft.configuration.controls.BaseDashboardItem* base class.

```
Ext.define("Terrasoft.configuration.controls.CustomDashboardItem", {
    extend: "Terrasoft.configuration.controls.BaseDashboardItem",
```

```
    // Displays the value transferred through the customValue property.
    updateRawConfig: function(config) {
        this.innerHtmlElement.setHtml(config.customValue);
    }

});
```

2.2. Add a new type and a class that implements this type to the enumeration.

Add the following source code to the module created on a previous step:

```
Terrasoft.DashboardItemClassName.CustomDashboardItem =
"Terrasoft.configuration.controls.CustomDashboardItem";
```

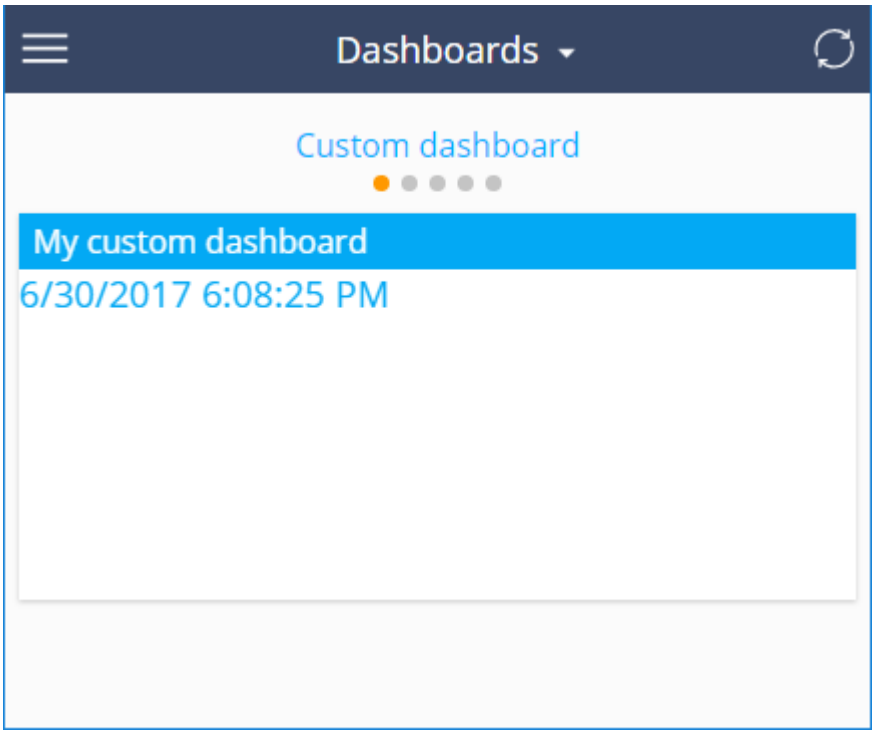
2.3. Add a name of a new custom schema to the mobile application manifest.

In the mobile application manifest file, add the name of the created module schema to the CustomSchemas array:

```
{
    "SyncOptions": {
        ...
    },
    "CustomSchemas": ["UsrMobileCustomDashboardItem"],
    "Modules": {...},
    "Models": {...}
}
```

After saving all changes, the widget will be displayed in the [Dashboars] section of the mobile application (Fig. 1).

Fig. 1. Case result



> ATTENTION
>
> Add the dashboard widget to the main bpm'online application. The mobile application will be synchronized with the main application and the widget will be displayed there.

Service products customization

- **Adding a new rule for the case deadline calculation**
- **Adding a macro handler in email templates**
- **Creating Web-to-Case landing pages**
- **How to hide feed area in the agent desktop**

Adding a new rule for the case deadline calculation

General information

Bpm'online enables implementation of the custom logic for obtaining time parameters for calculating the case deadlines. The strategy implemented by the developer will be used instead of one of the base strategies in the calculation or recalculation of the case deadlines.

The rules are selected from the [Case deadline calculation schemas] lookup. To add a new calculation rule:

1. Create the object schema, add columns for the estimated response time and resolution time and links for the calendar, service agreements and services.

2. Create a lookup based on this schema, fill it with the values for the deadline calculation.

3. Add the source code schema, declare there a class inherited from the *BaseTermStrategy* abstract class. Implement custom mechanism of getting the response and resolution time parameters of the case.

4. Connect a new rule.

Case description

Add custom rule for the calculation of the time parameters of the case for the [Lost data recovery] service according to the [78 – Elite systems] service agreement. Specify the following values for the new rule:

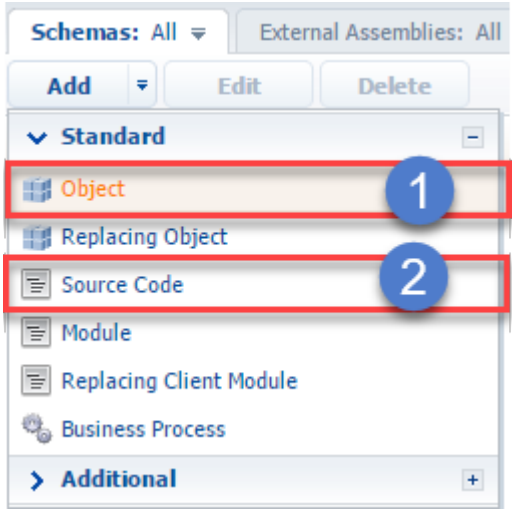
- Response time – 2 calendar hours;
- Resolution time – 1 working day;
- Calendar – [Default calendar].

Case implementation

1. Create an object schema with the columns for calculation

For this, select the custom package in the [Configuration] service on the [Schemas] tab execute the [Add] > [Object] command (Fig. 1, 1).

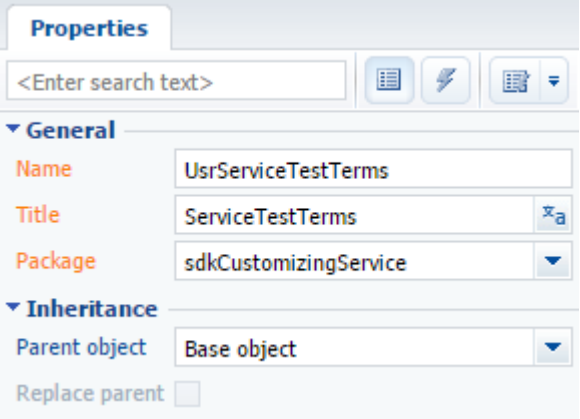
Fig. 1. Adding the schema



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

- [Name] – "UsrServiceTestTerms";
- [Title] – "ServiceTestTerms";
- [Package] – the user package where the development is taking place, for example the *Custom* package;
- [Parent object] – the [*BaseEntity*] schema of the *Base* package.

Fig. 2. The properties of the added object schema



For the created schema, create the columns with the properties as specified in the table 1.

Table 1. The properties of the added columns

Name	Header	Type	Details
UsrReactionTimeUnit	Response time unit	[Time unit] lookup	Specifies the time unit (calendar days, hours, etc.) that will be used for the [Response time] calculation.
UsrReactionTimeValue	Response time	Integer	A column for the response time value.
UsrSolutionTimeUnit	Resolve time unit	[Time unit] lookup	Specifies the time unit (calendar days, hours, etc.) that will be used for the [Resolution time] calculation.
UsrSolutionTimeValue	Resolution time	Integer	A column for the resolution time value.
UsrCalendarId	Calendar that is used	[Calendar] lookup	A calendar that will be used in the calculation of the case deadlines.
UsrServicePactId	Service agreement	[Service agreement] lookup	A link to the [Service agreement] object. Added for the filtering purposes.
UsrServiceItemId	Service	[Service] lookup	A link to the [Service] object. Added for the filtering purposes.

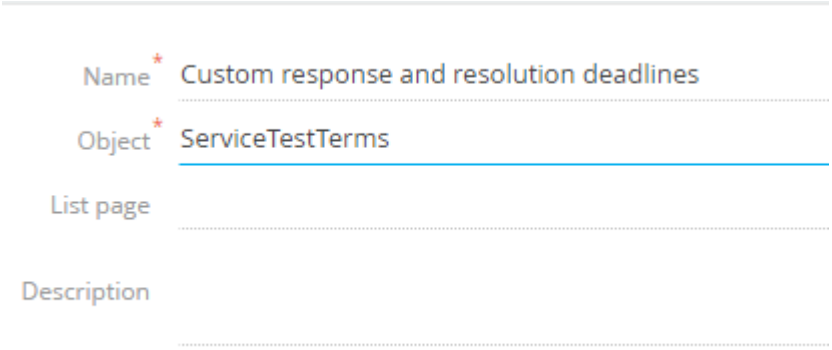
Publish the object schema after adding the columns.

2. Create a lookup; fill it with the values for the deadline calculation

Set the exact values of the response and resolution deadlines for their calculation. For this, create the lookup based on the created schema with following properties (Fig. 3):

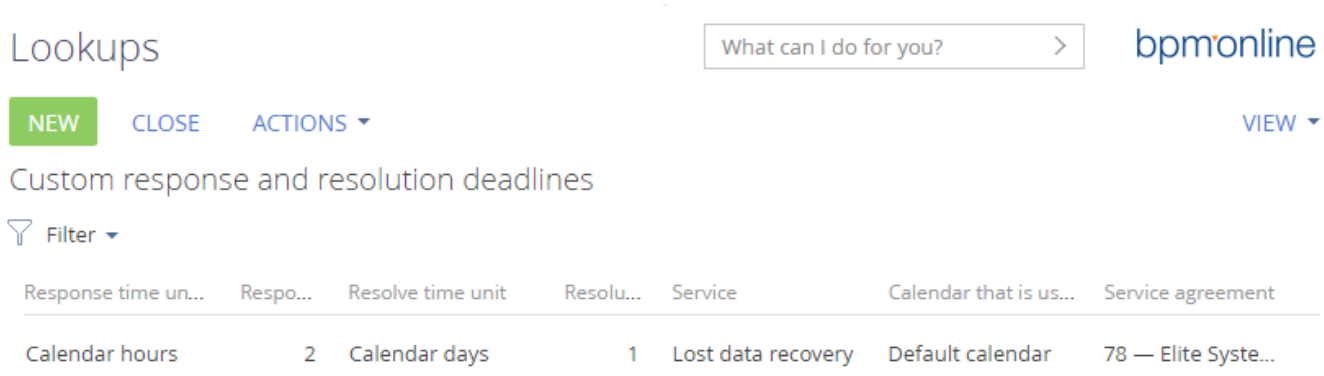
- [Name] – "Custom response and resolution deadlines";
- [Object] – ServiceTestTerms.

Fig. 3. The properties of the added lookup



Add at least one record in the lookup. Add the following data according to case conditions (Fig. 4):

Fig. 4. A record in the added lookup that meets the case conditions



3. Implement the class with the mechanism of getting the time parameters

Add the source code schema (Fig. 1, 2). Add a class inherited from the *BaseTermStrategy* abstract class declared in the *Calendar* package to the schema source code. Implement a parameterized constructor in the class, with the following parameters:

- *UserConnection userConnection* – current user connection;
- *Dictionary<string, object> args* – the arguments for the calculation.

Implement the *GetTermInterval()* abstract method that is declared in the base class. This method receives the mask of the filled values as an input parameter. The decision to fill exact time parameter of the *TermInterval* class that implements the *ITermInterval<TMask>* interface is based on this mask.

The 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>
    {
        // A container class for storing data received from an entry point.
        protected class StrategyData
        {
            public Guid ServiceItemId {
```

```
                get;
                set;
            }
            public Guid ServicePactId {
                get;
                set;
            }
        }
        //A field that stores data from the entry point.
        protected StrategyData _strategyData;
        // Parameterized constructor, for correct
        // initialization by a class-selector.
        public ServiceTestTermsStrategy(UserConnection userConnection,
Dictionary<string, object> args)
            : base(userConnection) {
            _strategyData = args.ToObject<StrategyData>();
        }
        // A method that receives data and returns it in an instance of the
CaseTermInterval class.
        public override CaseTermInterval GetTermInterval(CaseTermStates mask) {
            var result = new CaseTermInterval();
            // Create an EntitySchemaQuery query.
            var esq = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"UsrServiceTestTerms");
            // Adding columns to a 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 a query.
            esq.CreateFilterWithParameters(FilterComparisonType.Equal,
"UsrServiceItemId", _strategyData.ServiceItemId);
            esq.CreateFilterWithParameters(FilterComparisonType.Equal,
"UsrServicePactId", _strategyData.ServicePactId);
            // Executing and processing query results.
            EntityCollection entityCollection =
esq.GetEntityCollection(UserConnection);
            if (entityCollection.IsNotEmpty()) {
                // Addition to the return value of the reaction time.
                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 to the returned value of resolve time.
                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. Connect the new rule

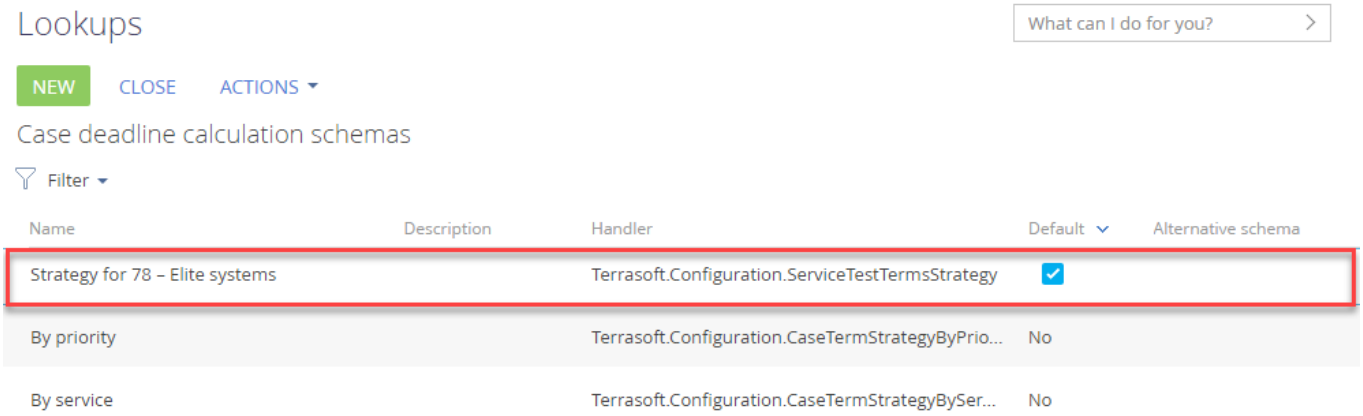
For this, add the value in the [Case deadline calculation schemas] lookup. Specify the full qualified name of the created class in the [Handler] column (including the namespace).

In the [Alternative schema] column you can specify the deadline calculation rule used when the deadline calculation with current schema is impossible. Note that the alternative schema class will be created if any of time parameters is not calculated by the strategy class. If the alternative schema cannot calculate the deadline, then its alternative strategy instance will be created. The queue of rules is formed in this way.

Check the [Default] checkbox at adding new record.

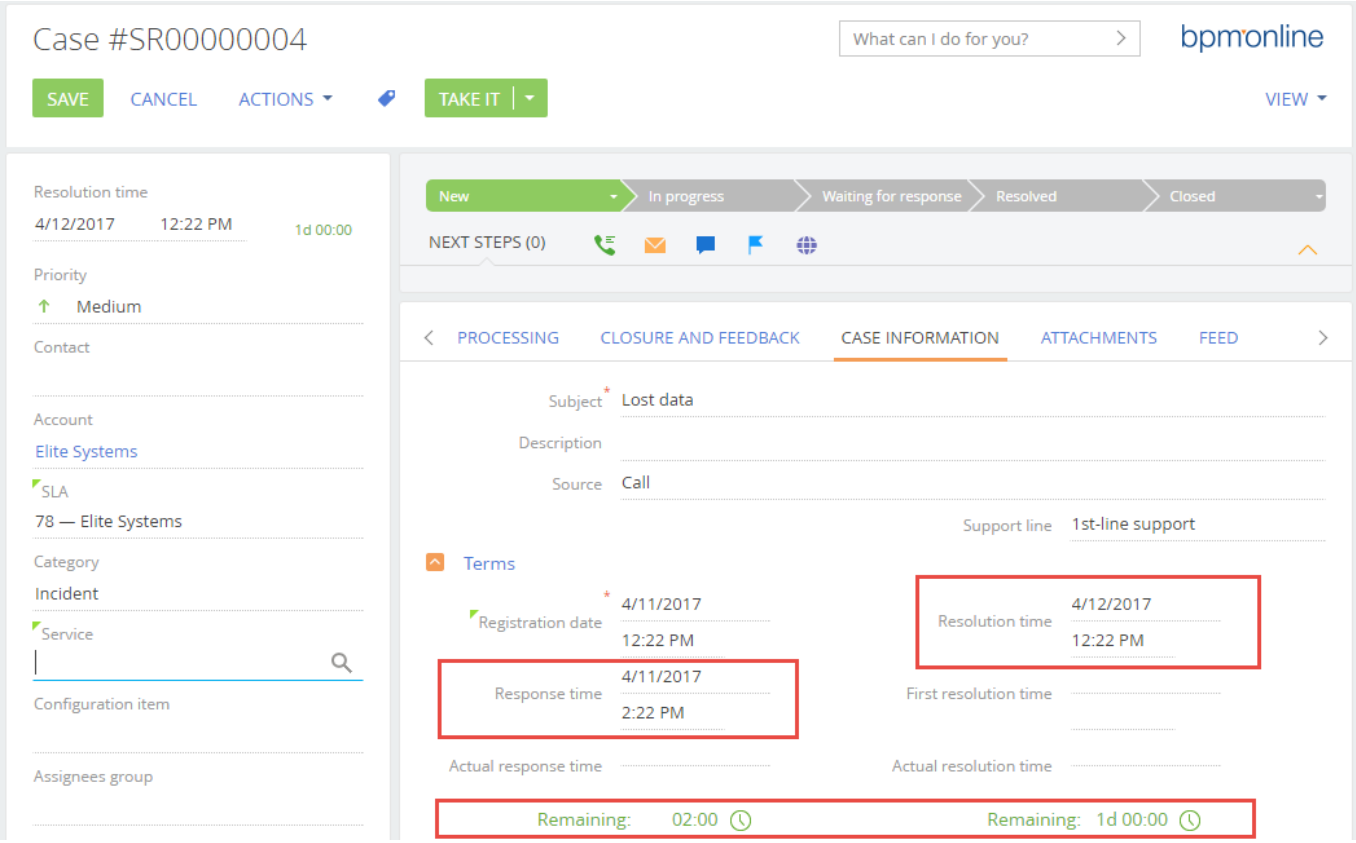
The example of adding the record to the [Case deadline calculation schemas] lookup (Fig. 5).

Fig. 5. The record of the custom deadline calculation rule



As a result, the new rules of the deadline calculation will be applied for the cases of the [78 – Elite systems] service agreement for the [Lost data recovery] service (Fig. 6).

Fig. 6. Case result



Adding a macro handler in email templates

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

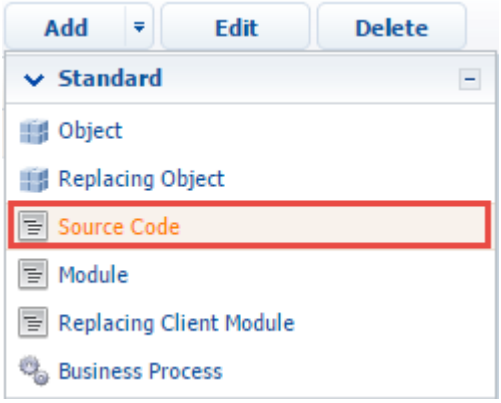
Add an email template macro handler that will return the "Test" string.

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:

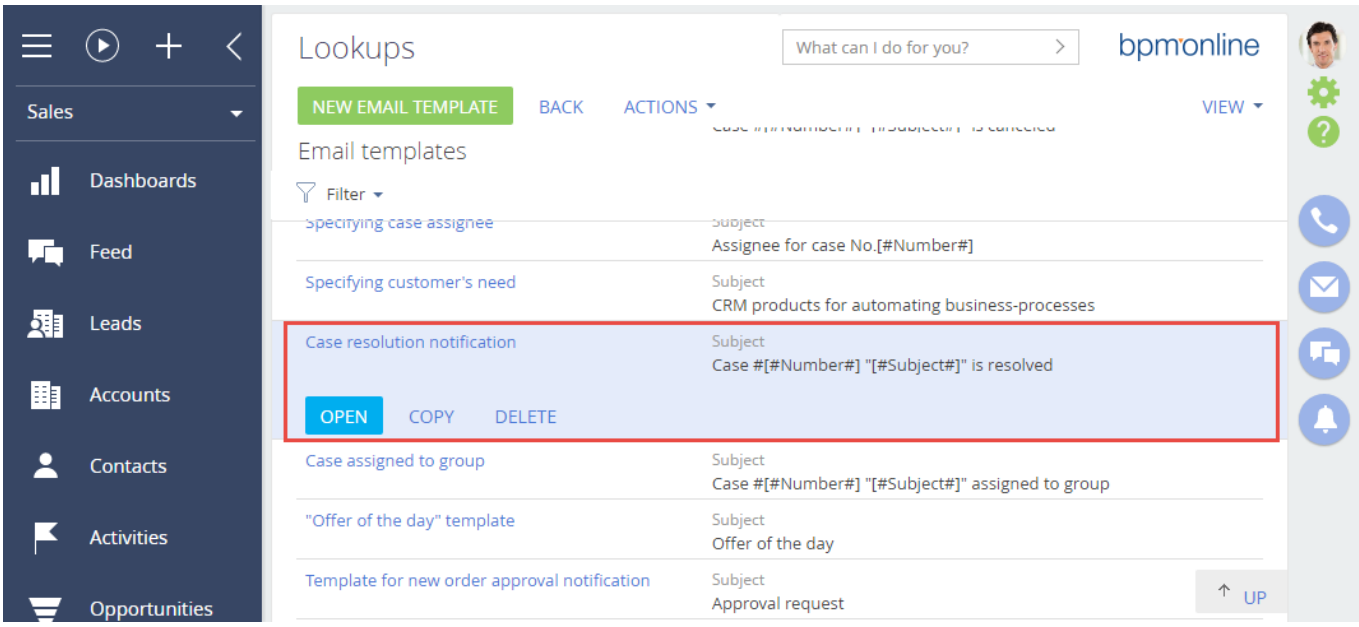
```
INSERT INTO EmailTemplateMacros(Name, Parentid, ColumnPath)
VALUES (
    'UsrTestStringGenerator',
    (SELECT TOP 1 Id
    FROM EmailTemplateMacros
```

```
    WHERE Name = '@Invoke'),
    'Terrasoft.Configuration.UsrTestStringGenerator'
)
```

3. Adding a macro handler in the email template

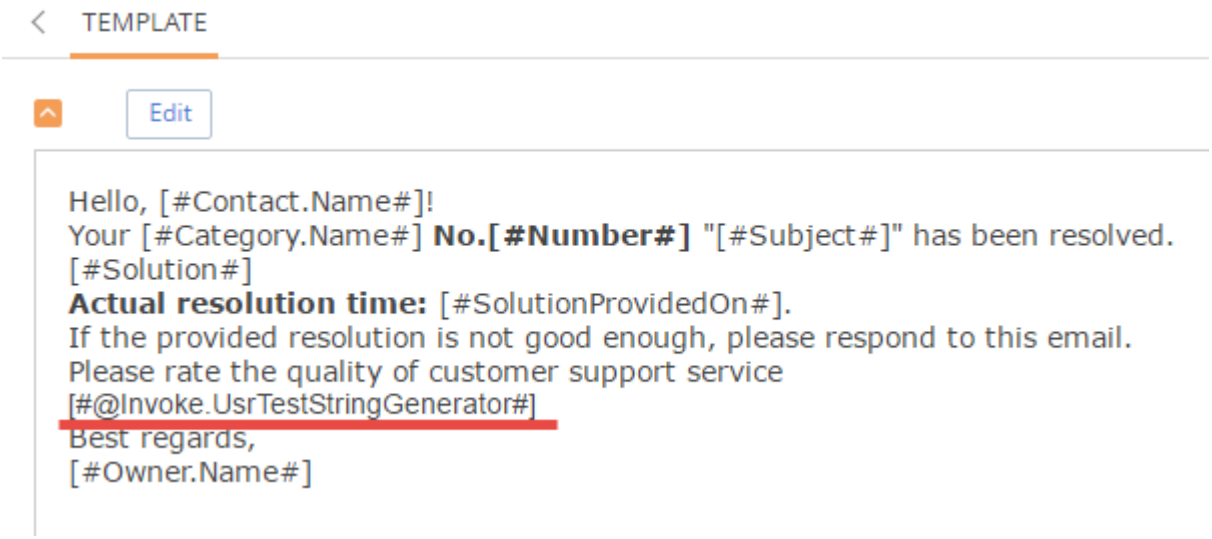
After registering the macro, you can start using it in email templates. To do this, you must change one or more records in the [Email templates] lookup (Fig. 2).

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 [Landings] 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.

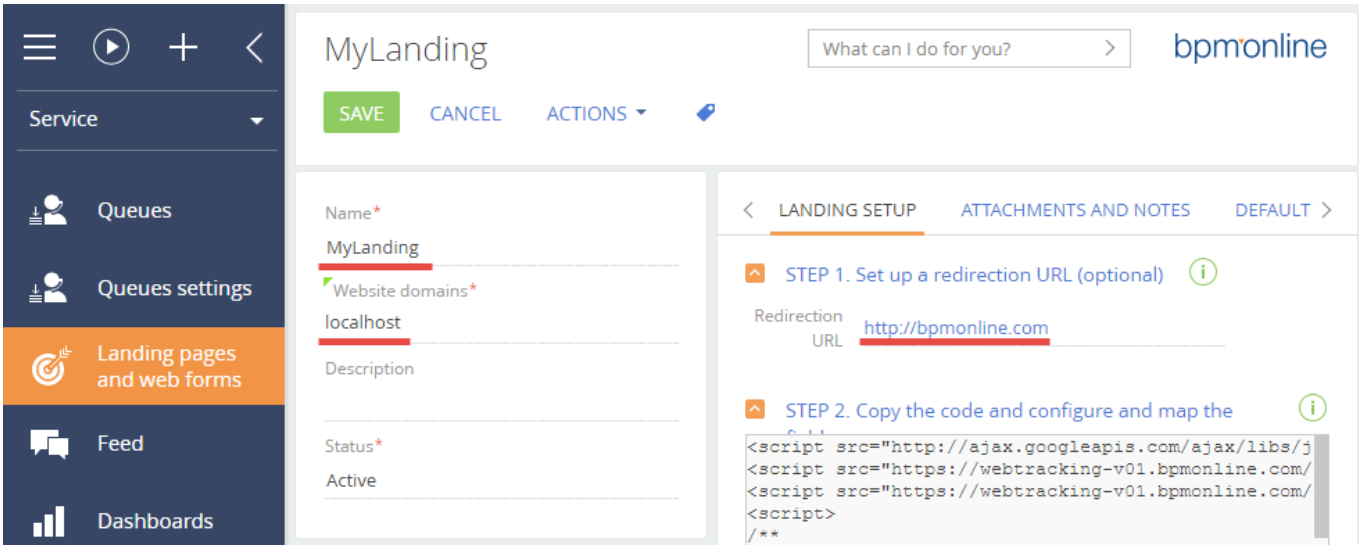
Steps to create Web-to-Case landing

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):

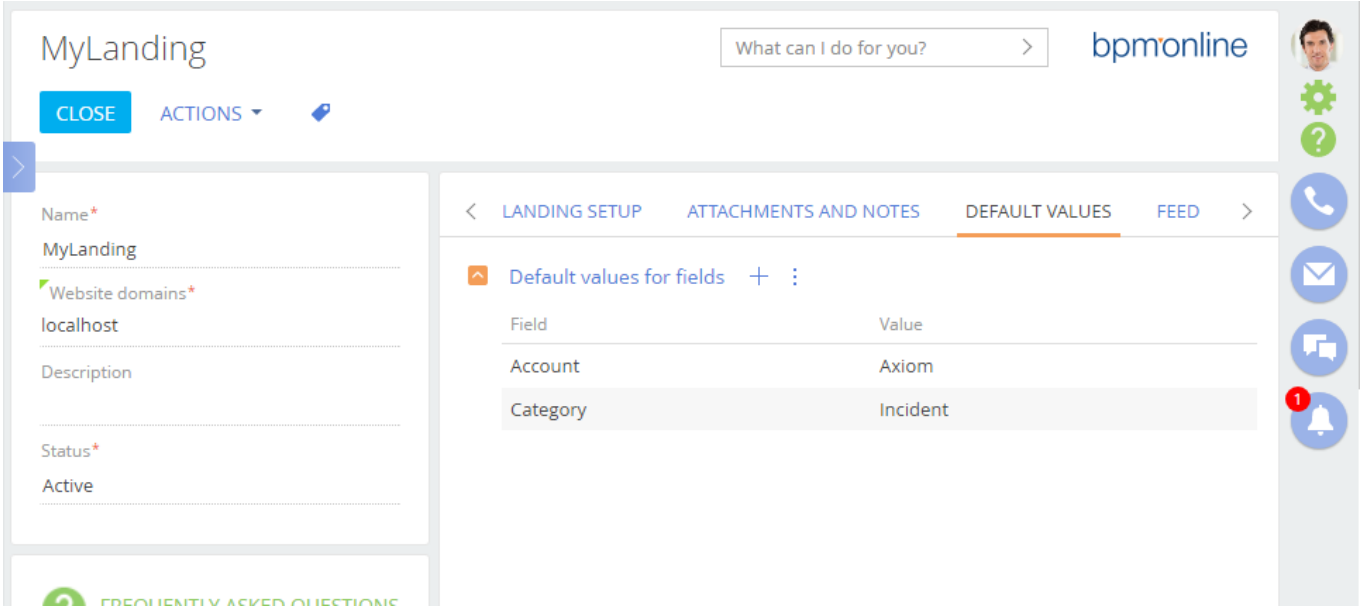
- [Name] – landing page name in bpm'online.
- [Website domains] – your landing page URL.
- [Status] – landing status.
- [Redirection URL] – the URL that is opened after the landing page form is completed.

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>
```

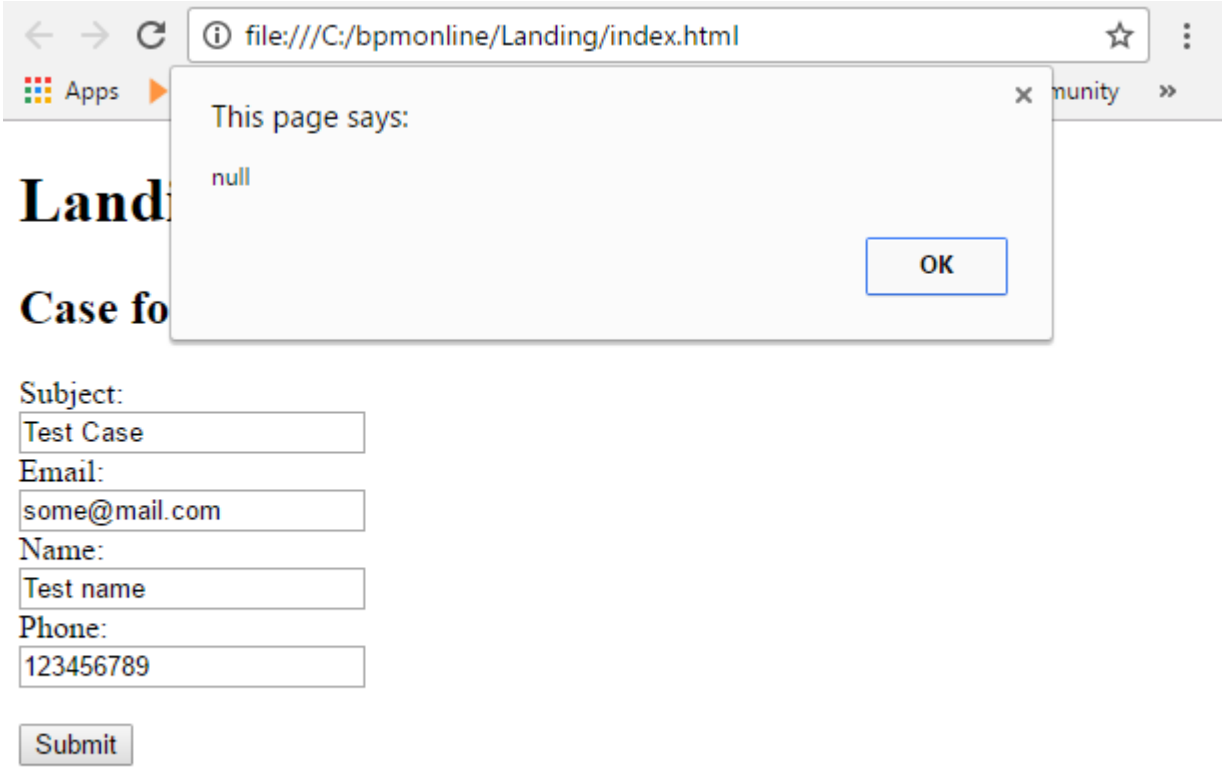
```
<head>
    <meta charset="UTF-8">
    <!--STEP 2-->
    <!--This part needs to be copied from the STEP 2 field of the lending edit page--
>
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js">
</script>
    <script src="https://webtracking-v01.bpmonline.com/JS/track-cookies.js"></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.

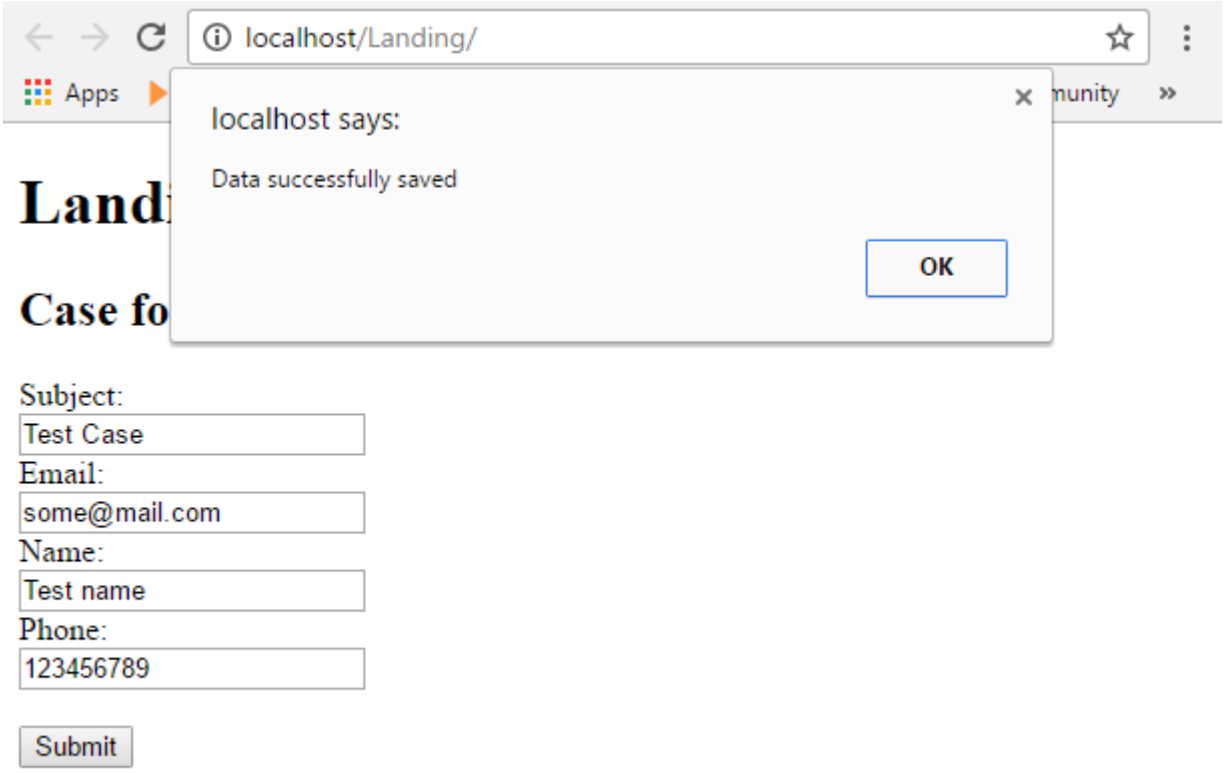
Fig. 3 Empty message



> NOTE
>
> The output of an empty message is configured in the *onError()* method of the configuration object.

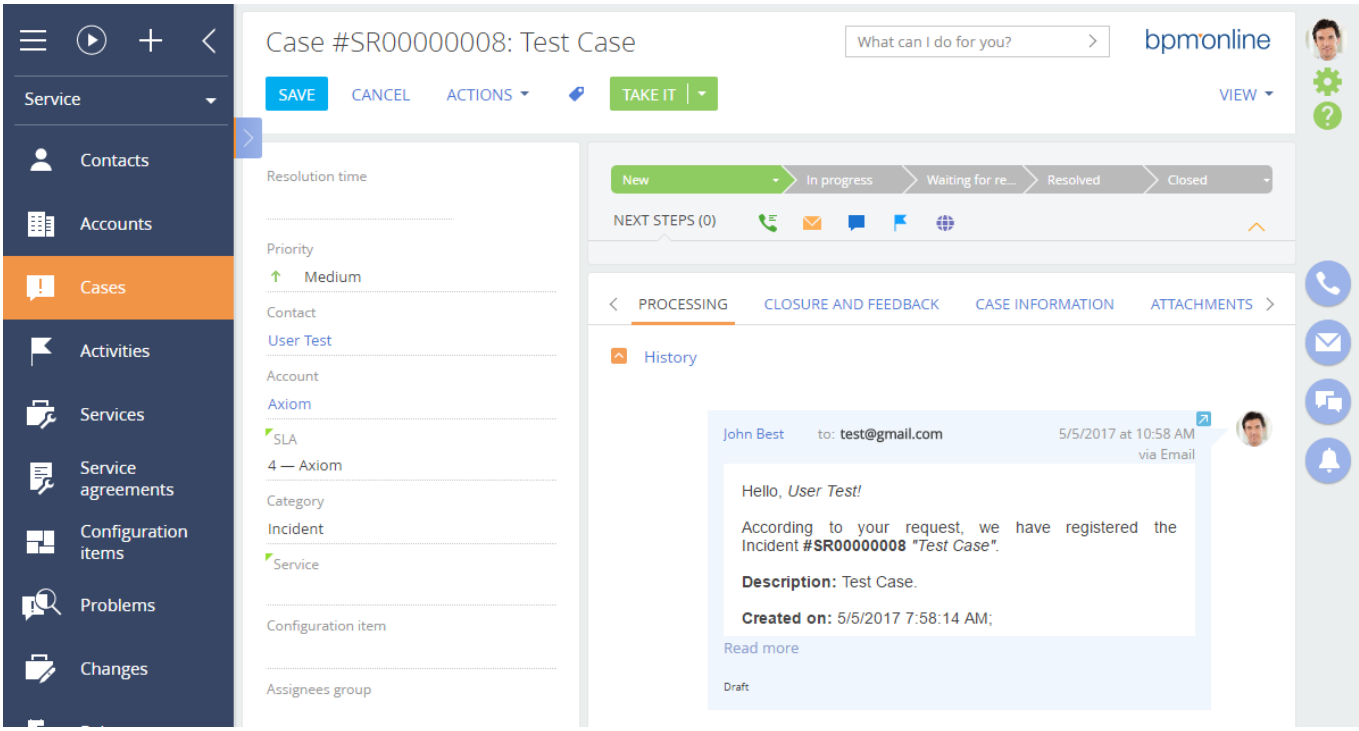
If you place the page on the local server of the computer that serves as the reserved domain name localhost (as specified in the landing setting , Fig. 1), then the script that adds the address from the web page of the landing will work correctly (Fig. 4)

Fig. 4 The correct adding of data



As a result, a case with specified parameters will be automatically created.

Fig. 5 Automatically created case

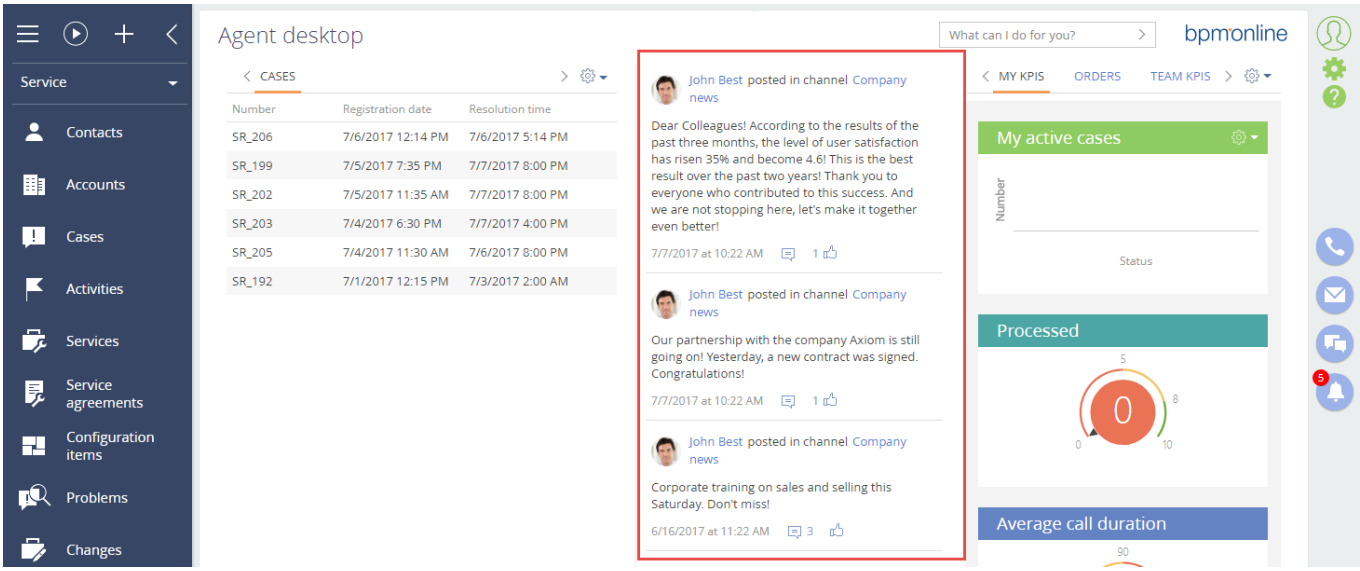


How to hide feed area in the agent desktop

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:

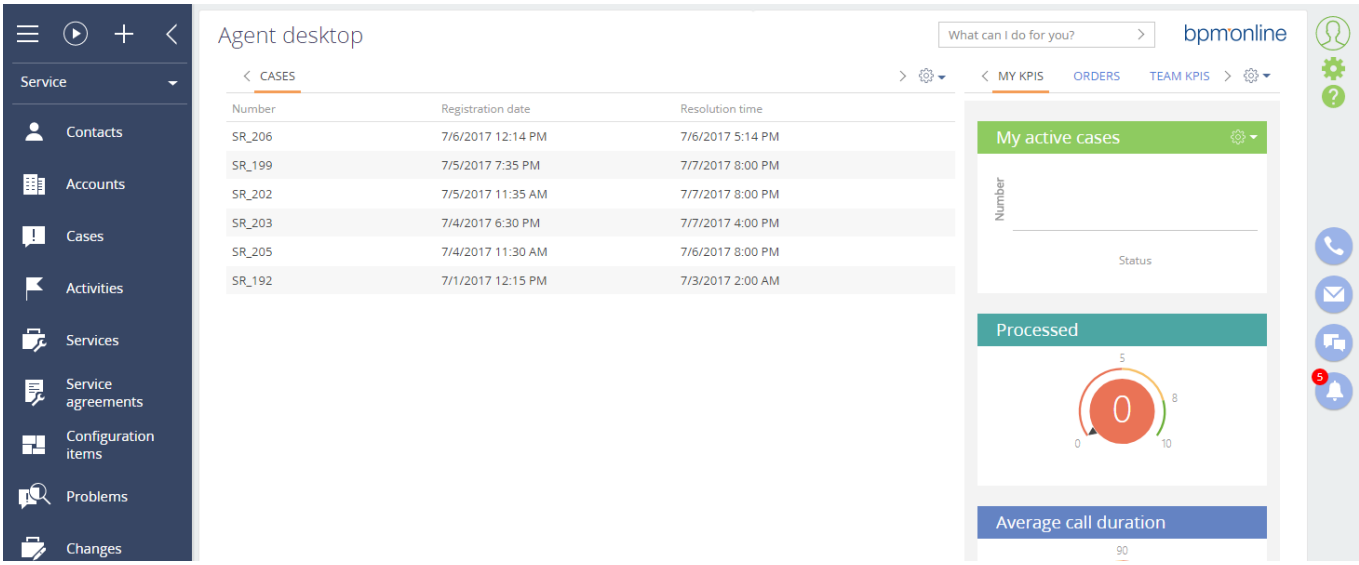
```
define("OperatorSingleWindowPage", [],
    function() {
        return {
            methods: {
                // Replacing the base method to exclude the ENSFeedModule feed module
from the loaded modules.
                loadContent: function() {
                    // the ESNFeedModule module 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



Lending product customization

- **How to create custom verification action page**
- **Using the EntityMapper schema**

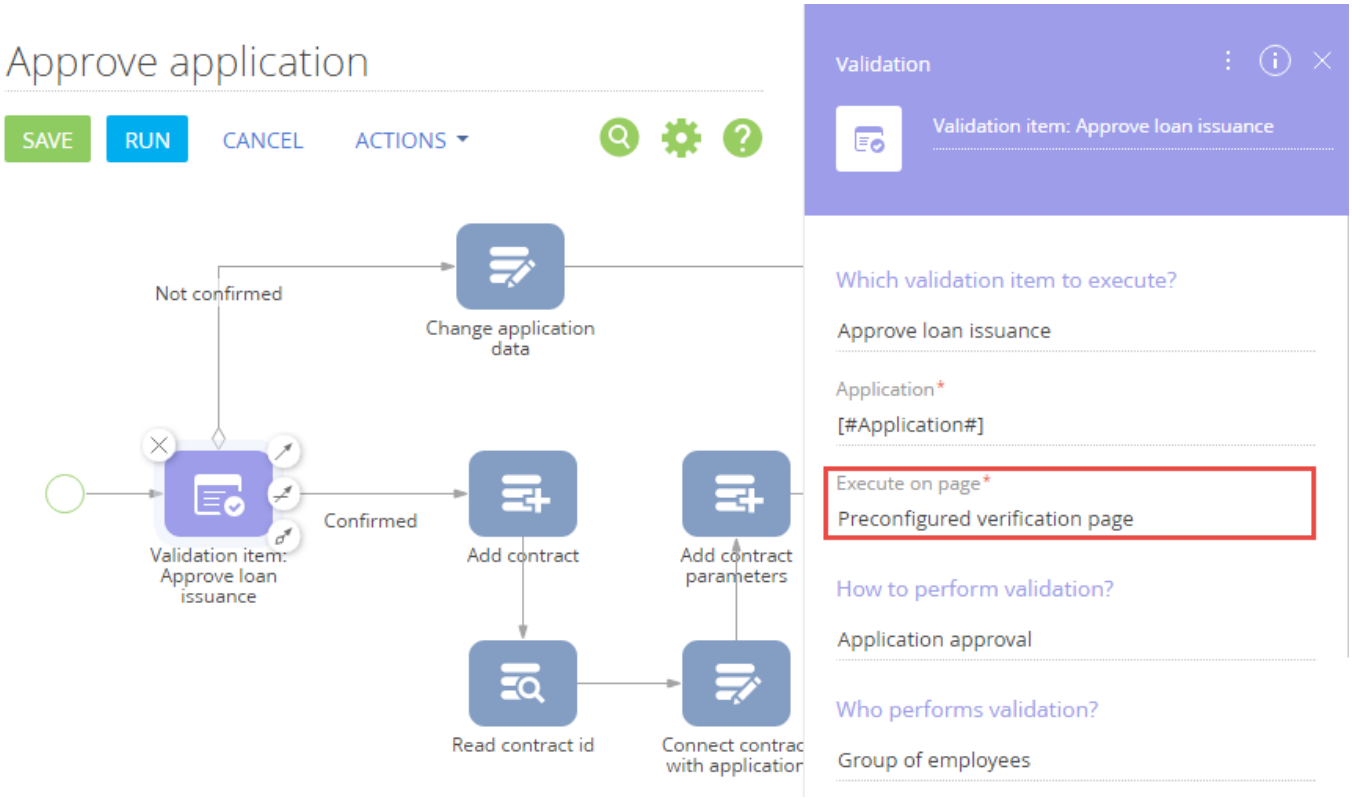
How to create custom verification action page

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.

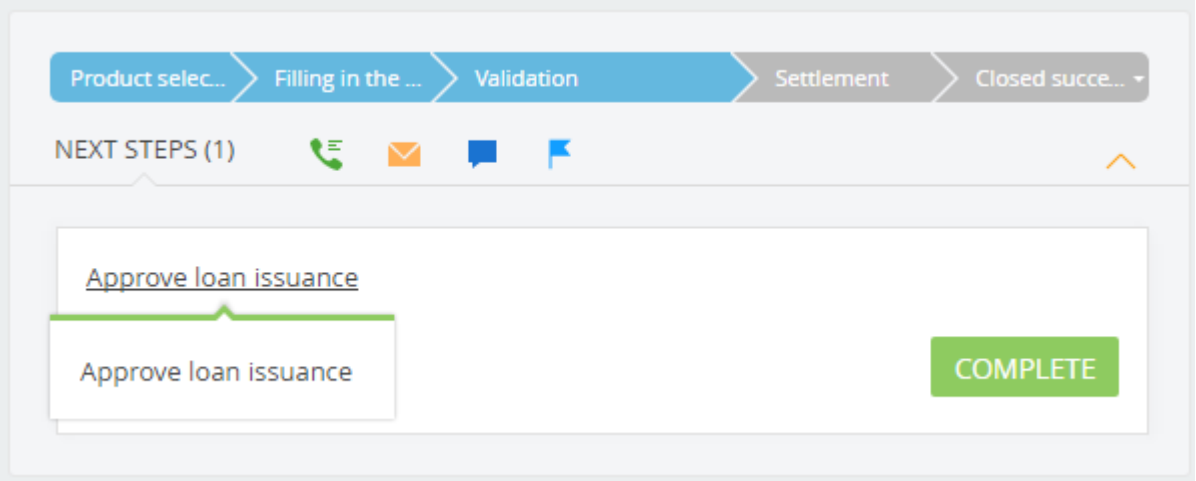
When creating a custom verification action page, for example, in the *Approve application* business process, you can select the [Preconfigured verification page].

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

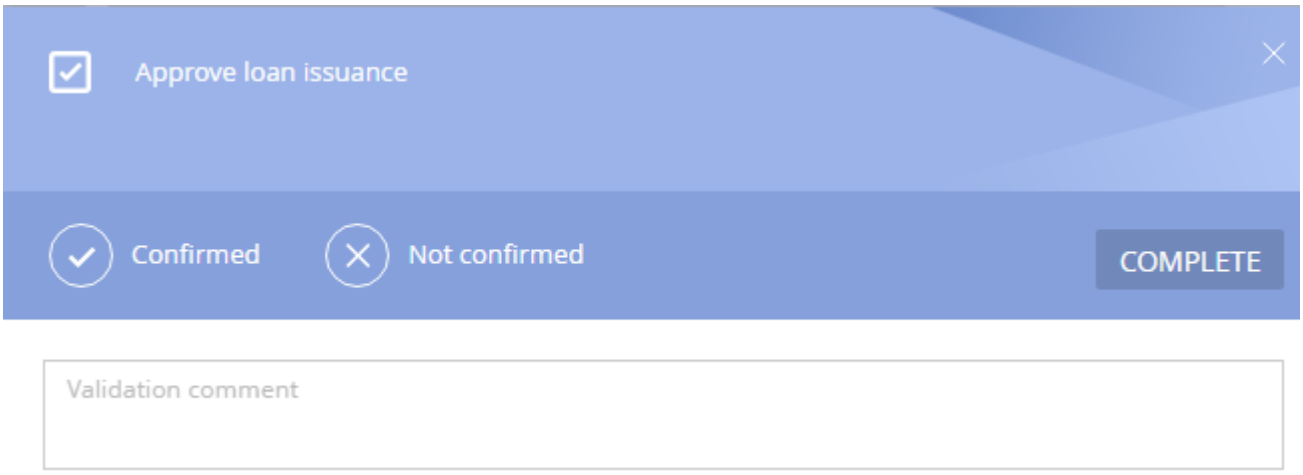


Preconfigured verification page contains (Fig. 3):

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

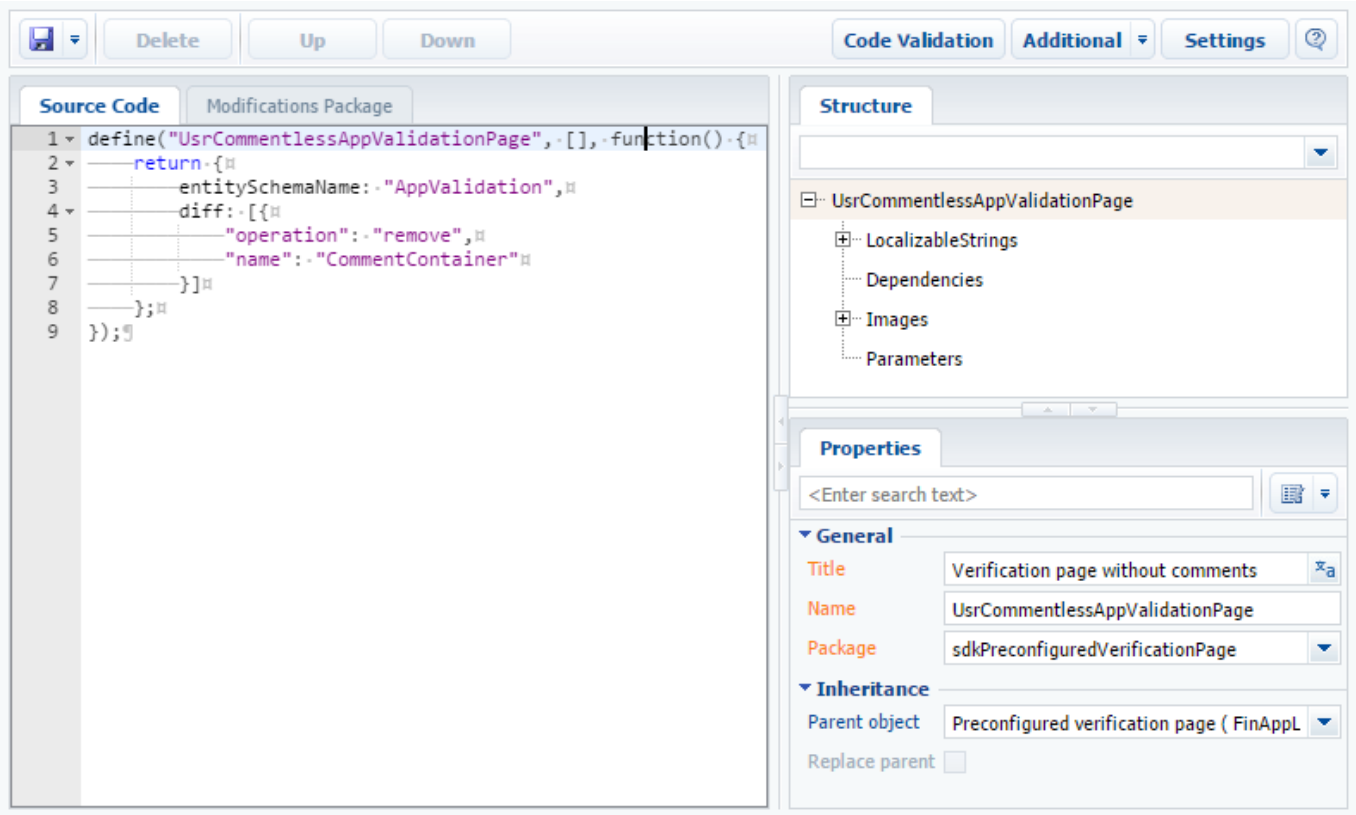
1. Create a schema of the verification action page

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"
        }]
    };
});
```

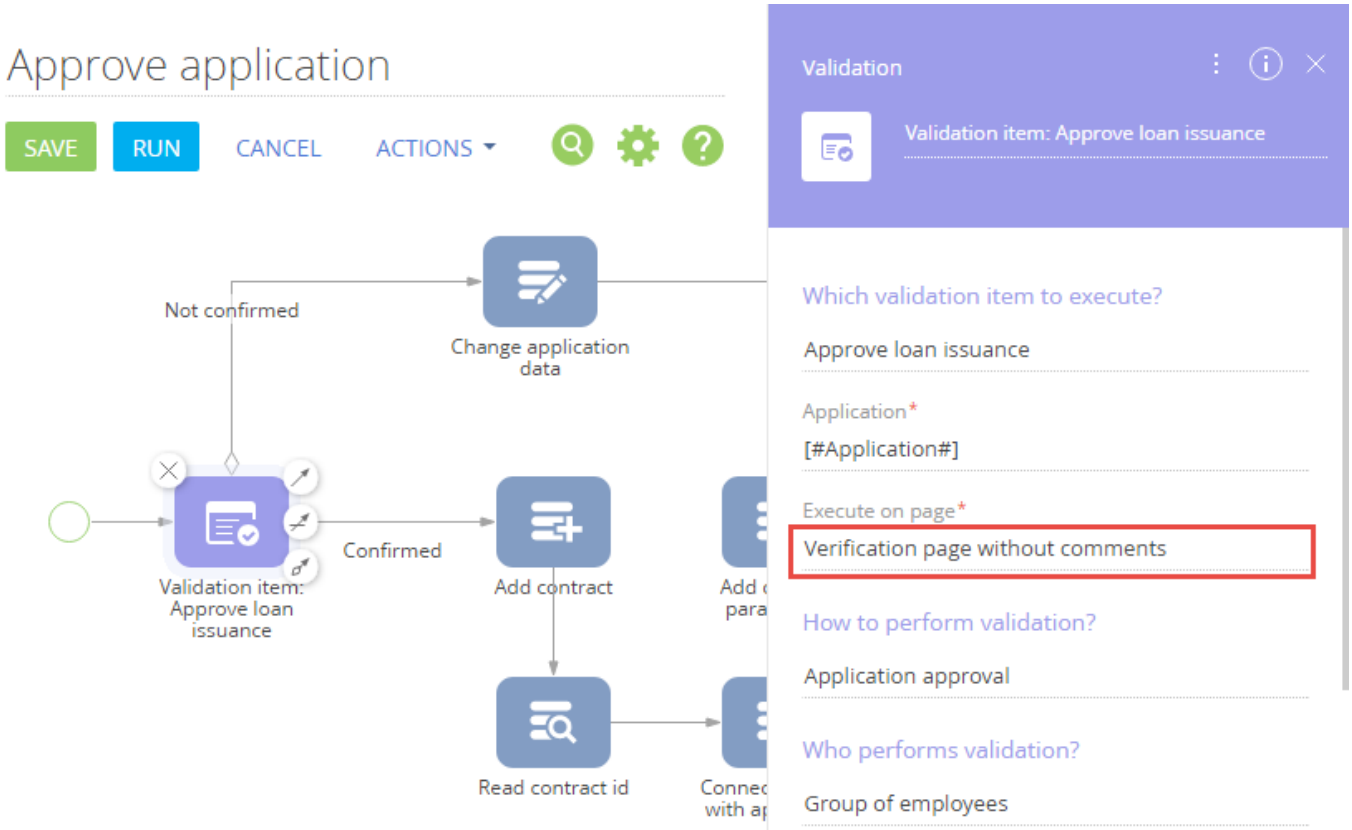
The [Comment] field is removed from the parent element in the **diff array**.

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

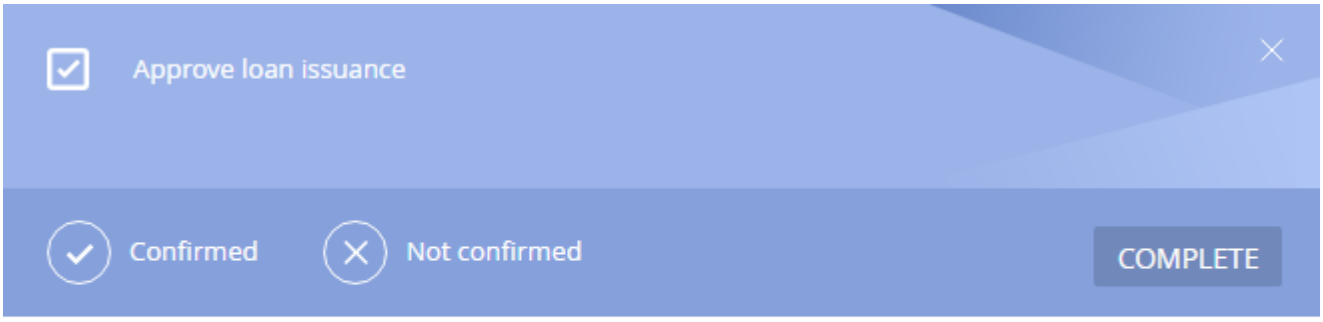


Save the business process to apply the changes.

> 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

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 own 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.

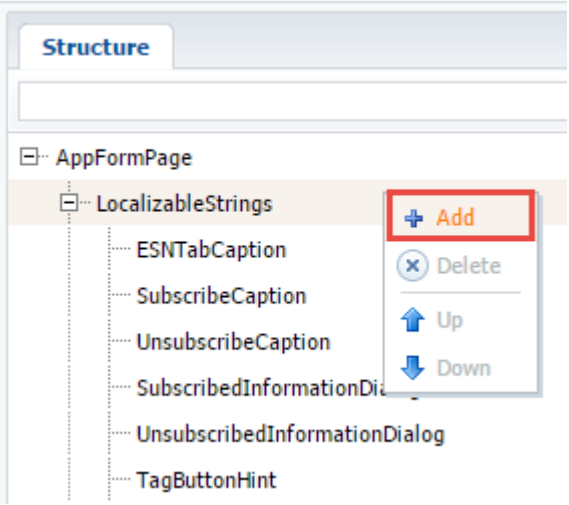
3. Adding a data mapping button to the application form edit page

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*/
    };
});
```

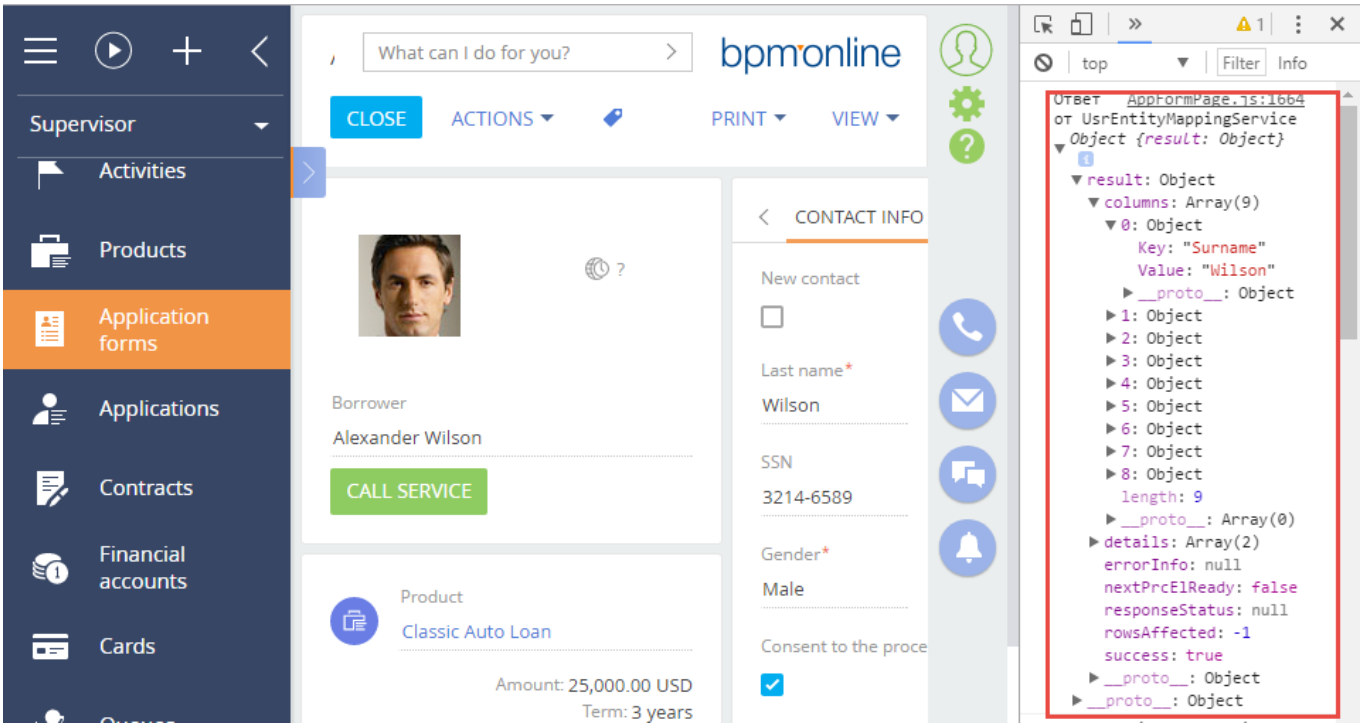
Add the configuration object to the **diff array**. The object will be used to add the data matching button to the application form edit page. The *requestContactData2()* method is called when the button is pressed, and the *UsrEntityMappingService* configuration service is called in the method with 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*.

Recommended integration options

- **Authenticating external requests to bpm'online web-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**

Authenticating external requests to bpm'online web-services

Introduction

All external requests to bpm'online web-services must be authenticated.

Two types of authentication are supported:

- **Basic authentication**. This is a standard authentication method that uses HTTP requests. User credentials (login and password) are added to the request title in Base64 encoding.

> Attention!
>
> Using Basic authentication, you can perform user authorization only in **DataService** and **OData**. We recommend **using AuthService.svc and corresponding cookies** for all other external requests.

> Attention!
>
> The Basic authentication is not a secure authentication method, since the data are passed unprotected. Use this method only if bpm’online is accessed via the SSL (HTTPS) protocol.
>
> To secure the transfer of the user credentials, it is recommended to authenticate external requests via the AuthService.svc.

> Attention!
>
> To use Basic authentication in a bpm'online-integrated application, turn off protection from CRSF attacks.

- **Forms authentication**. This is a standard authentication method that uses cookie. The AuthService.svc web service is used for the forms authentication.

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

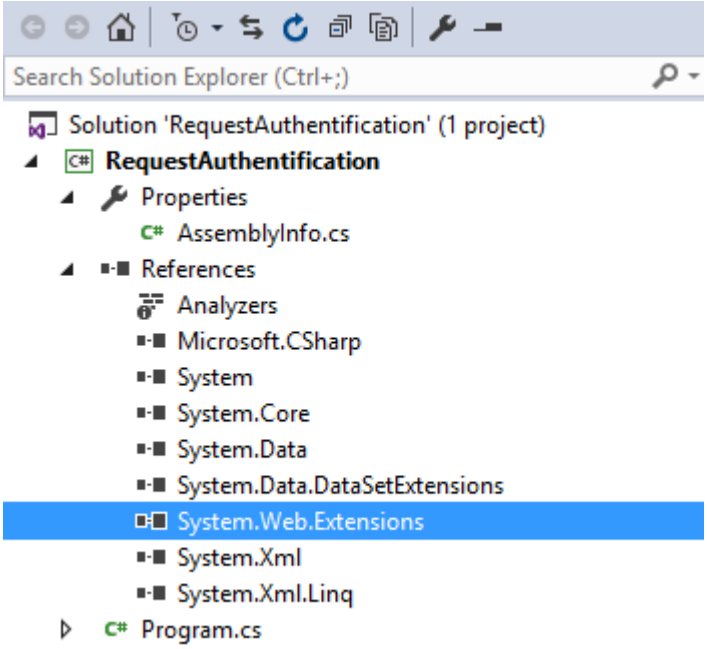
Property	Description
Code	Authentication status code. The authentication is successful if the value is "0". Otherwise the authentication has failed.
Message	The message that contains the reason for failing the authentication.
Exception	The object that contains a detailed description of an exception that 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

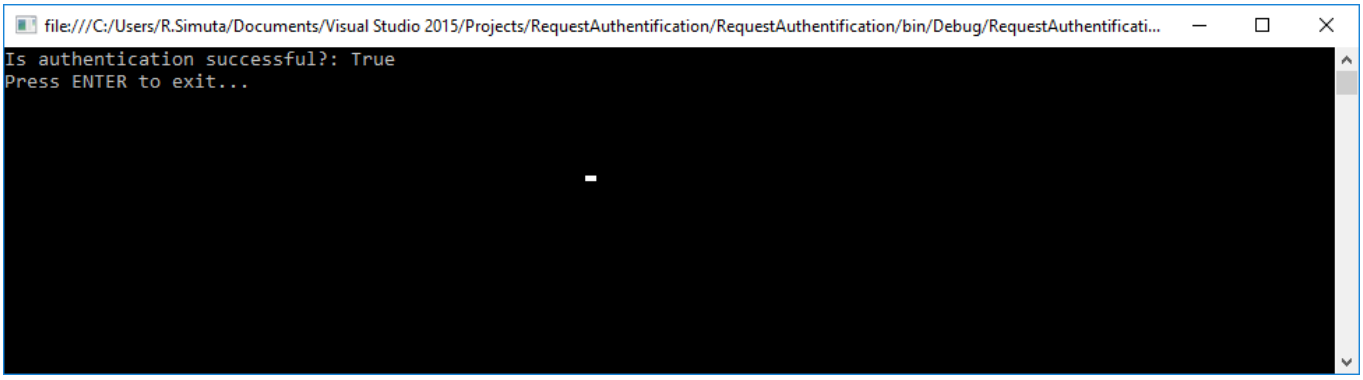
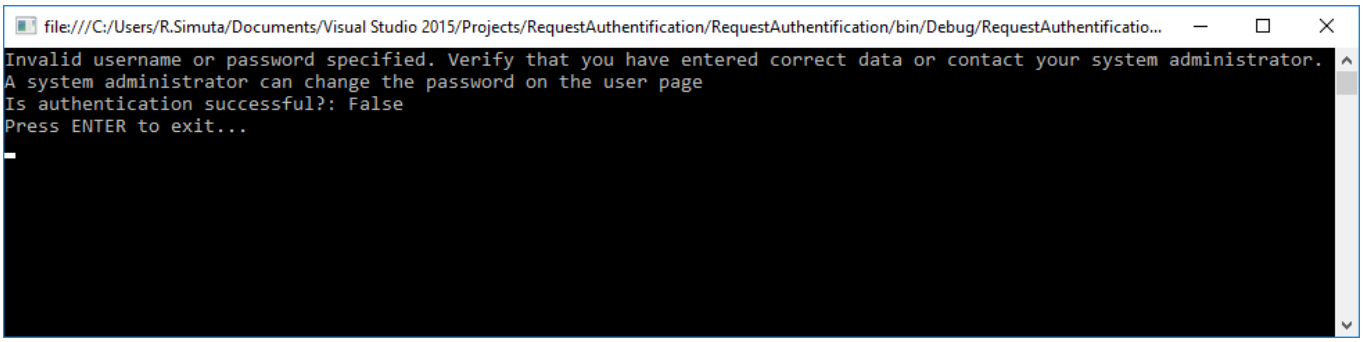


Fig. 3. Failed authentication



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 web-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.

DataService web service

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.

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 2012 Update 4) development environment, create a Visual C# console application project and name it *DataServiceInsertExample*. The [Target framework] project property must be set to .NET Framework 4.5.

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 web-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

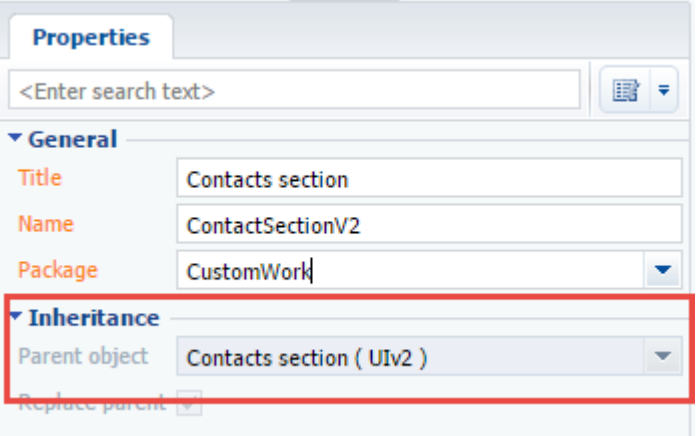
Case implementation algorithm

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.

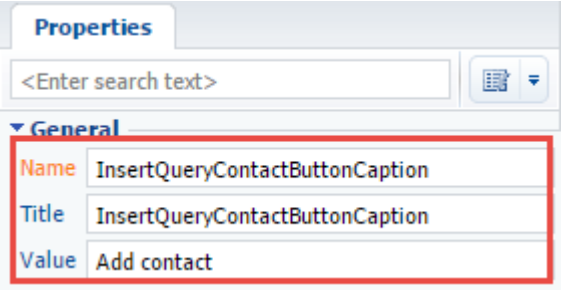
For this particular case, you need to create a replacement client module of the [Contacts] section (Fig. 1).

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

General provisions

The DataService web service of bpm'online is a RESTful (Representational State Transfer, REST) service. 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.

SelectQuery data contract

The SelectQuery data contract is used for reading 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 read data from DataService.
http(s)://[Bpm'online application address]/[Configuration number]/dataservice/[Data
fromat]/reply/SelectQuery
// URL example of the POST query to read data from DataService.
http(s)://example.bpmonline.com/0/dataservice/json/reply/SelectQuery
```

The SelectQuery data contract has a complex hierarchical structure with a number of nesting levels. In the bpm'online server core, it is represented by a SelectQuery class of theTerrasoft.Nui.ServiceModel.DataContract namespace of the Terrasoft.Nui.ServiceModel.dll library of classes. The hierarchical data structure of the SelectQuery data contract can be conveniently viewed in JSON format:

```
{
    "RootSchemaName":"[Object root schema name]",
    "OperationType":[Type of record operation],
    "Columns":{
        "Items":{
            "Name":{
                "OrderDirection":[Sorting order],
```

```
                "OrderPosition":[Column position],
                "Caption":"[Title]",
                "Expression":{
                    "ExpressionType":[Expression type],
                    "ColumnPath":"[Path to column]",
                    "Parameter":[Parameter],
                    "FunctionType":[Function type],
                    "MacrosType":[Macro type],
                    "FunctionArgument":[Function argument],
                    "DatePartType":[Type of date part],
                    "AggregationType":[Aggregation type],
                    "AggregationEvalType":[Aggregation scope],
                    "SubFilters":[Buit-in filters]
                }
            }
        }
    },
    "AllColumns":[Indicates that all columns are selected],
    "ServerESQCacheParameters":{
        "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 2012 Update 4 and up), create a Visual C# console application project and specify project name, for example, DataServiceSelectExample. Set ".NET Framework 4.5" 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 web-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.

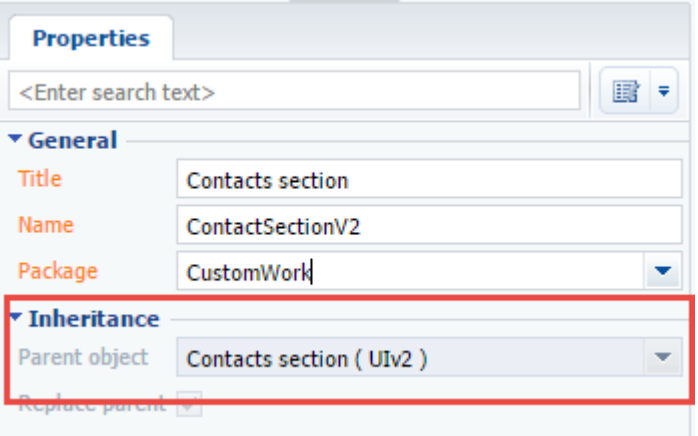
Case implementation algorithm

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**".

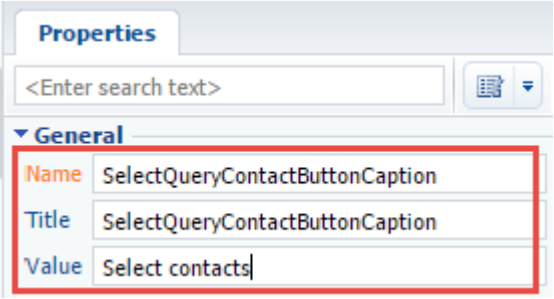
Create a replacing client module of the [Contacts] section (Fig. 1).

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

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

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.

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 2012 Update 4 and up), create a Visual C# console application project and specify project name, for example, DataServiceUpdateExample. Set ".NET Framework 4.5" 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 web-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

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.

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 2012 Update 4) development environment, create a Visual C# console application project and name it *DataServiceDeleteExample*. The [Target framework] project property must be set to .NET Framework 4.5.

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 web-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

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.

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 2012 Update 4 and up), create a Visual C# console application project and specify the project name, for example, *DataServiceBatchExample*. Set ".NET Framework 4.5" 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 web-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

- **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**

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 pplication_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

```
http://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 web-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. 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.

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 *00000000-0000-0000-0000-000000000000* 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 DELET*E 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 *00000000-0000-0000-0000-000000000000* 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/bpm'onlineWebApp/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 web-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

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.

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:

Template	Name	Description
		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'
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?$filter=Price mul 2.00m eq 35.55m
div	Division	Select all products, for which the price (the [Price] field) meets the condition (Price / 2) = 35.55 Request string: GET <BPMonline application address>/0/ServiceModel/EntityDataService.svc/ProductCollection?$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:** [0-9]+	Example: 255
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	Example: 2.0f

Data type	Definition	Examples
	Entry mask. **Entry mask:** [0-9]+.[0-9]+f	
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
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'

Integration of third-party sites via iframe

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:

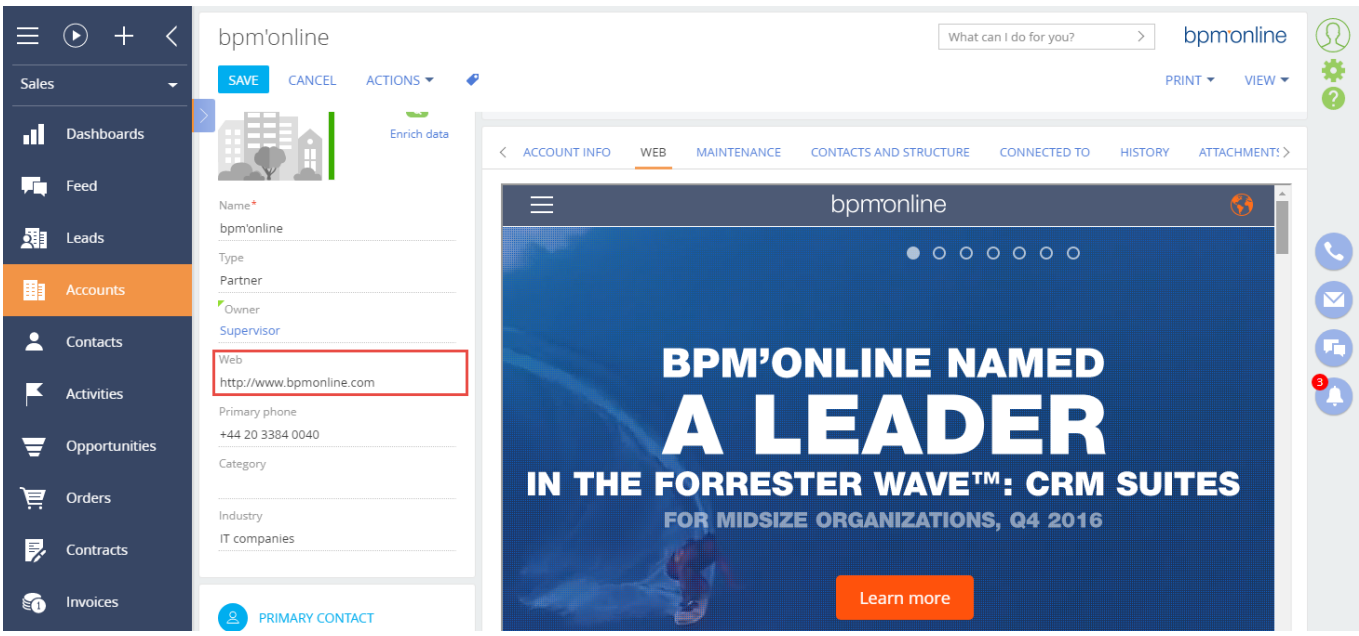
```
// Add a module in which the component is implemented in an array of dependencies.
define("AccountPageV2", ["UsrIframeControl", "css!UsrIframeControl"], function() {
    return {
        entitySchemaName: "Account",
        diff: /**SCHEMA_DIFF*/[
            // Adding the [WEB] tab.
            {
                "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


bpm'online
What can I do for you?
bpm'online
Sales
SAVE
CANCEL
ACTIONS
PRINT
VIEW
Dashboards
Feed
Leads
Accounts
Contacts
Activities
Opportunities
Orders
Contracts
Invoices
Enrich data
ACCOUNT INFO
WEB
MAINTENANCE
CONTACTS AND STRUCTURE
CONNECTED TO
HISTORY
ATTACHMENTS
Name*
bpm'online
Type
Partner
Owner
Supervisor
Web
http://www.bpmonline.com
Primary phone
+44 20 3384 0040
Category
Industry
IT companies
PRIMARY CONTACT
bpm'online
BPM'ONLINE NAMED
A LEADER
IN THE FORRESTER WAVE™: CRM SUITES
FOR MIDSIZE ORGANIZATIONS, Q4 2016
Learn more

Platform description

- **System Settings**
- **Working with data structure**
- **User interface**
- **Controls**
- **Dashboard widgets**
- **Integration**
- **Mobile application**
- **Self-service Portal**
- **ClientMessageBridge**
- **Sync Engine synchronization mechanism**
- **DataManager class description and use cases**
- **Feature Toggle. Mechanism of disabling functionality**
- **The MoneyUtilsMixin mixin**
- **The DecimalUtils module**
- **Basic macros in the MS Word printables**
- **Web-to-Case**
- **Development recommendations for Right-To-Left mode**

System Settings

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.

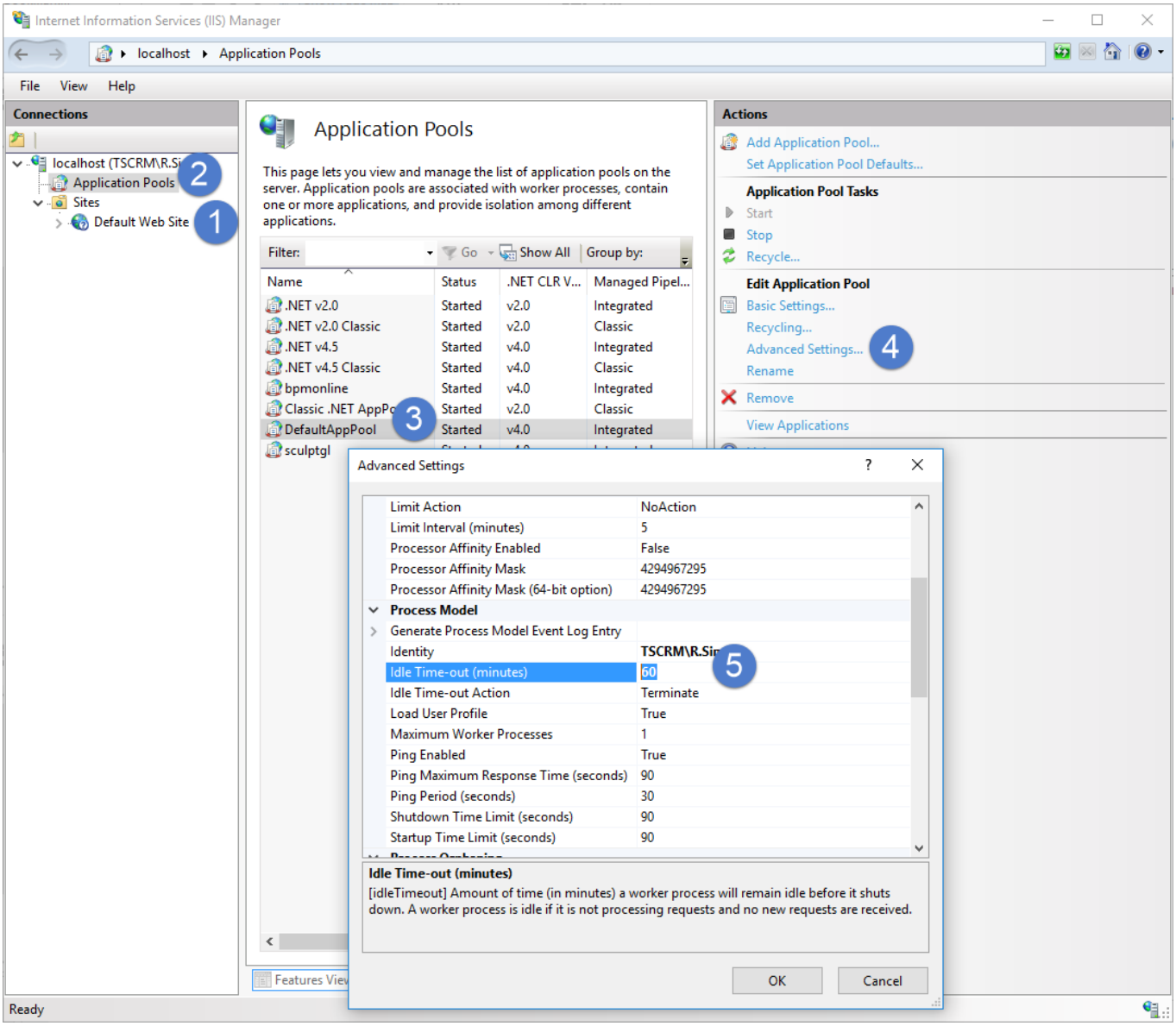
Setting user session timeouts for bpm’online 7.9.0 and earlier.

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

- **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).

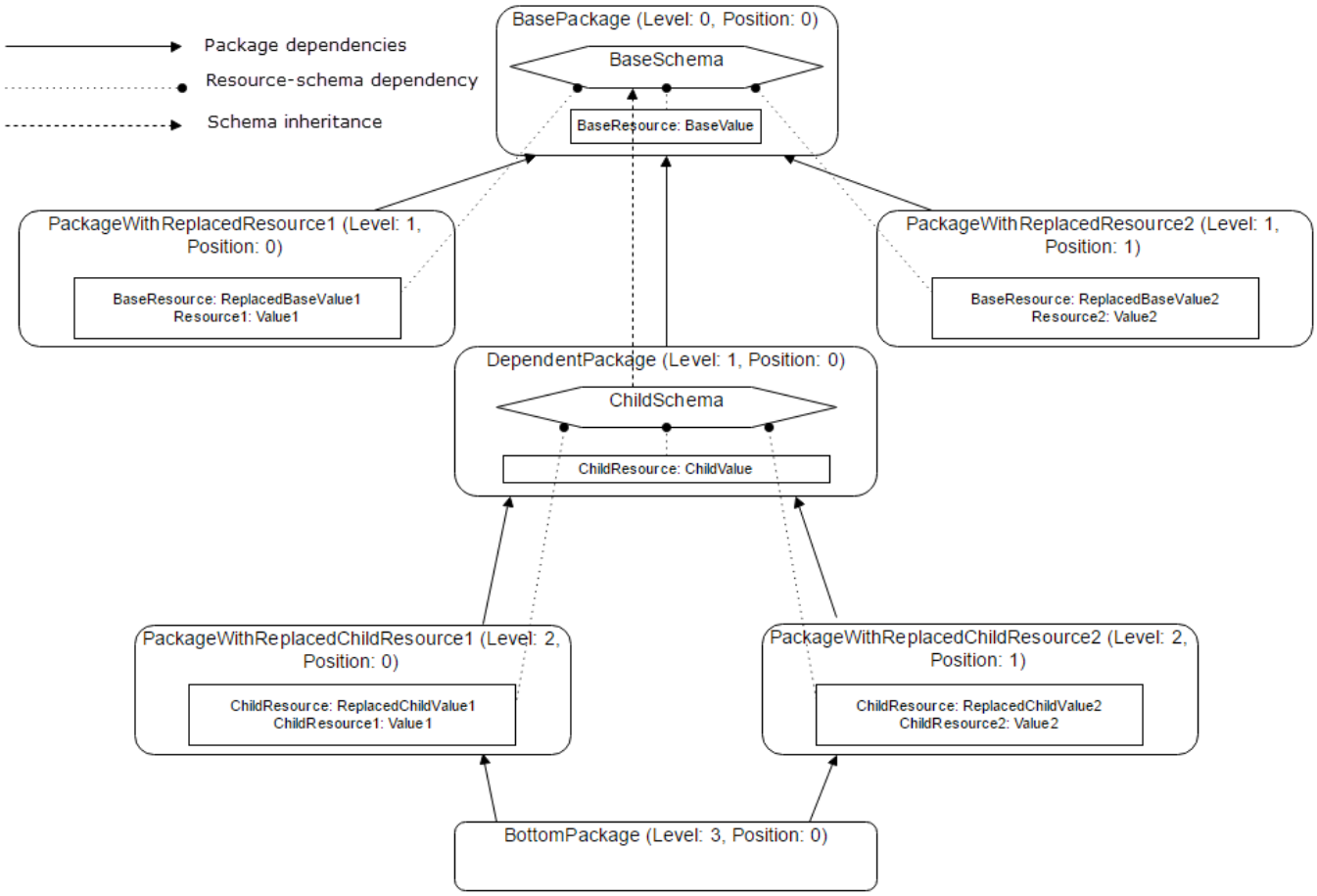
Design-time mode

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

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.

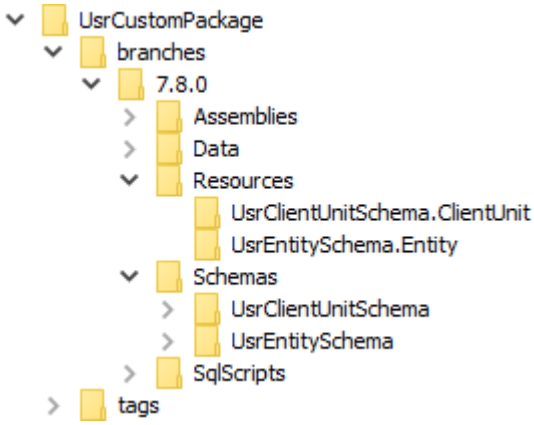
For each set of schema resources in the package-schema-culture bundle, the checksum is stored in the *SysPackageResourceChecksum* table, which allows you to quickly determine if there are any changes to the package resources when updating the package. The checksum allowed for resources to be separated from schemas, enabling users to create translation packages.

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

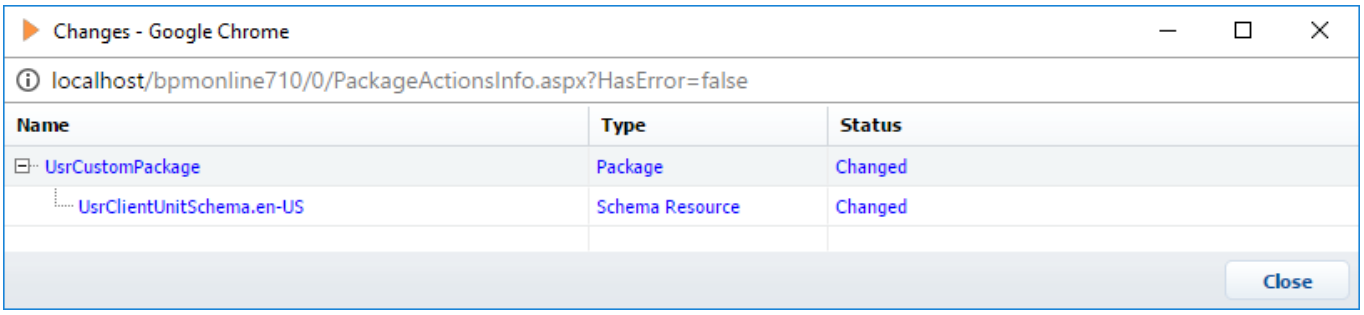
The format for storing resources in exported schemas has also changed. Now the resources in the exported schemas are stored in XML format.

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



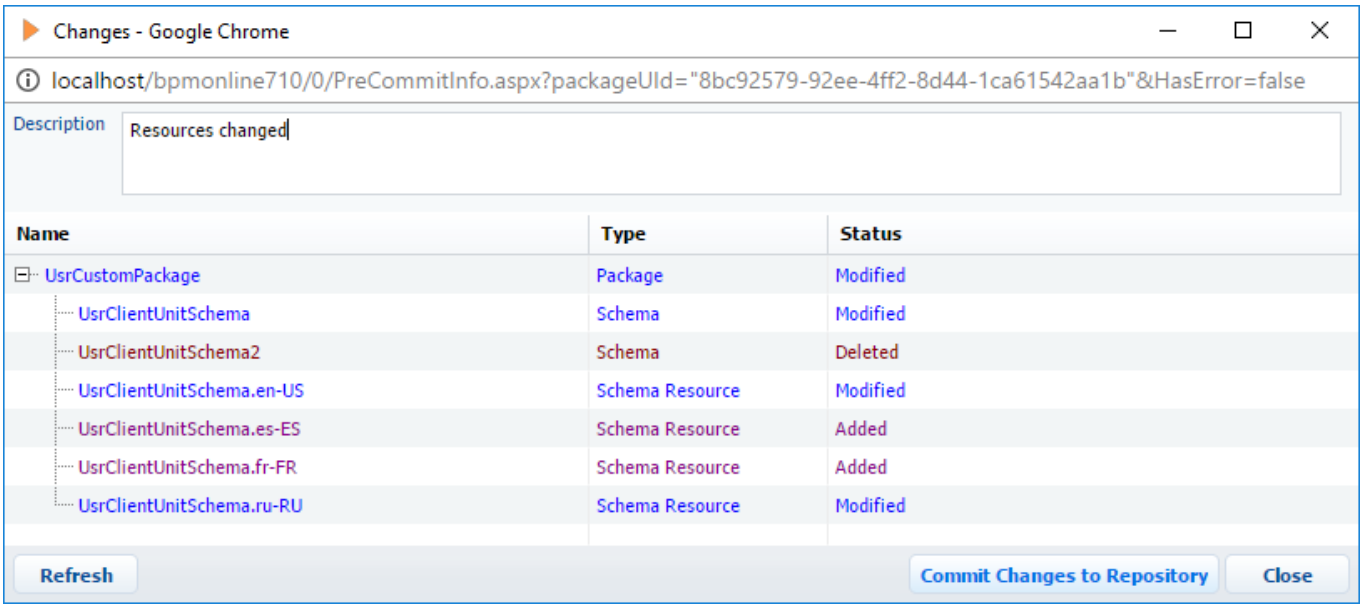
Possible resource states:

- [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).

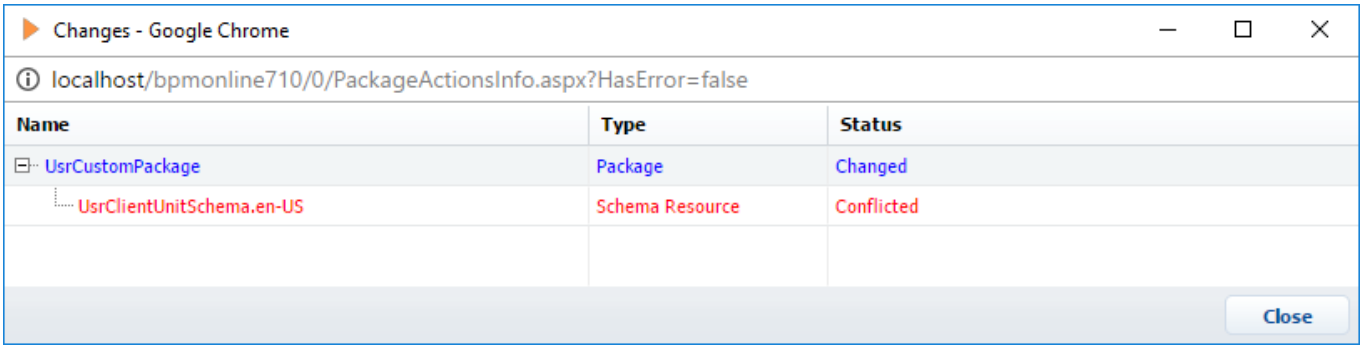
Fig. 3. Displaying resources when a package is committed



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

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.

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:

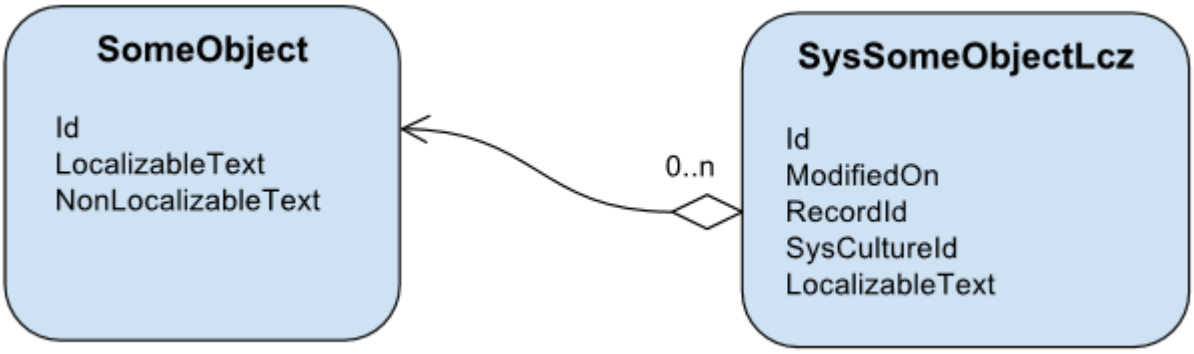
Table 1. – General localization table structure

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	

Column name	Description
LczColumn2 ... *LczColumnN*	Columns corresponding to the object's localizable columns.

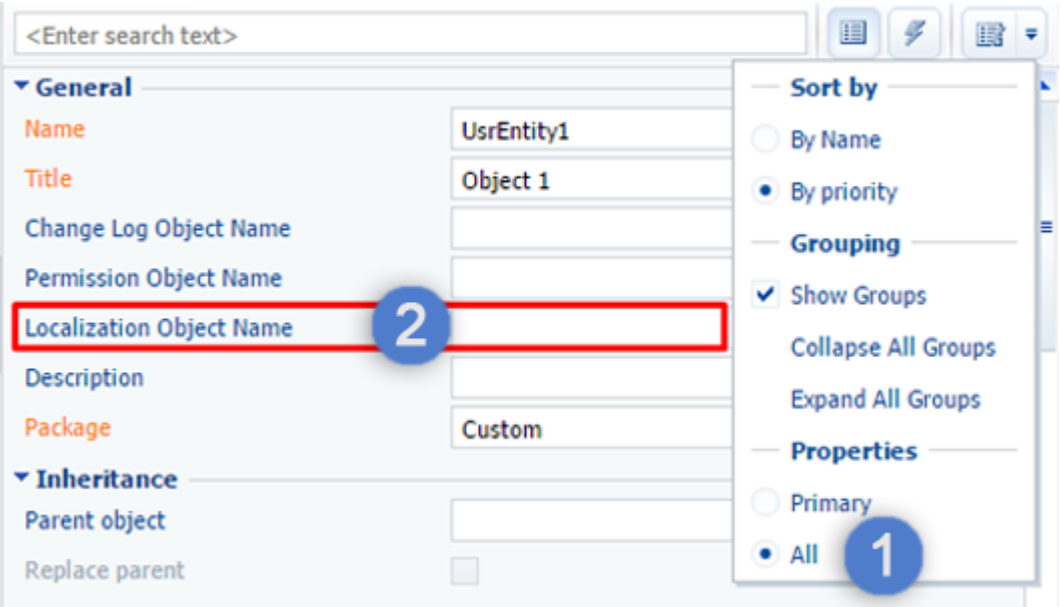
The localization table structure of the *Random* object with the *LocalizableText* column:

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).

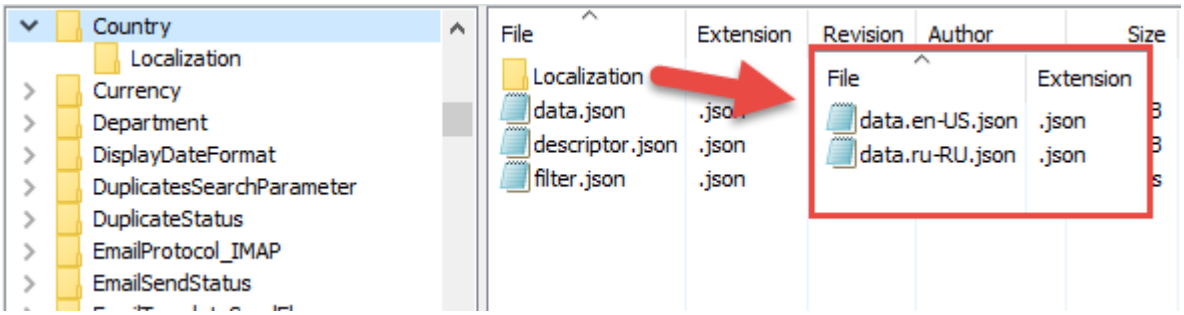
ATTENTION

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

- **AMD concept Modules**
- **Modular development principles in bpm'online**
- **Client Modules**
- **Client view model schemas**

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

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, SyncFusion 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;
});
```

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:

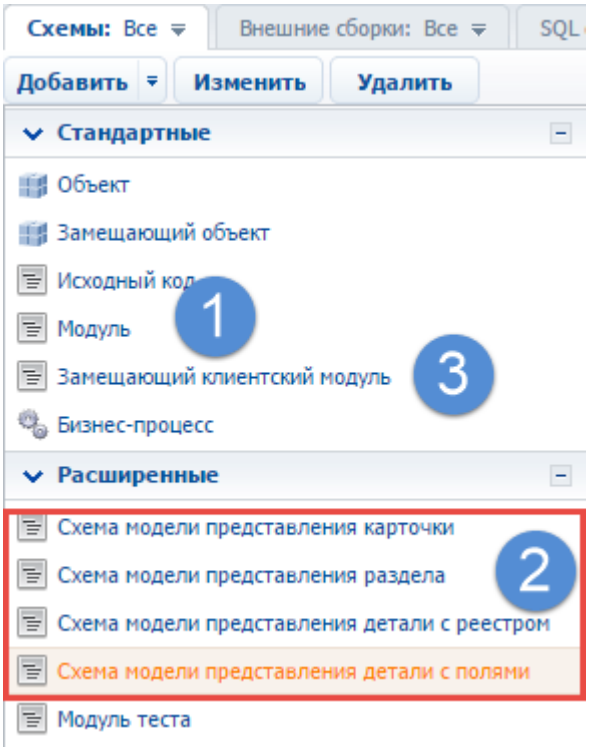
- non-visual modules (module schema)
- visual modules (view model schema)
- expanding modules and replacing client modules

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.

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 "**How to add a computed field**" 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

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

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 model. 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
  }
}
```

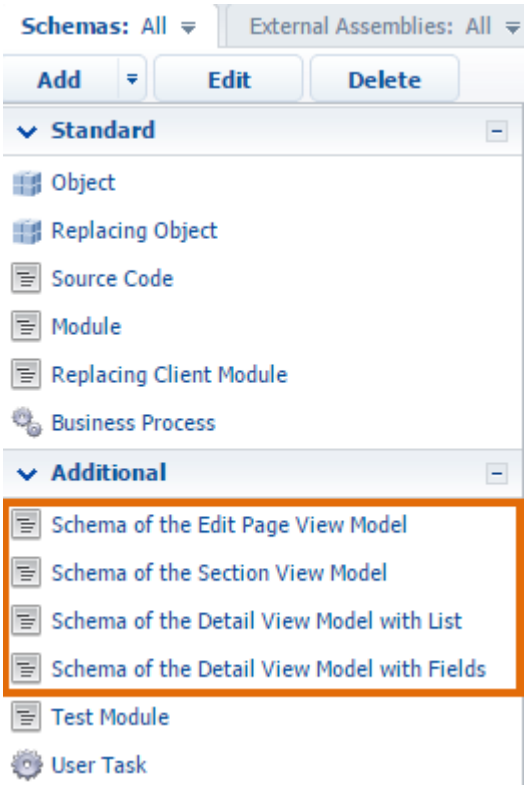
Number of operations

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:

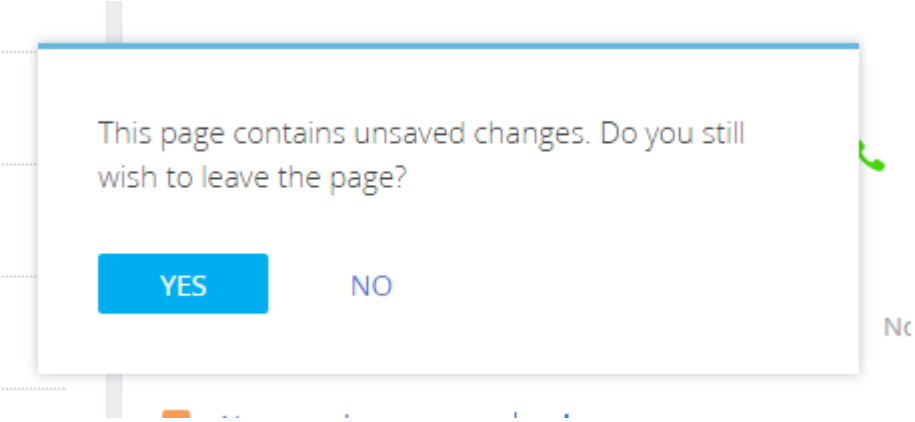
```
define("ClientSchemaName", [], function () {
    return {
        // Object schema (model).
        entitySchemaName: "EntityName",
        //...
    };
});
```

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);
                        },
                        ...
                    },
                ...
                }
            });
```

Controls

- **Controls. Introduction**
- **Details**
- **The [Connected entity profile] control**
- **SourceCodeEditMixin class description and work examples.**

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 "**('!Untranslated: JavaScript API клиентской части ядра платформы' in the on-line 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

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 "**('!Untranslated: JavaScript API клиентской части ядра платформы' 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 "**('!Untranslated: JavaScript API клиентской части ядра платформы' 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 "**('!Untranslated: JavaScript API клиентской части ядра платформы' 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 "**('!Untranslated: JavaScript API клиентской части ядра платформы' 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	The override of the *BaseDetailV2* method.

	function. *{Object} scope* – the context of the method execution.	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 that contains the properties of the detail.	The override of the *BaseDetailV2* method. 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* – collection of new elements. *{Object} options* – adding parameters.	Adds a collection of new elements to the list 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. *{Function} callback* – callback function. *{Object} scope* – the callback function context.	Saves the record.
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

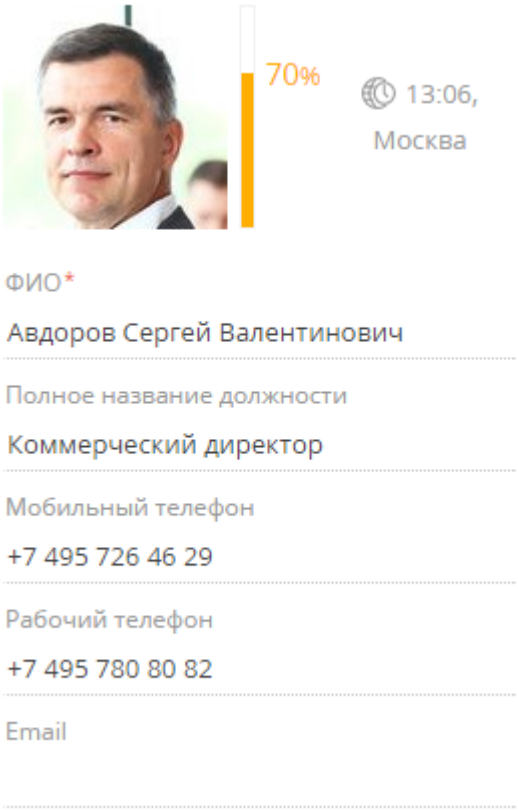
General Information

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

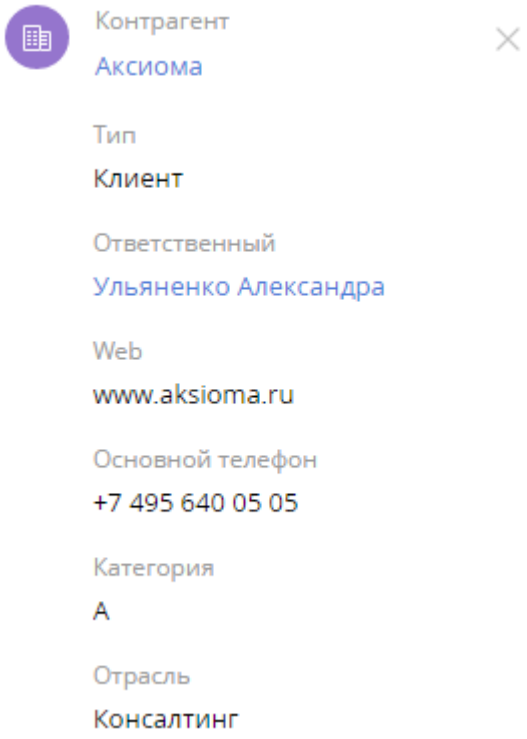
(communication via the [Account] column of the [Contact] object, Fig. 2). Learn more about record profiles and connected records in the "Record pages".

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.

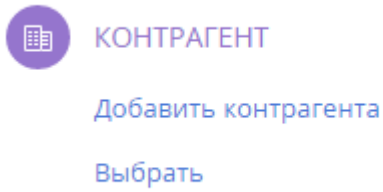
A view is described by the *diff* property (similar to the editing page description process). While embedding a module to the editing page, specify the parameter *masterColumnName* in the module attributes. The parameter stores the name of the column used to connect the profile to the main editing page 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.

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);
            }
        });
    });
```

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.

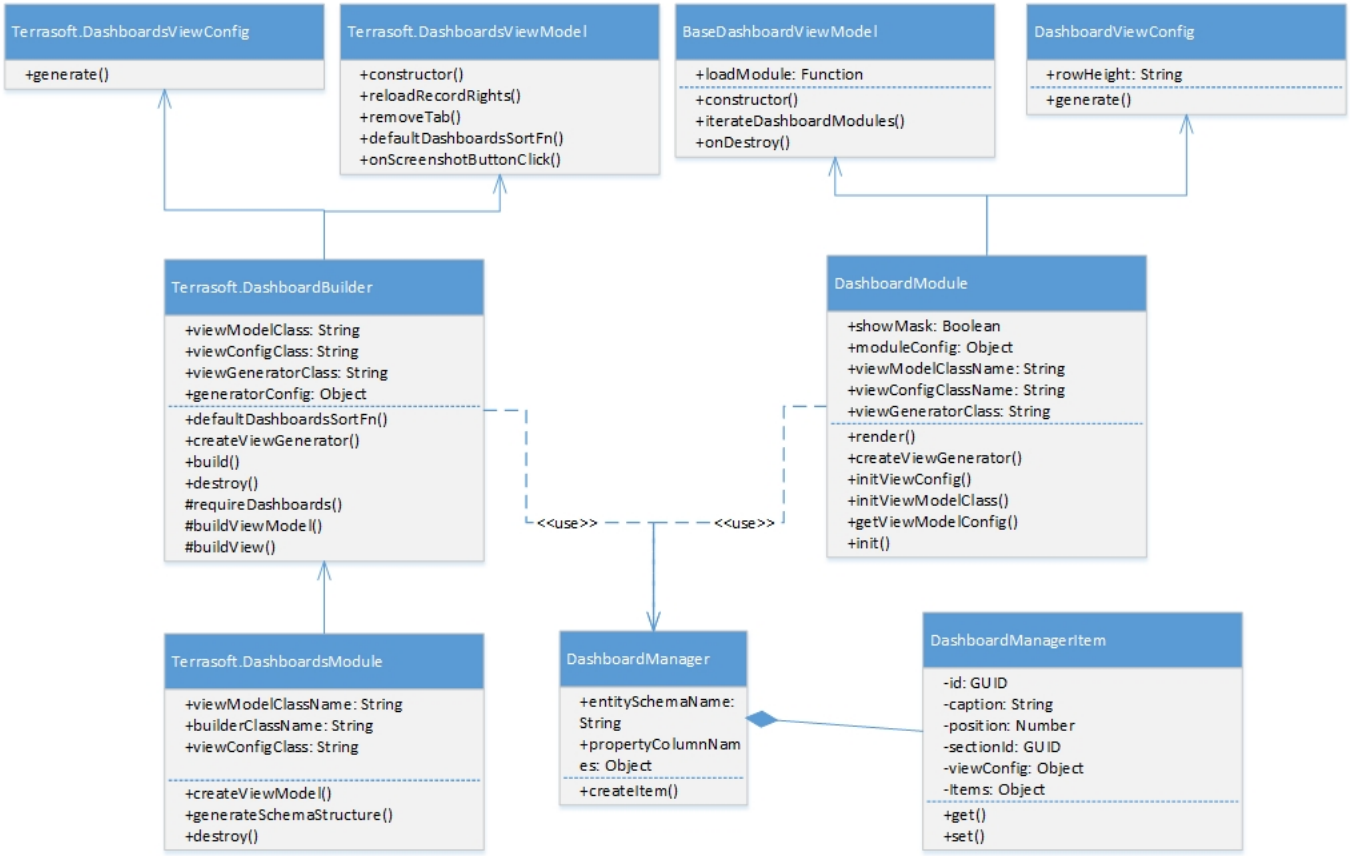
			layout: {
			columns: 4,
			rows: 4,
			colspan: 4,
			rowspan: 4
			}
			},
			{...}]

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.

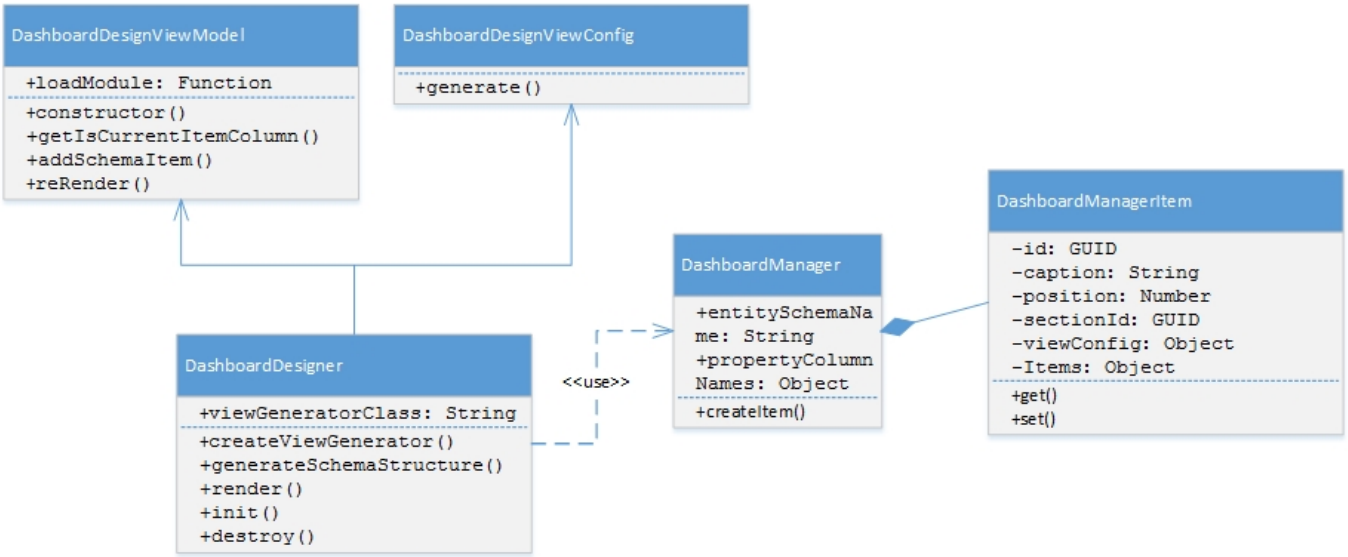
The *DashboardBuilder* module:

- *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:

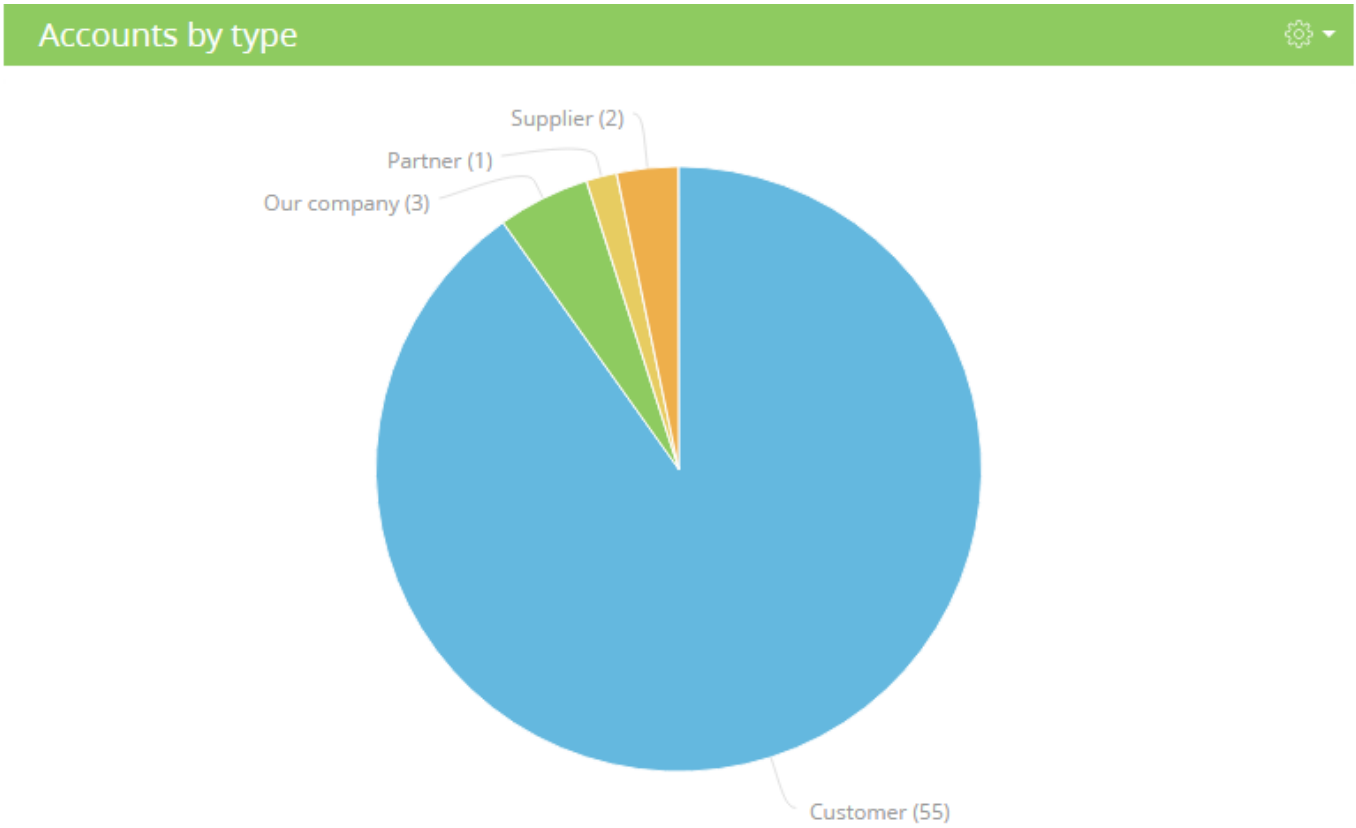
- *moduleName* – widget module name.
- *ConfigurationMessage* – the name of the module settings receiving message.
- *ResultMessage* – the name of the message that returns widget designer module settings.
- *StateConfig (stateObj)* – widget designer schema name.

Charts

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.

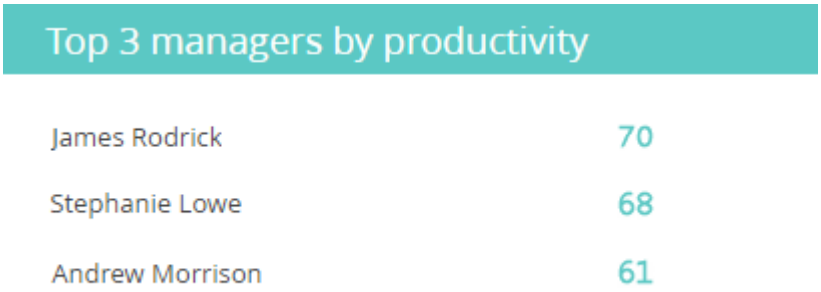
Lists

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.

Integration

- **Phone integration**
- **Email integration**

Phone integration

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.

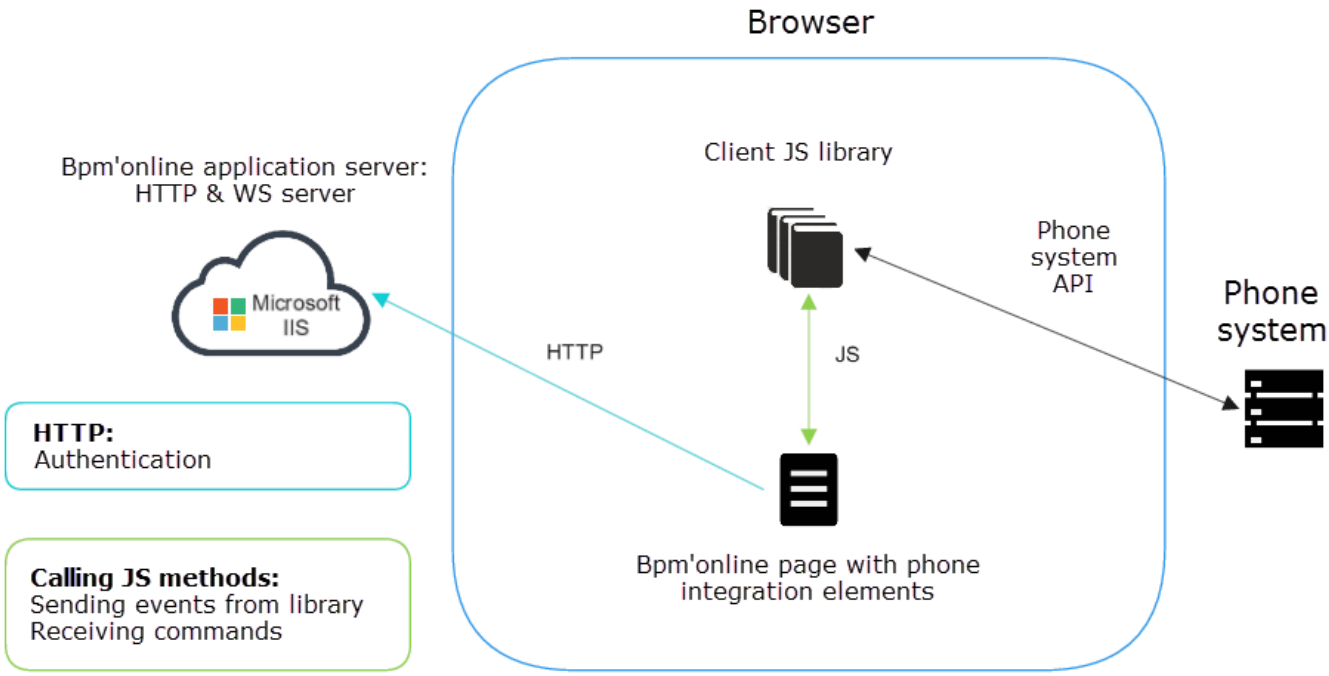
In a *first party* integration each user has a separate integration connection. Phone system events are handled as part of that connection.

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).

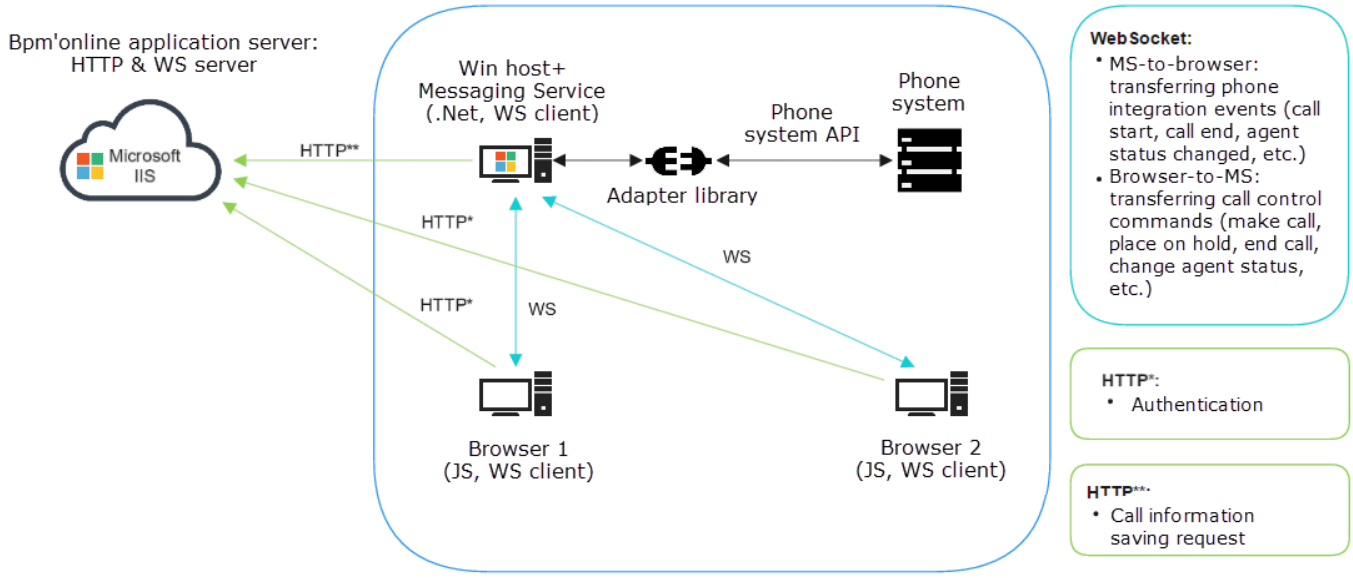
For incoming calls the phone server passes the new call start event and call parameters through WebSocket to the client integration library. When receiving a new call command, the library generates the *RingStarted* event that is passed to the application page.

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).

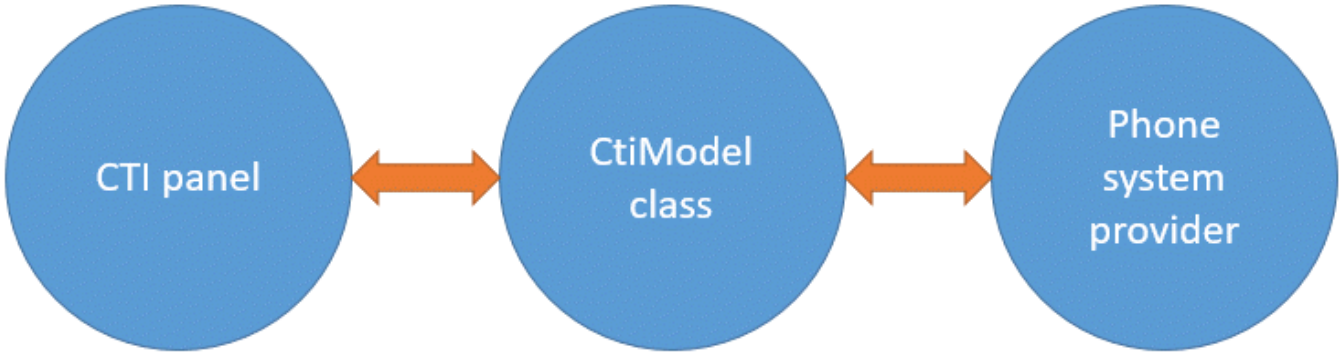
For incoming calls the phone system passes the new call start event and call parameters through the adapter library. When receiving a new call command, the Messaging Service generates the *RingStarted* event that is passed to the client.

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.

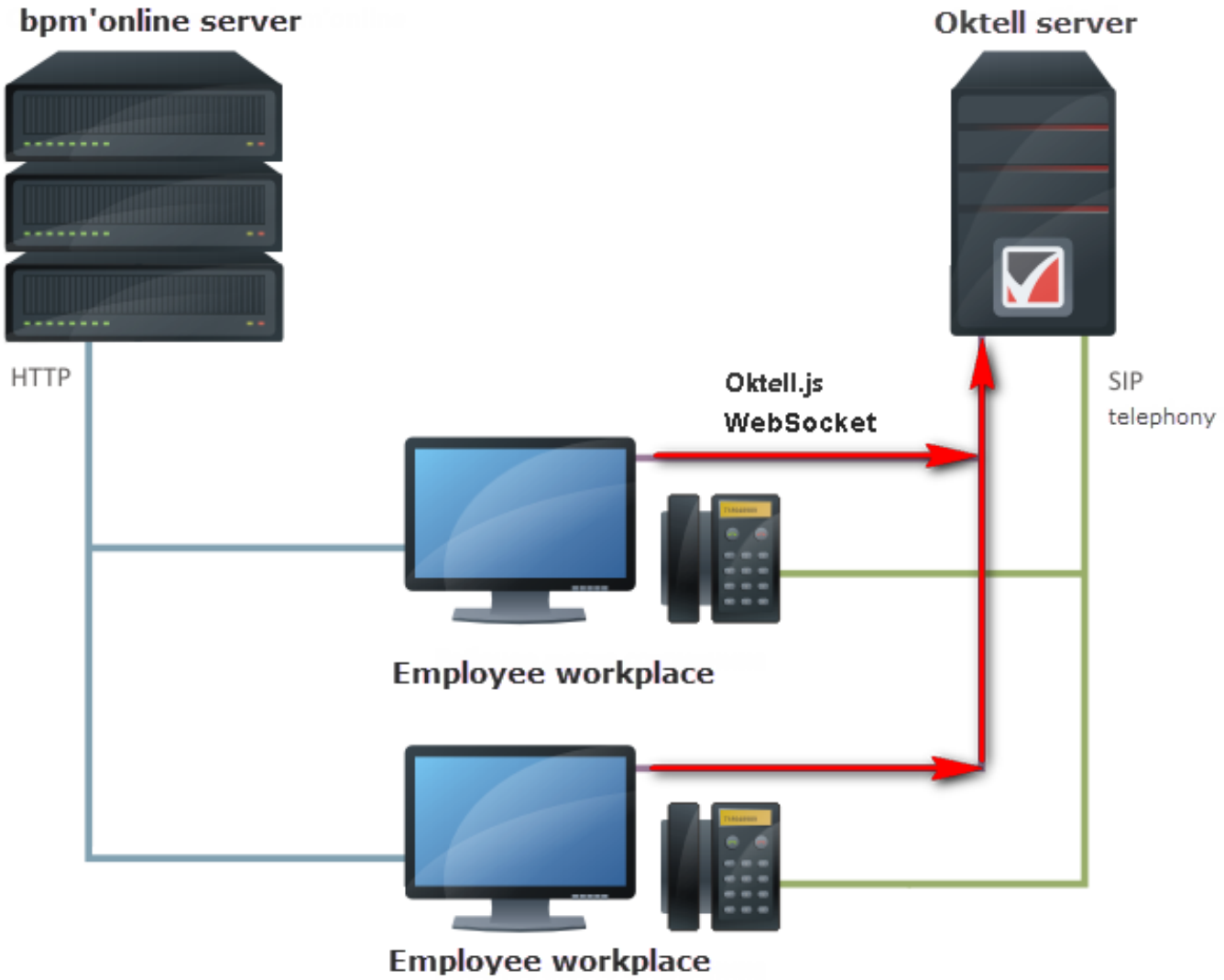
Oktell

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.

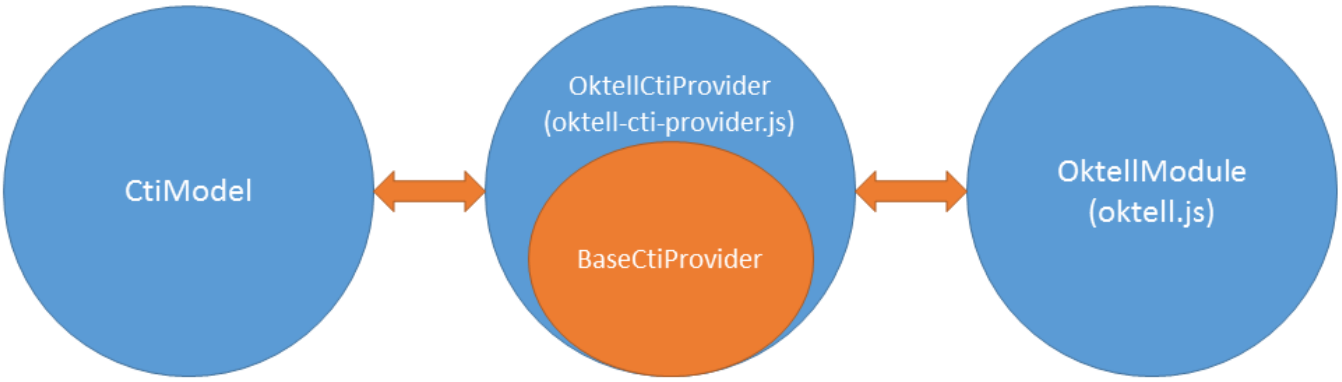
Voice transmission between subscribers

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

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.

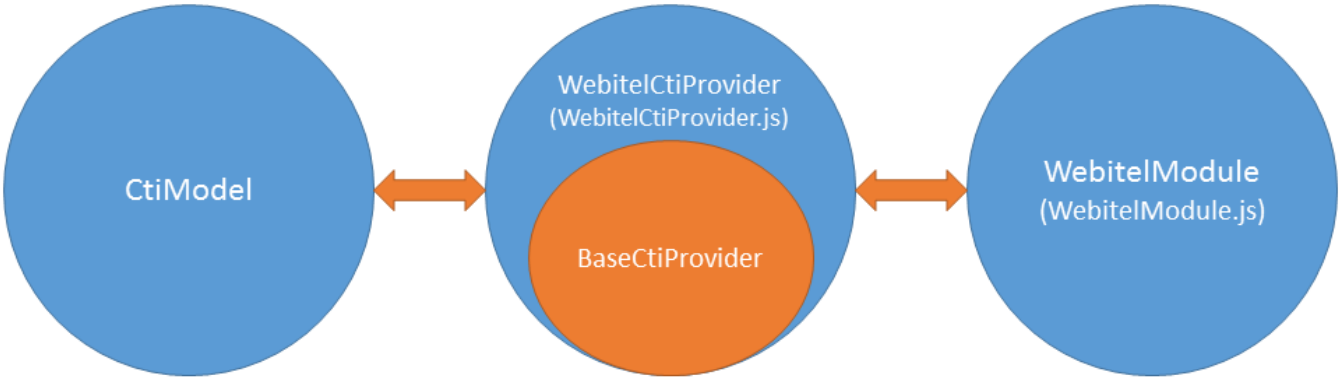
The WebitelCollaborations package implements basic interfaces for working with Webitel in bpm'online. The package contains the *WebitelCtiProvider* module, the *WebitelCtiProvider* class, Webitel connector, the connection parameters settings page, the lookup to edit Webitel users directly in bpm'online.

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

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

- **Working with email threads**

Working with email threads

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
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-	Re: relation	1C6E18E3-48B1-495E-8EF9-	F05B529D-C98C-4E26-BE00-	E1A0120E-B7C0-4261-9DE0-C63341BF1E0B

ActivityId	Title	CaseId	EmailMessageId	EmailMessageParentId
90E0376C3B5D		ACA35DB9DE0B	21F8721AEF58	
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.

Mobile application

- **Mobile app architecture**

- **Mobile application manifest**
- **Mobile application background update**
- **Resolving synchronization conflicts automatically**
- **Batch mode export**
- **Getting the settings and data from the [Dashboards] section**

Mobile app architecture

Introduction

There are three approaches to the implementation of mobile applications:

Native mobile application – an application initially developed for a specific mobile platform (iOS, Android, Windows Phone). Such applications are developed using high-level programming languages and compiled in a so-called "native OS code" that ensures the best performance. The main disadvantage of the native mobile applications is that they are not cross-platform.

Mobile web-application – a website adapted to specific mobile device. Web-applications are cross-platform, but they require constant Internet connection, since they are not physically located on the mobile device.

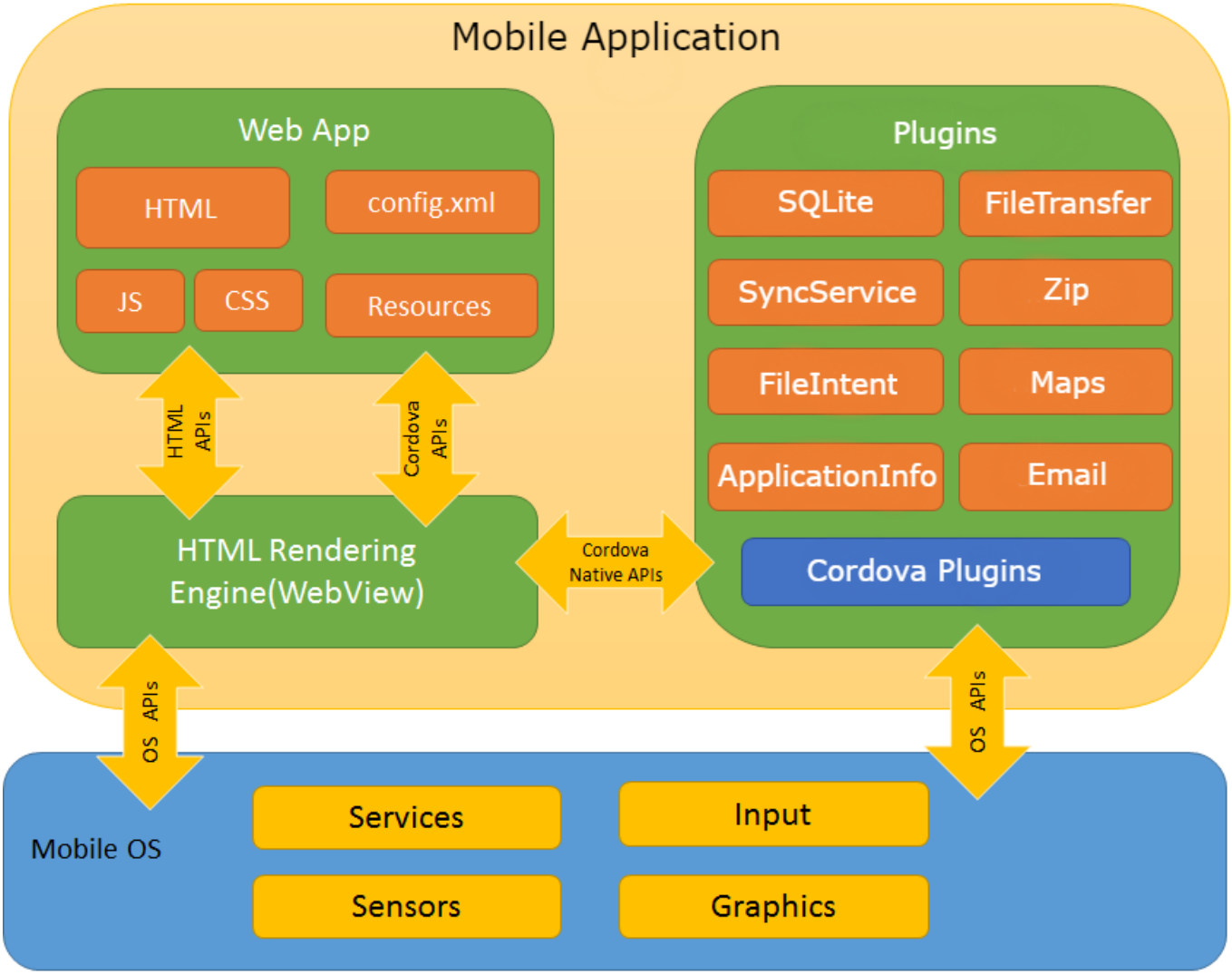
Hybrid application – a web application "encased" in a native shell. Hybrid applications are installed from the online shop (just like the native ones) and have access to the same functions of the mobile device, but are developed using HTML5, CSS and JavaScript. Unlike the native applications, hybrid applications can migrate between different platforms, although their performance is not as good as that of the native applications. Bpm'online mobile app is a hybrid application.

General information about the mobile application setup and synchronization, as well as online and offline operation mode specifics are available in the bpm'online user guide.

Mobile application architecture

The generalized representation of the mobile application architecture is available on Fig. 1.

Fig. 1 Mobile application architecture



The mobile application uses the capabilities of the Cordova framework to create hybrid applications that are treated as native on a mobile device. The Cordova framework provides access to the mobile application API for interacting with the database or hardware, such as cameras and memory cards. Cordova also provides so-called native plug-ins for working with the APIs of different mobile platforms (iOS, Android, Windows Phone, etc.). Additionally, developing custom plug-ins enables adding new functions and expanding the API. The list of available platforms and the functions of the base native Cordova plug-ins is available here.

The Sencha Touch framework functions are used for implementation of the mobile app core, visual components and configuration logic. This framework is designed specifically for mobile web applications. Sencha Touch is an analogue of the Ext JS library that contains a number of GUI components, optimized for tactile controls. Sencha Touch corresponds to the MVC template, which separates data, logic and views.

The mobile application core is a unified interface for interaction between all other client components of the application. The core uses Javascript files that can be divided into the following categories:

1. Base:

- MVC components (page views, controllers, models)
- Synchronization modules (data import/export, metadata import, file import, etc.)
- web service client classes
- classes that provide access to native plugins.

The base scripts are located in the application assembly, published in the app store.

2. Configuration:

- manifest
- section setup schemas

The application receives the configuration files during synchronization with bpm'online server and saves them locally in the device's file system.

Bpm'online mobile app operation

A bpm'online application in the app store is a set of modules required for synchronization with server (bpm'online server used by the "desktop" application). The desktop applications contain all settings and data needed for bpm'online mobile app setup. The following diagram provides an outlay of the bpm'online mobile app routine (Fig. 2):

Fig. 2 Bpm'online mobile app operation

After installing the application on a mobile device and connecting to tge bpm'online server, the mobile app obtains metadata (application structure and system data) and data from the server.

Due to this operation model, a bpm'online mobile application is compatible with all existing bpm'online products. Each product, each separate bpm'online website contains its own set of mobile application settings, logic and GUI. All the mobile app user has to do is install the mobile app and connect to the needed website.

Bpm'online mobile app operation modes

The mobile app can work in two modes:

- with the main server connection (online)
- without the main server connection (offline)

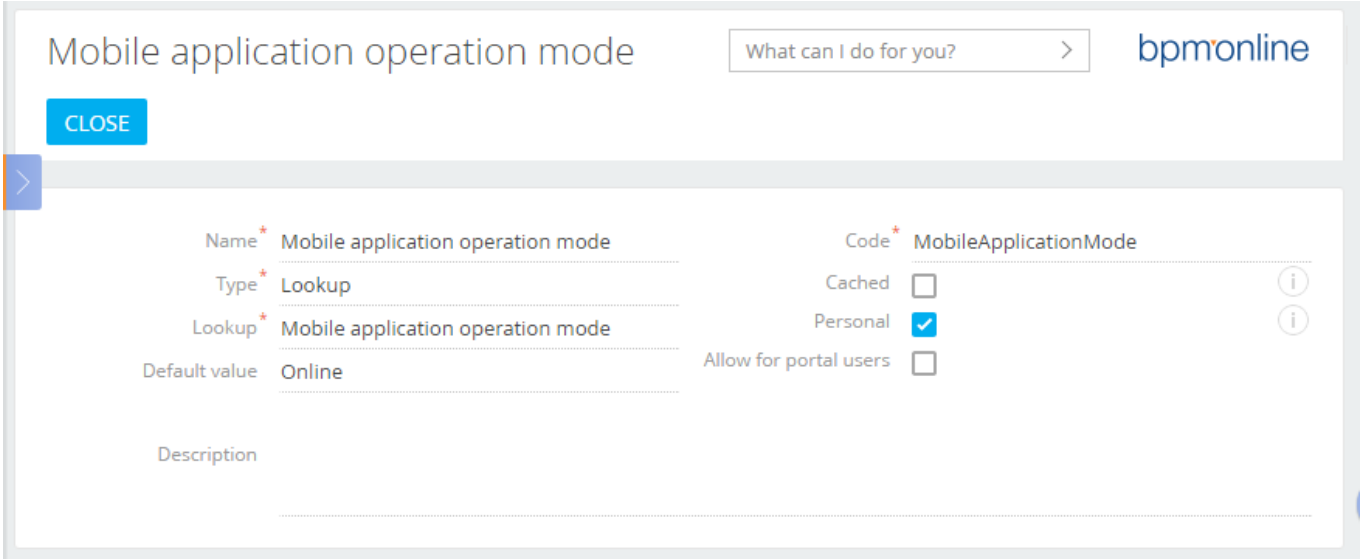
The table 1 shows the comparison between the two modes.

Table 1. Bpm'online mobile app operation modes

Online	Offline
Internet connection is required.	Internet connection is not required. Needed only for initial import and synchronization.
Users work with bpm'online server directly.	The data are saved locally, on the mobile device.
Synchronization is required only upon configuration changes (adding and deleting columns, changing business logic).	Synchronization is required to update the data and obtain configuration changes.

The mobile application operation mode is set in the [Mobile application operation mode] system setting in bpm'online. If you need to change the mobile application operation mode, edit the system setting value and clear the [Personal] checkbox. If different users must have different modes, each user must edit the system setting value with the [Personal] checkbox selected. These users must have access to edit these system settings.

Fig. 3 The [Mobile application operation mode] system setting



Synchronizing mobile application with bpm'online

In different mobile app operation modes, synchronization with bpm'online has different functions. In the online mode, the synchronization is required only to apply configuration changes. In the offline mode, the synchronization is required both to apply configuration changes and to synchronize the data between the mobile app and the bpm'online server. The general process for synchronization performed in the offline is available on Fig. 4.

Fig. 4 General procedure for synchronization in the offline mode

First, the application performs authentication. The current active server session is destroyed upon logout. The application requests data for generating the difference package from the server. The application analyzes the received data and requests updated and/or modified configuration schemas. After loading the schemas, the application obtains system data connected to the cached lookups (the so-called "simple lookups"), system settings, etc. After this, the data exchange with the server commences.

The specifics of the synchronization in the online mode is that it does not have the last two steps.

> NOTE
>
> Mobile application version 7.8.6 and up has another synchronization stage: "Data update". If this function is enabled, this stage is performed after data export and import. The data update stage compares the data available on the server with the local data and, if differences are found, loads the new data and deletes out-of-date data. This mechanism is designed to handle the situations when access permissions are changed or data has been deleted on the server. To enable this step, in the *SyncOptions* section of the manifest, edit the *ModelDataImportConfig* property for the required object-model and set the value of the *IsAdministratedByRights* property to *true*.

Mobile application manifest

General provisions

The mobile application manifest describes the structure of the mobile app, its objects and connections between them. The base version of the bpm'online mobile app is described in the manifest located in the *MobileApplicationManifestDefaultWorkspace* schema of the *Mobile* package.

In the process of the mobile app development, the users can add new sections and pages. All of them must be registered in the manifest for the application to be able to work with a new functionality. Since third-party developers have no ability to make changes to the manifest of the base app, the system automatically creates a new updated manifest each time a new section or page is added from the mobile application wizard. The manifest schema name is generated according to the following mask: MobileApplicationManifest[Workplace name]. For example, if the [Field sales] workplace is added to the mobile app, the system generates a new manifest schema with the name MobileApplicationManifestFieldForceWorkspace.

Mobile application manifest structure

The mobile application manifest is a configuration object whose properties describe the structure of the mobile app. Table 1 contains names and descriptions of the mobile application manifest.

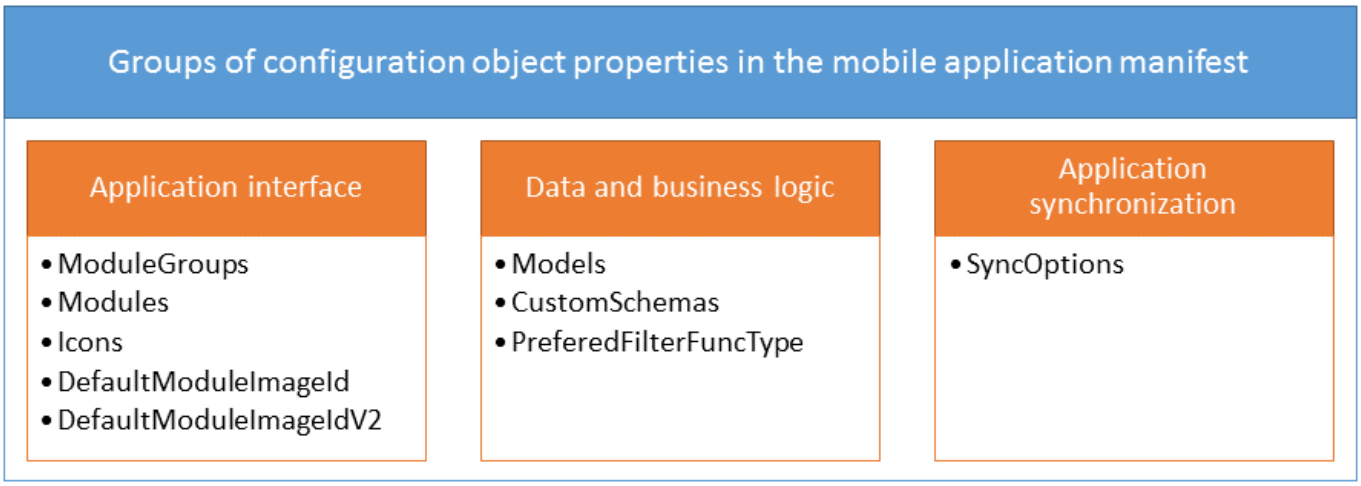
Table 1. Manifest configuration object properties

Property	Purpose
ModuleGroups	Contains upper-level settings of the main menu groups.
Modules	Describes the properties of the mobile app modules.
SyncOptions	Describes data synchronization parameters.
Models	Contains configuration of the imported application models.
PreferedFilterFuncType	Determines the operation that will be used to search and filter data.
CustomSchemas	Connects additional schemas to the mobile app.
Icons	Enables adding custom images to the app.
DefaultModuleImageId	Sets default image for UI V1.
DefaultModuleImageIdV2	Sets default image for UI V2.

All properties of a configuration object in the manifest can be split into three general groups (Fig. 1):

- *Application interface properties* contain properties that implement the mobile app interface. By using the properties in this group, the application sections and main menu are shaped and custom images are configured. For more information about this group's properties please refer to the "**Manifest. Application interface properties**" article.
- *Data and business logic properties* contain properties where imported data and custom logic is described. For more information about this group's properties please refer to the "**Manifest. Data and business logic properties**
- *Application synchronization properties* contain a single property for synchronization with the primary application. For more information about this property please refer to the "**Manifest. Application synchronization properties**" article.

Fig. 1. Groups of configuration object properties in the configuration manifest



Manifest. Application interface properties

General information

The conditional property group of the configuration object manifest contains properties that form the mobile application interface. By using the properties of this group, you can form application sections, main menus, custom images, etc. Read more about the mobile application manifest and its properties in the "**Mobile application manifest**" article.

ModuleGroups property

Application module groups. Describes the upper-level group setting of the mobile application main menu. The ModuleGroups property sets a list of named configuration objects for each menu group with the only possible Position property (see table 1).

Table 1 The configuration object property for the menu group setup.

Property	Value
Position	Group position in the main menu. Strats with 0.

Example

Setting up the mobile application menu with two groups — the main group and the [Sales] group.

```
// Mobile application module groups.
"ModuleGroups": {
    // Main menu group setup.
    "main": {
        // Group position in the main menu.
        "Position": 0
    },
    // [Sales] menu group setup.
    "sales": {
        // Group position in the main menu.
        "Position" 1
    }
}
```

Modules property

Mobile application modules. A module is an application section. Each module in the [Modules] configuration object manifest describes a configuration object with properties given in table 2. The name of the configuration section object must match the name of the model that provides section data.

Table 2 Section configuration object properties.

Property	Value
Group	The application menu group that contains the section. Set by the string containing the menu section name from the *ModuleGroups* property of the manifest configuration object.
Model	Model name that contains the section data. Set by the string containing the name of one of the models included in the *Models* property of the manifest configuration object.
Position	Section position in the main menu group. Set by a numeric value starting with 0.
isStartPage	Checkbox that defines whether a page of the current section is an application starting page (*true* — starting page, otherwise — *false*). Optional property. By default — *false*.
Title	Section title. String with the section title localized value name. Section title localized value name should be added to the *[LocalizableStrings]* manifest schema block.
Icon	This property designed to import custom images to the version 1 user interface menu section.
IconV2	This property designed to import custom images to the version 2 user interface menu section.
Hidden	Checkbox that defines a section is displayed in the menu (*true* — hidden, *false* — displayed). Optional property. By default — *false*.

Example

Set up the application sections:

1. Main menu sections: [Contacts], [Accounts].
2. The application starting page: the [Contacts] section.

Strings containing the section titles should be created in the [LocalizableStrings] manifest schema block:

- ContactSectionTitle with the "Contacts" value.
- AccountSectionTitle with the "Accounts" value.

```
// Mobile application modules.
"Modules": {
    // "Contact" section.
    "Contact": {
        // The application menu group that contains the section.
        "Group": "main",
        // Model name that contains the section data.
        "Model": "Contact",
        // Section position in the main menu group.
        "Position": 0,
        // Application starting page checkbox.
        "isStartPage": true,
        // Section title.
        "Title": "ContactSectionTitle",
        // Custom image import to section.
        "Icon": {
            // Unique image ID.
            "ImageId": "4c1944db-e686-4a45-8262-df0c7d080658"
        },
        // Custom image import to section.
        "IconV2": {
            // Unique image ID.
            "ImageId": "9672301c-e937-4f01-9b0a-0d17e7a2855c"
        },
        // Menu display checkbox.
        "Hidden": false
    },
```

```
    // "Account" section.
    "Account": {
        // The application menu group that contains the section.
        "Group": "main",
        // Model name that contains the section data.
        "Model": "Account",
        // Section position in the main menu group.
        "Position": 1,
        // Application starting page checkbox.
        "isStartPage": false,
        // Section title.
        "Title": "AccountSectionTitle",
        // Custom image import to section.
        "Icon": {
            // Unique image ID.
            "ImageId": "c046aa1a-d618-4a65-a226-d53968d9cb3d"
        },
        // Custom image import to section.
        "IconV2": {
            // Unique image ID.
            "ImageId": "876320ef-c6ac-44ff-9415-953de17225e0"
        },
        // Menu display checkbox.
        "Hidden": false
    }
}
```

Icons property

This property is designed to import custom images to the mobile application.

It is set by the configuration objects array, each containing properties from the table 3.

Table 3 The configuration object properties for the custom image import.

Property	Value
ImageListId	Image list ID.
ImageId	Custom image ID from the *ImageListID* list.

Example

```
// Custom images import.
"Icons": [
    {
        // Image list ID.
        "ImageListId": "69c7829d-37c2-449b-a24b-bcd7bf38a8be",
        // Imported image ID.
        "ImageId": "4c1944db-e686-4a45-8262-df0c7d080658"
    }
]
```

DefaultModuleImageId and DefaultModuleImageIdV2 properties

Properties are designed to set unique default image IDs for newly created sections or sections that don't contain IDs of the images in the *Icon* or *IconV2* properties of the *Modules* property of the configuration object manifest.

Example

```
// Custom interface V1 default image ID.
"DefaultModuleImageId": "423d3be8-de6b-4f15-a81b-ed454b6d03e3",
// Custom interface V2 default image ID.
"DefaultModuleImageIdV2": "1c92d522-965f-43e0-97ab-2a7b101c03d4"
```

Manifest. Data and business logic properties

General provisions

The groupof properties of a configuration object in the mobile app manifest. contains properties that describe imported data, as well as custom business logic for processing data in the mobile app. For more information about the mobile application manifest and all its properties, please refer to the "**Mobile application manifest**" article.

The Models property

The Models property contains imported application models. Each model in a property is described by a configuration object with a corresponding name. The model configuration object properties are listed in table 1.

Table 1. Model configuration object properties

Property	Value
Grid	Model list page schema name. The page will be generated automatically with the following name: Mobile[Model_name][Page_type]Page. Optional.
Preview	Preview page schema name for model element. The page will be generated automatically with the following name: Mobile[Model_name][Page_type]Page. Optional.
Edit	Edit page schema name for model element. The page will be generated automatically with the following name: Mobile[Model_name][Page_type]Page. Optional.
RequiredModels	Names of the models that the current model depends on. Optional property. All models, whose columns are added to the current model, as well as columns for which the current model has external keys.
ModelExtensions	Model extensions. Optional property. An array of schemas, where additional model settings are implemented (adding business rules, events, default values, etc.).
PagesExtensions	Model page extensions. Optional property. An array of schemas where additional settings for various page types are implemented (adding details, setting titles, etc.).

Case example

Add the following model configurations to the manifest:

1. Contact. Specify list page, view and edit page schema names, required models, model extension modules and model pages.
2. Contact address. Specify only the model extension module.

The *Models* property of a manifest configuration item must look like this:

```
// Importing models.
"Models": {
    // "Contact" model.
    "Contact": {
        // List page schema.
        "Grid": "MobileContactGridPage",
        // Display page schema.
        "Preview": "MobileContactPreviewPage",
        // Edit page schema.
        "Edit": "MobileContactEditPage",
        // The names of the models the "Contact" model depends on.
        "RequiredModels": [
            "Account", "Contact", "ContactCommunication", "CommunicationType",
"Department",
            "ContactAddress", "AddressType", "Country", "Region", "City",
"ContactAnniversary",
            "AnniversaryType", "Activity", "SysImage", "FileType",
"ActivityPriority",
```

```
            "ActivityType", "ActivityCategory", "ActivityStatus"
        ],
        // Model extensions..
        "ModelExtensions": [
            "MobileContactModelConfig"
        ],
        // Model page extensions.
        "PagesExtensions": [
            "MobileContactRecordPageSettingsDefaultWorkplace",
            "MobileContactGridPageSettingsDefaultWorkplace",
            "MobileContactActionsSettingsDefaultWorkplace",
            "MobileContactModuleConfig"
        ]
    },
    // "Contact addresses" model.
    "ContactAddress": {
        // List, display and edit pages were generated automatically.
        // Model extensions..
        "ModelExtensions": [
            "MobileContactAddressModelConfig"
        ]
    }
}
```

The PreferedFilterFuncType property

The property defines the operation used for searching and filtering data in the section, detail and lookup lists. The value for the property is specified in the a an enumeration Terrasoft.FilterFunctions enumeration. The list of filtering functions is available in table 2.

Table 2. Filtering functions (Terrasoft.FilterFunctions)

Function	Value
SubStringOf	Determines whether a string passed as an argument, is a substring of the *property* string.
ToUpper	Returns values of the column specified in the *property* in relation to upper list.
EndsWith	Verifies if the *property* column value ends with a value passed as argument.
StartsWith	Verifies if the *property* column value starts with a value passed as argument.
Year	Returns year based on the *property* column value.
Month	Returns month based on the property column value.
Day	Returns day based on the property column value.
In	Checks if the property column value is within the value range passed as the function argument.
NotIn	Checks in the property column value is outside the value range passed as the function argument.
Like	Determines if the *property* column value matches the specified template.

If the current property is not explicitly initialized on the manifest, then by default the *Terrasoft.FilterFunctions.StartWith* function is used for search and filtering, as this ensures the proper indexes are used in the SQLite database tables.

Case example

Use the substring search function for data search.

The PreferedFilterFuncType property of the configuration object in the manifest must look like this:

```
// Substring search function is used to search for data.
"PreferedFilterFuncType": "Terrasoft.FilterFunctions.SubStringOf"
```

> ⚠ *ATTENTION*
>
> If the function specified as the data filtering function in the PreferedFilterFuncType section is not *Terrasoft.FilterFunctions.StartWith*, then *indexes will not be used while searching* database records.

The CustomSchemas property

The CustomSchemas property is designed for connecting additional schemas to the mobile app (custom schemas with source code in JavaScript) that expand the functionality. This can be additional classes implemented by developers as part of a project, or utility classes that implement functions to simplify development, etc.

The value of the property is an array with the names of connected custom schemas.

Case example

Connect additional custom schemas for registering actions and utilities.

```
//
"CustomSchemas": [
    // Custom action registration schema.
    "MobileActionCheckIn",
    // Custom utility schema.
    "CustomMobileUtilities"
]
```

Manifest. Application synchronization properties

General information

The conditional property group of the manifest configuration object contains a single property used to synchronize data with the main application. Read more about the mobile application manifest and its properties in the "**Mobile application manifest**" article.

SyncOptions Property

Describes the options for configuring data synchronization. Contains the configuration object with properties presented in table 1.

Table 1 The configuration object properties for the synchronization setup.

Property	Value
ImportPageSize	The number of pages imported in the same thread.
PagesInImportTransaction	The number of import threads.
SysSettingsImportConfig	Imported system settings array.
SysLookupsImportConfig	Imported system lookups array.
ModelDataImportConfig	An array of models that will load the data during synchronization.

In the *ModelDataImportConfig* model array, you can specify additional synchronization parameters, the list of available columns and filter conditions for each model. If you need to load a full model during synchronization, specify the object with the model name in the array. If the model needs to apply additional conditions for synchronization, the configuration object with properties given in table 2 is added to the *ModelDataImportConfig* array.

Table 2 The configuration object properties for the synchronization model setup.

Property	Value
Name	Model name (see *Models* property of the manifest configuration object).

SyncColumns	The column models array for which data is imported. In addition to the listed columns, the system columns (CreatedOn, CreatedBy, ModifiedOn, ModifiedBy) and primary displayed columns will also be imported during synchronization.
SyncFilter	The filter applied to the model during import

The *SyncFilter* is applied to the model during import is a configuration object with properties given in table 3.

Table 3 Filter model configuration object properties.

Property	Value
type	Filter type. Set by the enumeration value *Terrasoft.FilterTypes*. Optional property. By default *Terrasoft.FilterTypes.Simple*. Filter types (*Terrasoft.FilterTypes*): *Simple* — Filter with one condition. *Group* — Group filter with multiple conditions.
logicalOperation	The logical operation for combining a collection of filters (for filters with *Terrasoft.FilterTypes.Group* type). Set by the enumeration value *Terrasoft.FilterLogicalOperations*. By default - *Terrasoft.FilterLogicalOperations.And*. Logical operation types (*Terrasoft.FilterLogicalOperations*): *Or* — Logical operation OR. *And* — Logical operation AND.
subfilters	A collection of filters applied to a model. Obligatory property for the filter type *Terrasoft.FilterTypes.Group*. The filters are interconnected by the logical operation set in the *logicalOperation* property. Each filter is a configuration filter object.
property	Filtered column model name. Obligatory property for the filter type *Terrasoft.FilterTypes.Simple*.
valueIsMacrosType	The checkbox that defines whether the filtered value is a macro. Optional property can be: *true* if the filter uses a macro, and *false* if it doesn't.
value	Value of the column filtration set in the *property* property. Obligatory property for the filter type *Terrasoft.FilterTypes.Simple*. Can be set directly by the filter value (including *null*) or a macro (the valueIsMacrosType property must be set to *true*). Macros that can be used as the property value are contained in the *Terrasoft.ValueMacros* enumeration. Value macros (*Terrasoft.ValueMacros*): *CurrentUserContactId* — Current user ID. *CurrentDate* — Current date. *CurrentDateTime* — Current date and time. *CurrentDateEnd* — Current date end. *CurrentUserContactName* — Current contact name. *CurrentUserContact* — Current contact name and ID. *SysSettings* — System setting value. The system setting name is included in the *macrosParams* property. *CurrentTime* — Current time. *CurrentUserAccount* — Current account name and ID. *GenerateUId* — Generated ID.
macrosParams	Values transitioned to macros as parameters. Optional property. This property is now used only for the *Terrasoft.ValueMacros.SysSettings* macro.

isNot	Applied to the negation operator filter. Optional property. Takes the *true* value if the the negation operator is applied to the filter, otherwise — *false*.
funcType	Function type applied to the model column set in the *property* property. Optional property. Takes values from the *Terrasoft.FilterFunctions* enumeration. Argument values for the filtration functions are set in the *funcArgs* property. The value to compare the result of the function is specified by the *value* property. Filtration functions (*Terrasoft.FilterFunctions*): *SubStringOf* — Determines whether the string passed in as an argument is a substring of the *property* column. *ToUpper* — Changes the column value set in the *property* to uppercase. *EndsWith* — Checks whether the value in the *property* column ends with the value set as an argument. *StartsWith* — Checks if the value of the *property* column starts with the value set as an argument. *Year* — Returns the year value according to the *property* column. *Month* — Returns the month value according to the *property* column. *Day* — Returns the day value according to the *property* column. *In* — Checks the occurrence of the value of the column property in the range of values that is passed as argument to the function. *NotIn* — Checks for the absence of the value of the column property in the range of values that is passed as an argument to the function. *Like* — Determines whether the value of the column property with the specified template.
funcArgs	An array of argument values for the function filter defined in the *funcType* property. The order of the values in the array *funcArgs* must match the order of parameters of the *funcType* function.
name	The name of a filter or group of filters. Optional property.
modelName	Filtered model name. Optional property Specifies whether the filtering is performed by the columns of the connected model.
assocProperty	Connected model column by which the main model is connected. The primary column serves as a connecting column of the main model.
operation	Filtration operation type. Optional parameter. Takes values from the *Terrasoft.FilterOperation* enumeration. By default — *Terrasoft.FilterOperation.General*. Filtration operations (*Terrasoft.FilterOperation*): *General* — Standard filtration. *Any* — Filtration by the *exists* filter.
compareType	Filter comparison operation type. Optional parameter. Takes values from the *Terrasoft.ComparisonType* enumeration. By default — *Terrasoft.ComparisonType.Equal*. Comparison operations (*Terrasoft.ComparisonType*): *Equal* — Equal. *LessOrEqual* — Less or equal. *NotEqual* — Not equal. *Greater* — Greater. *GreaterOrEqual* — Greater or equal.

Less Less.

Example

During synchronization, the data for the following models has to be loaded into the mobile application:

1. Activity All columns are loaded. While the model is being filtered, only the activities with the current user listed as a participant are loaded.
2. Activity type — a full model is loaded.

The *SyncOptions* property of the manifest configuration object must look like this:

```
// Synchronization settings
"SyncOptions": {
    // The number of pages imported in the same thread.
    "ImportPageSize": 100,
    // The number of import threads.
    "PagesInImportTransaction": 5,
    // Imported system settings array.
    "SysSettingsImportConfig": [
        "SchedulerDisplayTimingStart", "PrimaryCulture", "PrimaryCurrency",
"MobileApplicationMode", "CollectMobileAppUsageStatistics",
"CanCollectMobileUsageStatistics", "MobileAppUsageStatisticsEmail",
"MobileAppUsageStatisticsStorePeriod", "MobileSectionsWithSearchOnly",
"MobileShowMenuOnApplicationStart", "MobileAppCheckUpdatePeriod",
"ShowMobileLocalNotifications", "UseMobileUIV2"
    ],
    // Imported system lookups array.
    "SysLookupsImportConfig": [
        "ActivityCategory", "ActivityPriority", "ActivityResult",
"ActivityResultCategory", "ActivityStatus", "ActivityType", "AddressType",
"AnniversaryType", "InformationSource", "MobileApplicationMode",
"OppContactInfluence", "OppContactLoyality", "OppContactRole", "OpportunityStage",
"SupplyPaymentDelay", "SupplyPaymentState", "SupplyPaymentType"],
    // An array of models that will load the data during synchronization.
    "ModelDataImportConfig": [
        // Activity model configuration.
        {
            "Name": "Activity",
            // The filter applied to the model during import
            "SyncFilter": {
                // Filtered column model name.
                "property": "Participant",
                // Filtered model name.
                "modelName": "ActivityParticipant",
                // Connected model column by which the main model is connected.
                "assocProperty": "Activity",
                // Filtration operation type.
                "operation": "Terrasoft.FilterOperations.Any",
                // A macro is used for filtration.
                "valueIsMacros": true,
                // Column filtration value — current contact ID and name.
                "value": "Terrasoft.ValueMacros.CurrentUserContact"
            },
            // The column models array for which data is imported.
            "SyncColumns": [
                "Title", "StartDate", "DueDate", "Status", "Result",
"DetailedResult", "ActivityCategory", "Priority", "Owner", "Account", "Contact",
"ShowInScheduler", "Author", "Type"
            ]
        },
        // The ActivityType model is loaded in full.
```

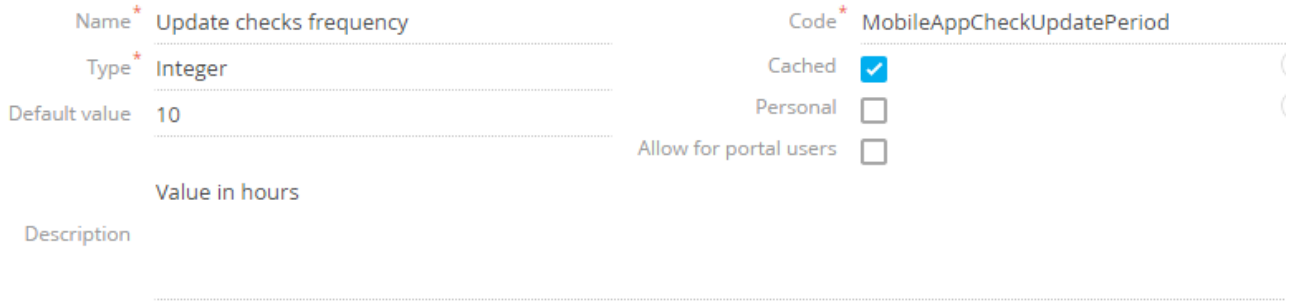
```
        {
            "Name": "ActivityType",
            "SyncColumns": []
        }
    ]
}
```

Mobile application background update

Introduction

The bpm'online mobile application implements a synchronization mechanism for the application structure, which can work automatically in the background. Use the [Update checks frequency] system setting to manage the automatic synchronization process.

Fig. 1. The [Update checks frequency] system setting



The frequency is specified as a time interval between automatic configuration updates initiated by the mobile app. The interval is specified in hours. If the setting is set to "0", the application will always download configuration updates.

Working conditions

The application starts the background synchronization only if the following conditions are met:

- the mobile device uses the iOS or Android platform
- synchronization has not started yet
- the time between synchronizations (specified in the [Update checks frequency] system setting) has elapsed
- the application is launched, or the application is activated (i.e. it was previously minimized).

If changes were made during the structure update, the application will automatically restart to apply the changes when the user minimizes it or switches to another application.

Platform specifics

1. The background mode is implemented through a parallel running service on the Android platform. This approach ensures that the running synchronization will be completed even if the user manually closes the application.
2. On the iOS platform, the application works in the main *webView* while the synchronization uses the second *webView*. This ensures that the user can continue working with the application while the structure synchronization is in progress.

   Unlike the Android platform, the synchronization can be interrupted when the application is closed manually or if the iOS platform closes the app itself.
3. On the Windows 10 platform, the application checks for updates on the server at startup. There is no background update check.

   If updates are available, a page with the relevant information is displayed.

Resolving synchronization conflicts automatically

Introduction

During the synchronization of a mobile app working in the offline mode, the transferred data sometimes cannot be saved. This happens if:

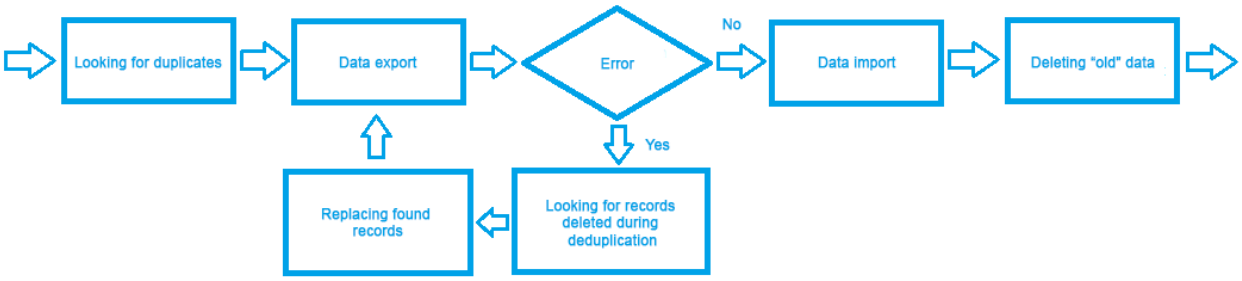
- A record was merged with a duplicate, and therefore does not exist.
- A record was deleted.

The mobile application processes both cases automatically.

Record merged with a duplicate

The algorithm of resolving synchronization conflicts caused by duplicate merging is shown in Fig. 1:

Fig. 1. Resolving a conflict of merged duplicates



The application first checks the records that have been merged since the last synchronization. Namely, what records were deleted and which records replaced them. If there were no errors during export, the import is performed. If the Foreign Key Exception or the Item Not Found Exception errors occur, the following steps are taken to resolve the conflict:

- The system checks for columns with the "old" record.
- The "old" record will be replaced with a new record which includes the merged data.

The record is sent to bpm'online afterwards. When the import is finished and the information on merged duplicates is found, the "old" records are deleted locally.

Record not found

If the server returns a "Record not found" error, the application performs the following actions:

1. The application first checks the records that have been deleted when merged with another record (see: "Record merged with a duplicate").
2. If there is no deleted record in the list, the application deletes it locally.
3. The application deletes the record information from the synchronization log.

After this, the application considers this conflict as resolved and continues to export data.

Batch mode export

By default, the mobile application sends the changes made by the users to the server one at a time, i.e. every adjustment results in at least one server request. A large number of changes may lead to a significant amount of time for processing them.

Starting with bpm'online version 7.9, it is possible to send data in batch mode, and significantly speed up the

process of sending data to the server.

To enable the batch mode, set the *UseBatchExport* property to *true* in the *SyncOptions* section of the mobile application manifest. As a result, all user changes will be grouped into several batch requests according to the operation type. Possible operation types - insert, update and delete.

Getting the settings and data from the [Dashboards] section

Introduction

Getting the settings and the dashboards data is implemented in the *AnalyticsService* service and in the *AnalyticsServiceUtils* utility in the *Platform* package.

AnalyticsService

Class that implements the *AnalyticsService* service conteins following public methods:

- *public Stream GetDashboardViewConfig(Guid id)* – returns the settings of a view and widgets on the dashboards tab by the dashboard page Id.
- *public Stream GetDashboardData(Guid id, int timeZoneOffset)* – returns the data from all widgets on the dashboards tab by the dashboard page Id.
- *public Stream GetDashboardItemData(Guid dashboardId, string itemName, int timeZoneOffset)* – returns data from athe specific widget by the dashboard page Id and the widget name.

timeZoneOffset {int} – the time zone offset (in minutes) from the UTC. Dashboards data will be received using this time zone.

An example of the requests to the AnalyticsService service

HEADERS

```
Accept:application/json
```

The GetDashboardViewConfig() method

URL

```
POST /0/rest/AnalyticsService/GetDashboardViewConfig
```

The content of the request

```
{
    "id": "a71d5c04-dff7-4892-90e5-9e7cc2246915"
}
```

The content of the response

```
{
  "items": [
    {
      "layout": {
        "column": 0,
        "row": 0,
        "colSpan": 12,
        "rowSpan": 5
      },
      "name": "Chart4",
      "itemType": 4,
      "widgetType": "Chart"
```

```
    }
  ]
}
```

The GetDashboardData() method

URL

```
POST /0/rest/AnalyticsService/GetDashboardData
```

The content of the request

```
{
    "id": "a71d5c04-dff7-4892-90e5-9e7cc2246915",
    "timeZoneOffset": 120
}
```

The content of the response

```
{
    "items": [
        {
            "name": "Indicator1",
            "caption": "Average time for activity",
            "widgetType": "Indicator",
            "style": "widget-green",
            "data": 2
        }
    ]
}
```

The GetDashboardItemData() method

URL

```
POST /0/rest/AnalyticsService/GetDashboardItemData
```

The content of the request

```
{
    "dashboardId": "a71d5c04-dff7-4892-90e5-9e7cc2246915",
    "itemName": "Chart4",
    "timeZoneOffset": 120
}
```

The content of the response

```
{
  "name": "Chart4",
  "caption": "Invoice payment dynamics",
  "widgetType": "Chart",
  "chartConfig": {
    "xAxisDefaultCaption": null,
    "yAxisDefaultCaption": null,
    "seriesConfig": [
      {
        "type": "column",
        "style": "widget-green",
        "xAxis": {
          "caption": null,
```

```
          "dateTimeFormat": "Month;Year"
        },
        "yAxis": {
          "caption": "Actually paid",
          "dataValueType": 6
        },
        "schemaName": "Invoice",
        "schemaCaption": "Invoice",
        "useEmptyValue": null
      }
    ],
    "orderDirection": "asc"
  },
  "style": "widget-green",
  "data": []
}
```

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

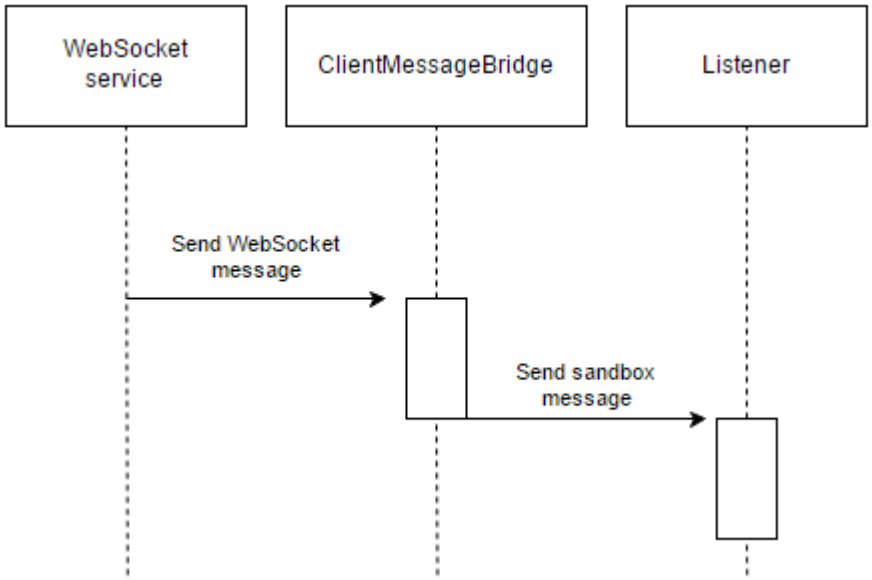
- **ClientMessageBridge . Message history save mechanism**
- **ClientMessageBridge. API description**
- **ClientMessageBridge. The client-side WebSocket message handler**

ClientMessageBridge . Message history save mechanism

General information

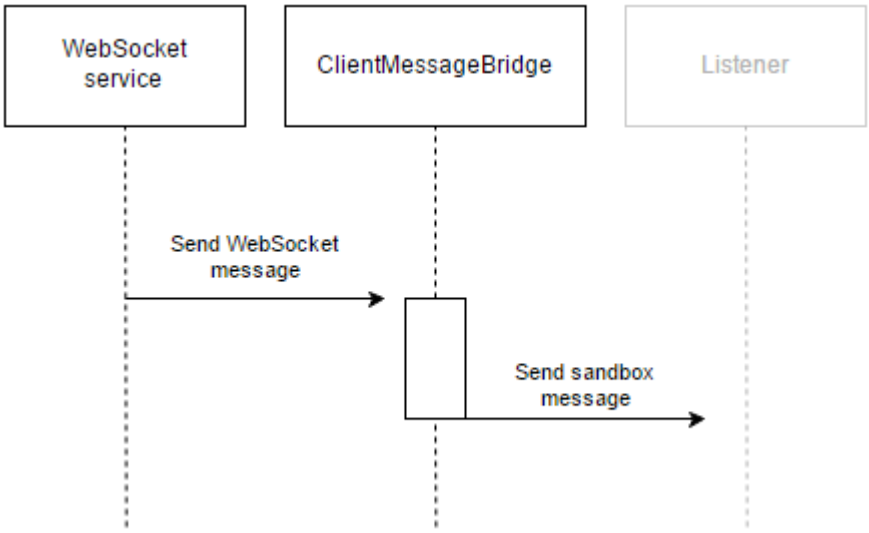
In a best-case scenario a handler is located within the system at the time of publication of the message (Fig. 1).

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

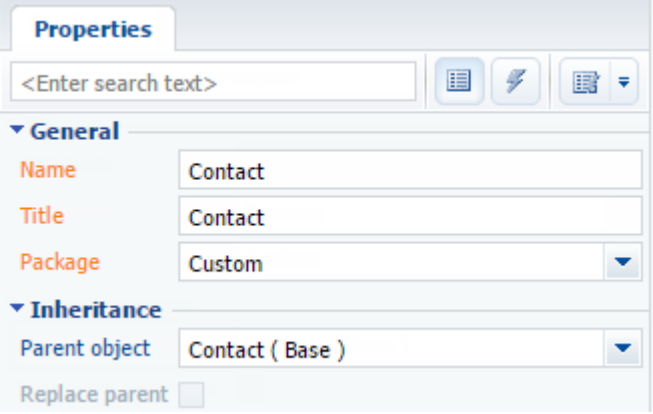
When a contact is saved, you need to publish a message with the *NewUserSet* name that contains information about the contact's birth date and name on the server side. On the client side, you must implement the NewUserSet messaging within the system. Additionally, before messaging, you must process the *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).

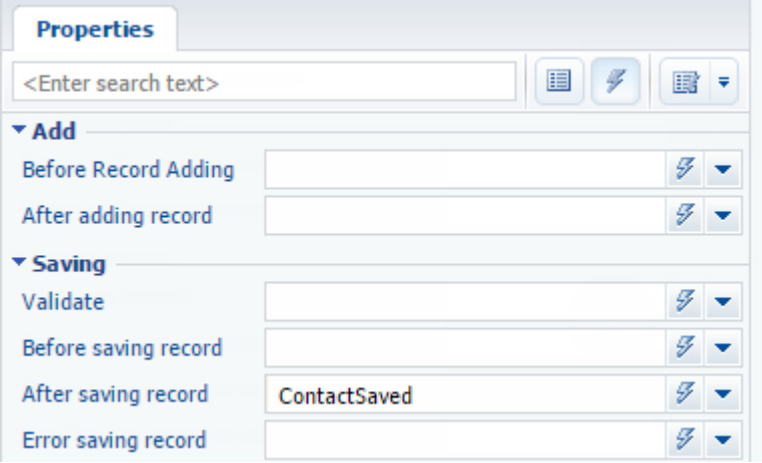
Fig. 1. Creating the replacing [Contact] object



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

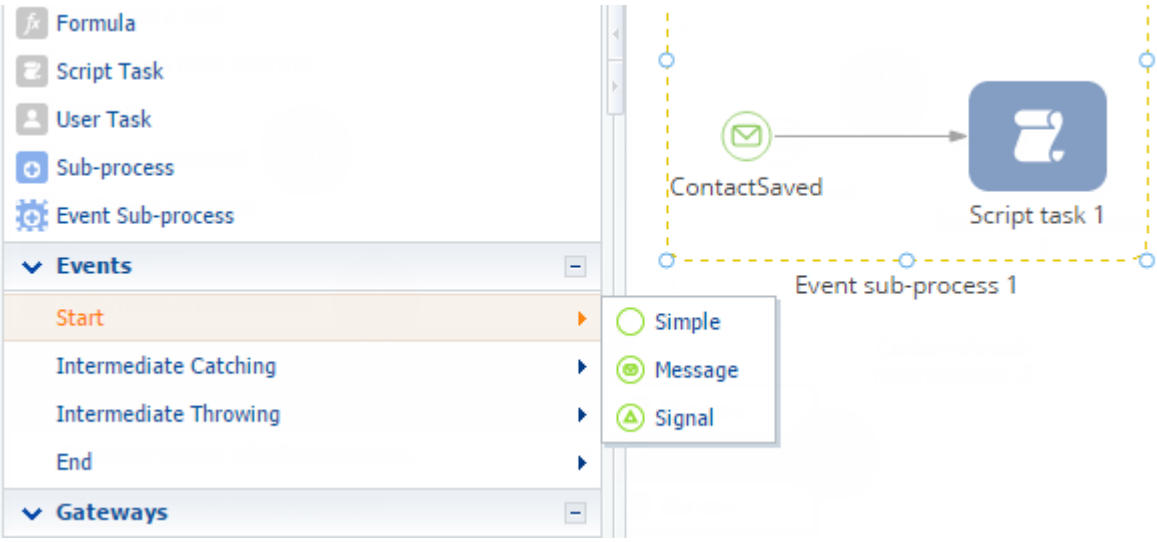
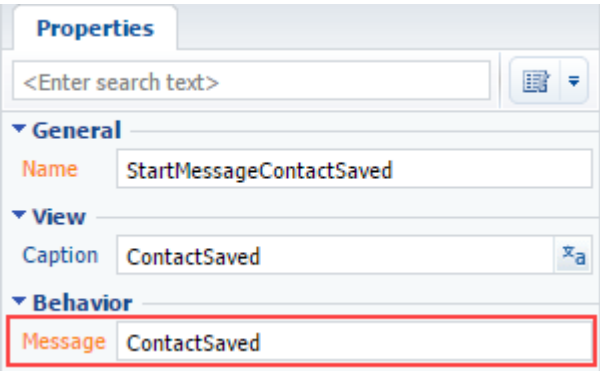


Fig. 4. Initial message properties



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

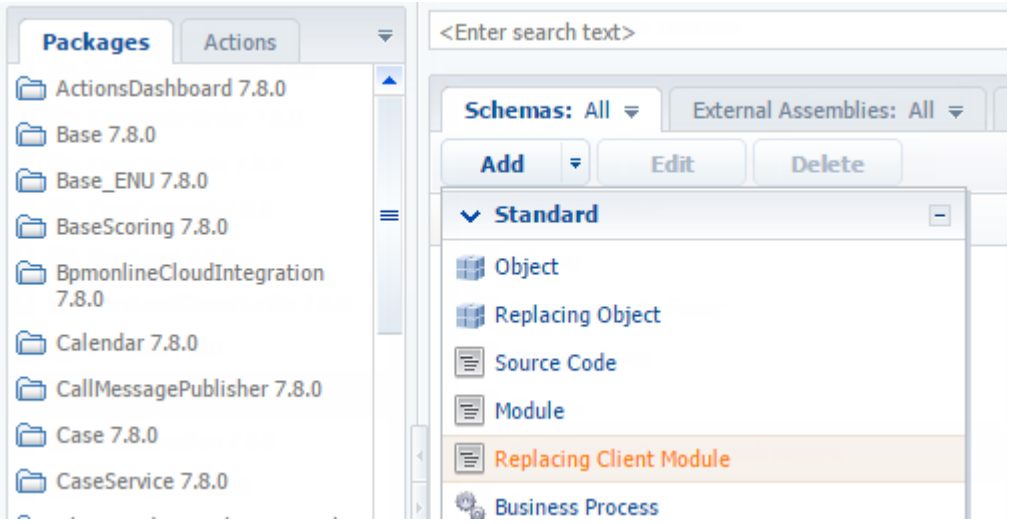
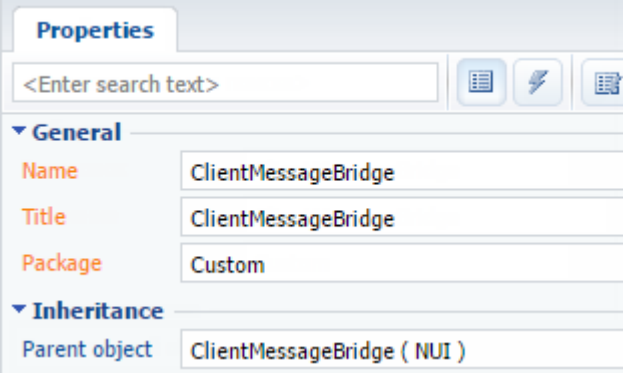


Fig. 6. Client module properties.



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

- **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, ref LocalItem localItem)	*Void*	Fills in properties of an element of external 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.

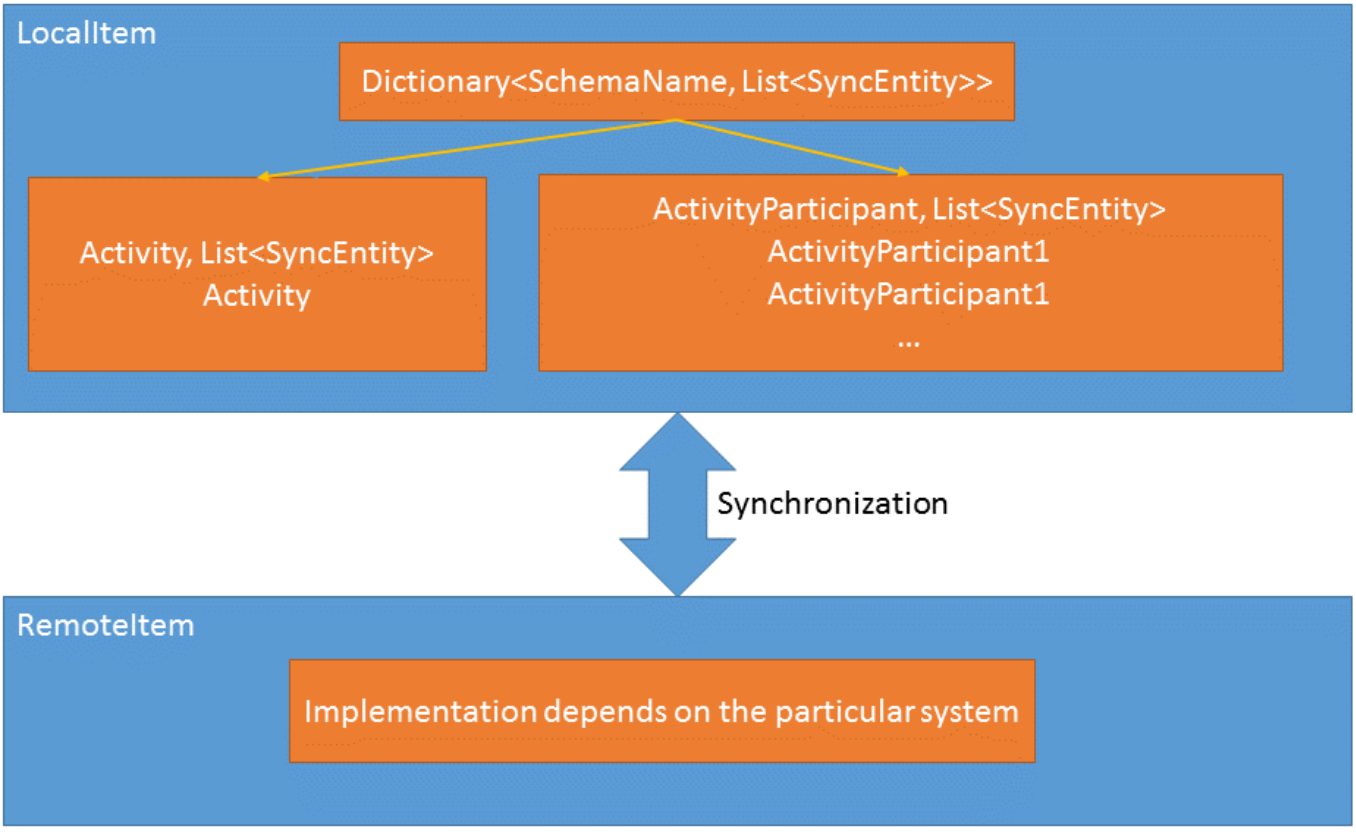
Synchronization example

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).

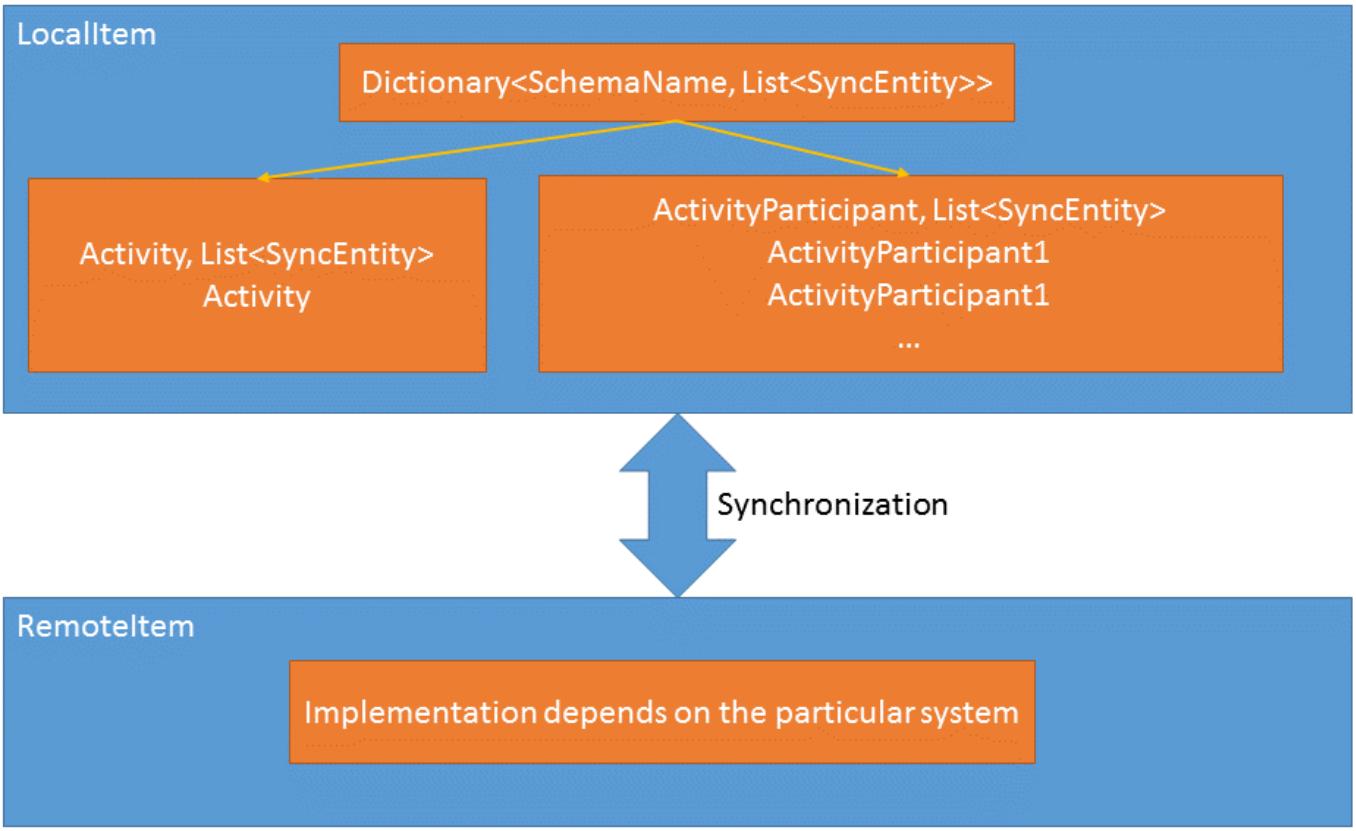
Only information about synchronized elements is stored in the metadata.

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

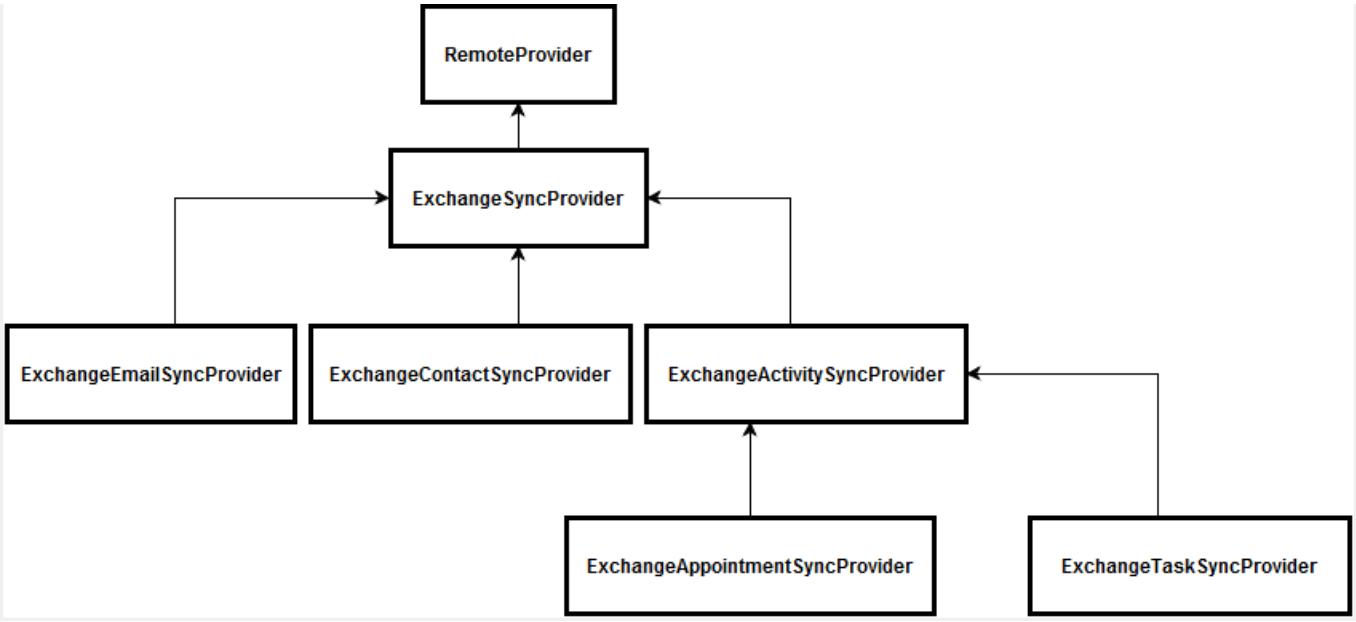
Integration with various entities of MS Exchange via EWS protocol (Exchange Web Services) is supported by the Sync Engine synchronization mechanism. This article describes synchronization of tasks between bpm'online and MS Exchange. The task synchronization algorithm is no different from that described 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

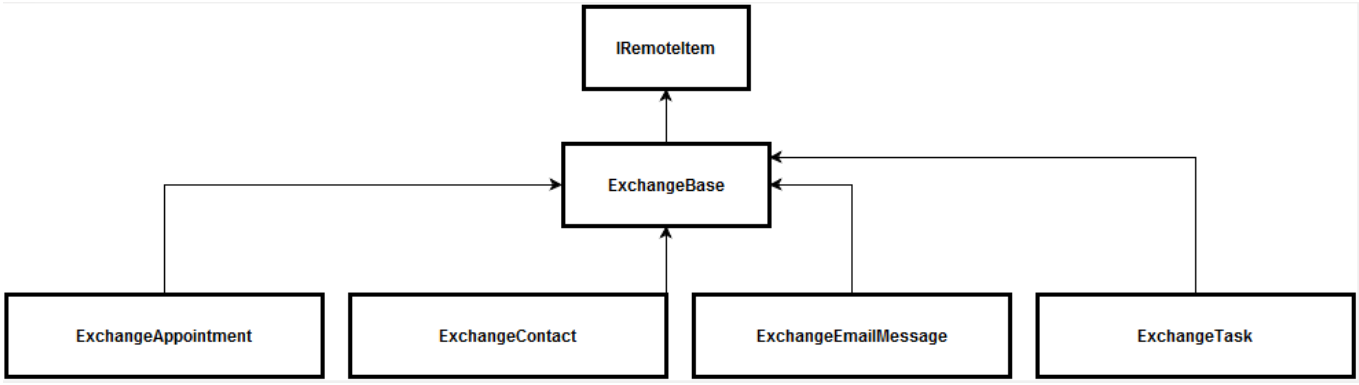
As described in the "**Bpm'online synchronization with external storages**" article, in order to implement an integration using this mechanism, you need a class that implements the logic of the external storage (an heir of the *RemoteProvider* class). The hierarchy of provider classes is shown in figure 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

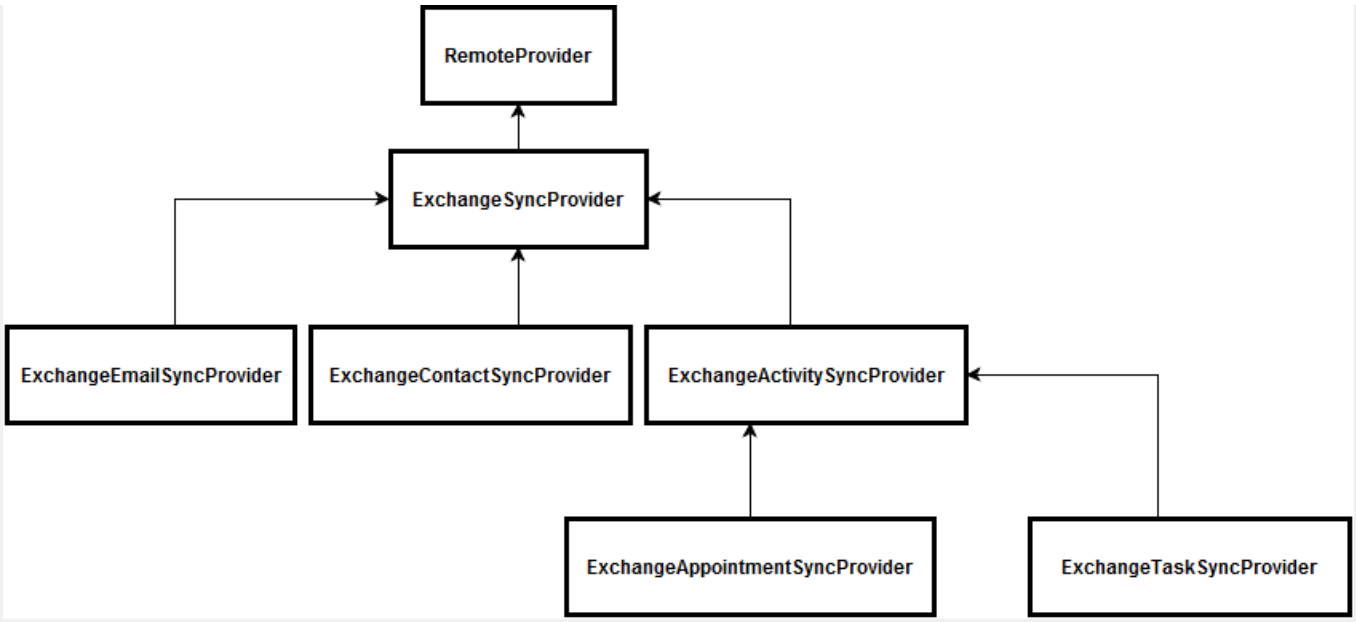
Synchronization with various services of MS Exchange via EWS protocol (Exchange Web Services) is supported by the Sync Engine synchronization mechanism. This article describes the synchronization of email in bpm'online with MS Exchange. Email in bpm'online is synchronized only from MS Exchange. Since 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



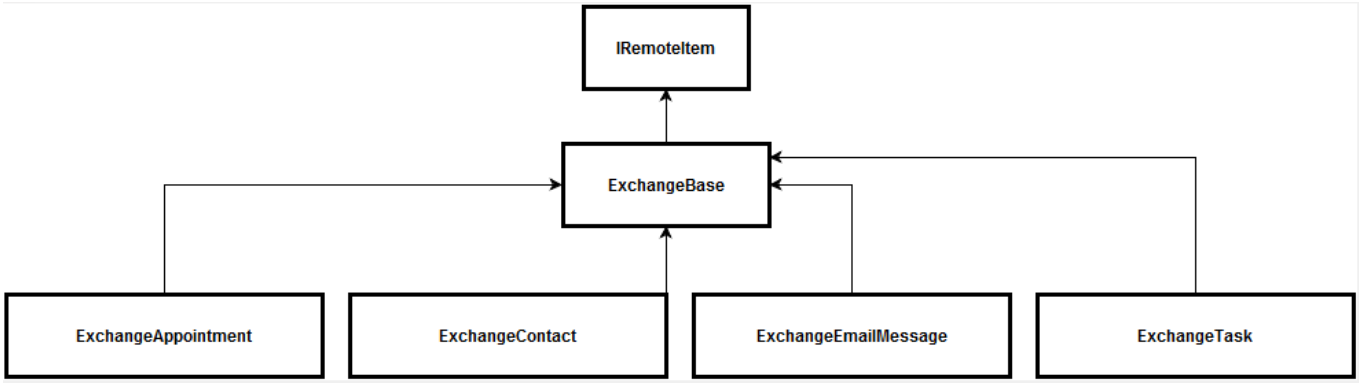
The *ExchangeEmailSyncProvider* class is the service provider for the exchange external storage. This class implements the logic of selecting data from MS Exchange.

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

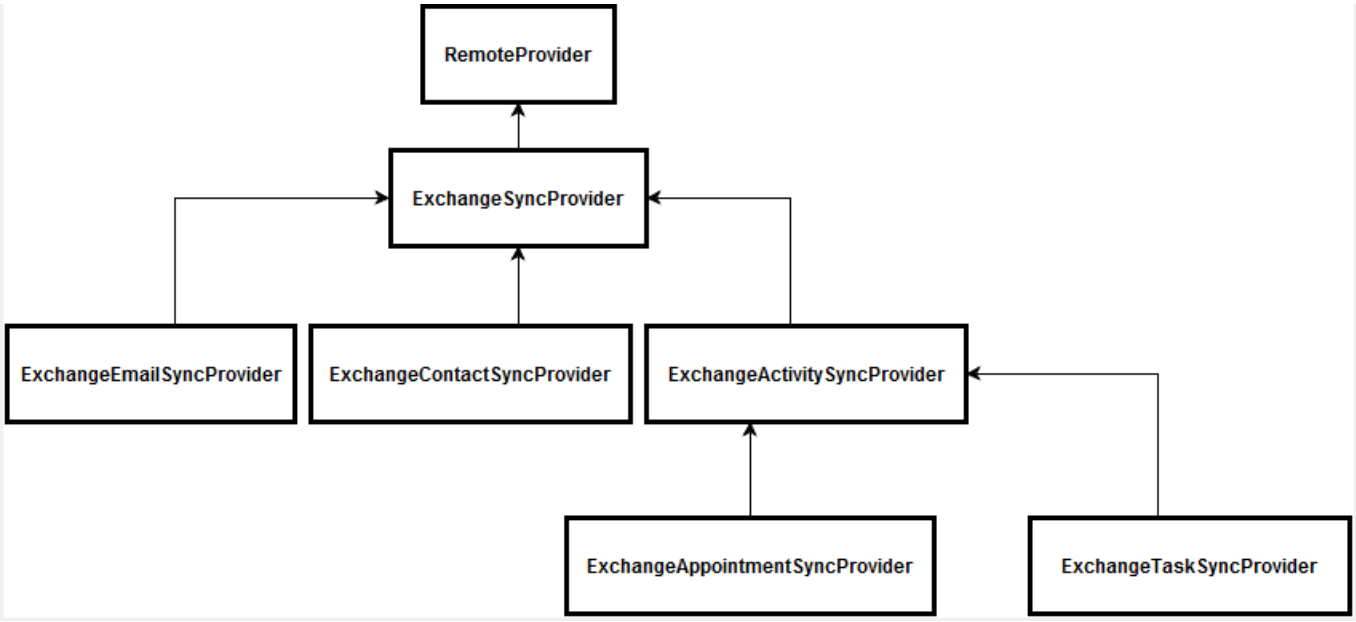
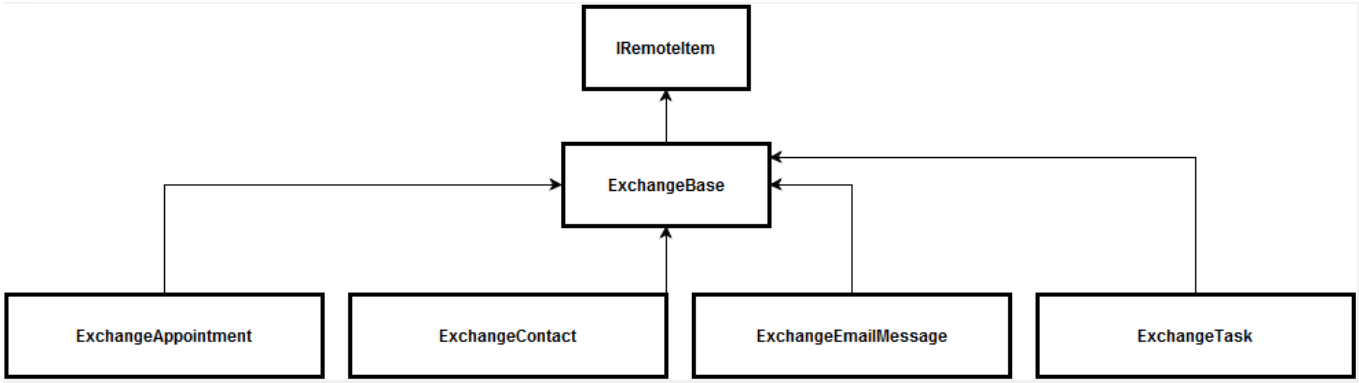


Fig. 2. RemoteItem hierarchy schema



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.

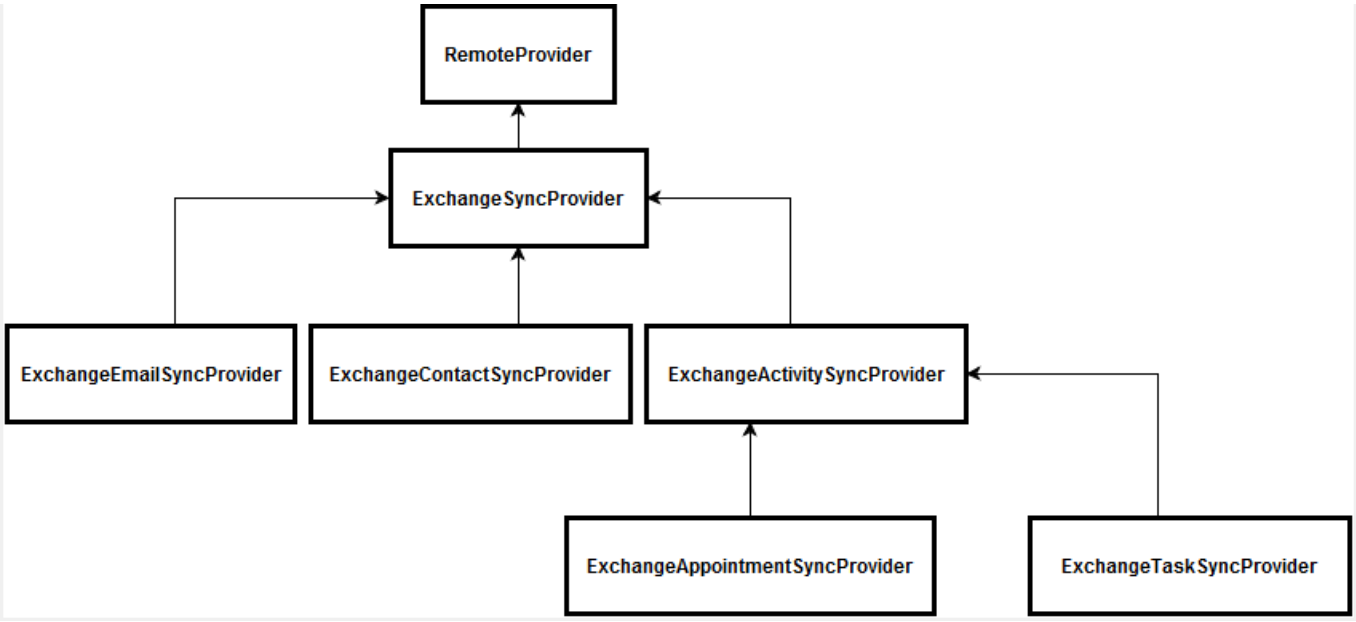
The appointment synchronization algorithm is no different from that described in the "**Bpm'online synchronization with external storages**" article. The process runs in three stages:

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).

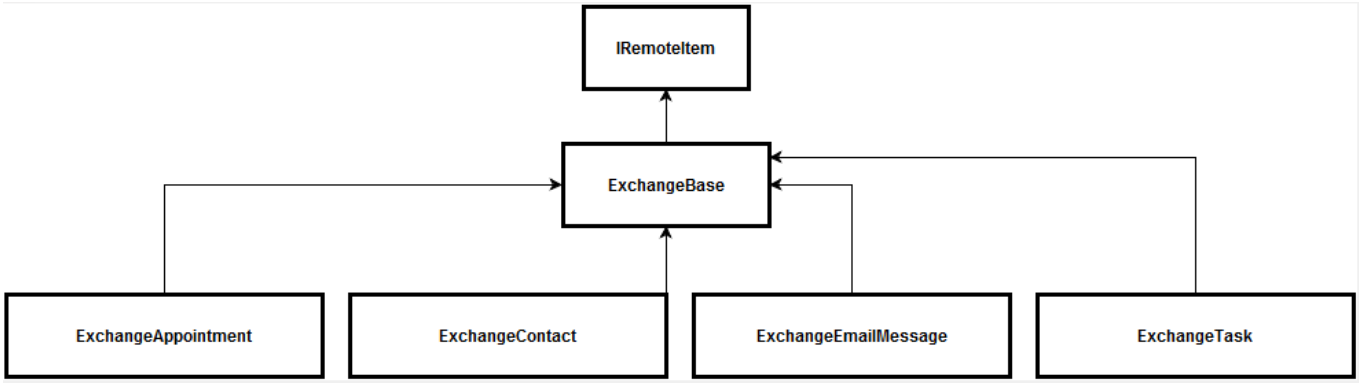
Fig. 1. RemoreProvider hierarchy schema



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

('!Untranslated: Обогащение контактов из email' in the on-line documentation)

- **API of cloud service for enriching contacts from email**

API of cloud service for enriching contacts from email

Introduction

The cloud service for enriching contacts from email uses different search technologies to find information about contacts and their communications (including web sites, profiles in social networks, phone numbers, etc). The service is a RESTful JSON Web service.

Base address of cloud service for all HTTP methods: https://cloud-service.bpmonline.com/Enrichment

Interactive documentation: http://docs.enrichcloud.apiary.io/

Using the Service

Service request

Extracting named entities from the body of an email:

- relative address – /getEmailEntities
- HTTP method for call – POST
- request (response) encoding – application/json
- search parameters – Table 1.

Table 1. The query parameters

Parameter	Type	Description
key	*string*	API key to bpm’online cloud services. The key is generated and specified automatically in the *CloudServicesAPIKey* system setting when deploying in the bpm'online cloud. For on-site applications, you must contact the support team for assistance.

emailBody	*string*	Email body.
msgType	*string*	Email content type. Possible values: html, plain.
senderEmail	*string*	Optional parameter. Sender's email address. If the email is specified, then its domain will be checked for belonging to public or temporary.

Example of request body:

```
{ "key":"string", "emailBody":"string", "msgType": "plain", "senderEmail":"string"}
```

Response from the service

Format of the response JSON object:

textEntities (object) – an object that contains typed collections of recognized named entities.

- *contacts (array)* – collection of recognized objects.
  - *id (guid)* – contact unique Id.
  - *organizationId (guid)* – Id of the associated account.
  - *givenName (string)* – contact name.
  - *surname (string)* – contact surname.
  - *middleName (string)* – contact middle name.
  - *jobTitle (string)* – contact job title.
  - *jobTitle (string)* – The original spelling of the entity in the email body.
  - *isFoundInLastEmail (boolean)* – a flag, whether the entity was found in the last email of the chain.
- *organizations (array)* – collection of recognized account objects.
  - *id (guid)* – account unique Id.
  - *parentId (guid)* – Id of the parent account.
  - *addressId (guid)* – Id of the account address.
  - *value (string)* – The original spelling of the entity in the email body.
  - *isFoundInLastEmail (boolean)* – a flag, whether the entity was found in the last email of the chain.
- *communications (array)* – A collection of recognized communication objects (phone numbers, email addresses, social networks, etc.)
  - *id (guid)* – communication option unique Id.
  - *contactId (guid)* – Id of the associated contact.
  - *type (string)* – type of the communication option. Email, phone number (fax, mobile), link to social network (facebook, twitter, etc.), messenger (Skype.)
  - *isPhone (boolean)* – a flag, whether the communication option is a phone number.
  - *isFreeDomain (boolean)* – a flag, whether the domain is public or temporary, if the entity is an e-mail address.
  - *value (string)* – The original spelling of the entity in the email body.
  - *isFoundInLastEmail (boolean)* – a flag, whether the entity was found in the last email of the chain.
- *addresses (array)* – collection of recognized address objects.
  - *id (guid)* – address unique Id.
  - *contactId (guid)* – Id of the associated contact.
  - *organizationId (guid)* – Id of the associated account.
  - *countryCode (string)* – country code.

cityNames (array) – an array of cities or possible spellings of one city, for example, Kiev/Kyiv. Currently only Russian and English are supported.

value (string) – The original spelling of the entity in the email body.

isFoundInLastEmail (boolean) – a flag, whether the entity was found in the last email of the chain.

jobs (array) – collection of recognized job titles of a contact.

id (guid) – job title unique Id.

contactId (guid) – Id of the associated contact.

value (string) – The original spelling of the entity in the email body.

isFoundInLastEmail (boolean) – a flag, whether the entity was found in the last email of the chain.

isSenderEmailDomainFree (boolean) – a flag, whether the domain of the sender's e-mail address is temporary or free. If the *senderEmail* incoming parameter is not specified, the *false* will be returned.

code (int) – error code (reserved).

status (string) – service response in text view ("ok", "error").

message (string) – detailed error message,

Example of response body:


```
{
  "textEntities": {
    "contacts": [{
        "givenName": "Homer",
        "surname": "Simpson",
        "middleName": "J."
        "jobTitle": "Dispatcher",
        "organizationId": "ea174f94be56455bb644c5415cacf1dd",
        "id": "ecc608e70e8241a49896e2326a56356b",
        "value": "Homer J. Simpson",
        "isFoundInLastEmail": true
    }],
    "organizations": [{
        "id": "ea174f94be56455bb644c5415cacf1dd",
        "value": "Nuclear Power Plant",
        "isFoundInLastEmail": true
    }],
    "jobs": [{
        "id": "da174f94be56455bb644c5415cacf1dd",
        "value": "Dispatcher",
        "contactId": "ecc608e70e8241a49896e2326a56356b",
        "isFoundInLastEmail": true
    }],
    "communications": [{
        "type": "email",
        "isFreeDomain": true,
        "isPhone": false,
        "contactId": "ecc608e70e8241a49896e2326a56356b",
        "id": "28887c9b737f42a98d0171e542786519",
        "value": "homer.simpson@gmail.com",
        "isFoundInLastEmail": false
    }],
    "addresses": [{
        "cityNames": ["Springfield"],
        "contactId": "ecc608e70e8241a49896e2326a56356b",
        "id": "17260c87b828474d982a71e63a2ad49f",
        "value": "Springfield, Evergreen Terrace, 742",
        "isFoundInLastEmail": true
    }]
```

```
  },
  "isSenderEmailDomainFree": true,
  "status": "ok",
  "code": 0
}
```

Example of service call

An example of service call from bpm'online configuration code with the RestSharp library:

```
using Enrichment.EmailMiningInterfaces;
using Newtonsoft.Json;
using NUnit.Framework;
using RestSharp;
using Terrasoft.Configuration.EmailMining;
using Terrasoft.Configuration.EnrichmentDto;

public class EmailMiningTestCase
{
       public TextEntities SearchEntities() {
                  var restClient = new RestClient();

                  // The URL of the enrichment service specified
                  // in the TextParsingService system setting.
                  restClient.BaseUrl = "https://cloud-
service.bpmonline.com/enrichment";

                  // General parameters of the POST request.
                  var request = new RestRequest {
                         Method = Method.POST,
                         RequestFormat = DataFormat.Json,
                         Resource = "getEmailEntities",
                         JsonSerializer = new NewtonsoftBasedSerializer()
                  };

                  // An example of a possible email body.
                  string emailBody = @"Hi!
                         Regards,
                         Homer Simpson";

                  // Add a request body. A valid API key is installed.
                  request.AddBody(new GetEmailEntitiesRequest {
                         ApiKey = "7QuPjMqxs1uXB0iE29HmMEJDCEA1VQQlXUzFMr5Y",
                         EmailBody = emailBody,
                         MsgType = "plain"
                  });

                // Receiving a response.
                  IRestResponse response = restClient.Post(request);
                  GetEmailEntitiesResponse responseInfo = JsonConvert
.DeserializeObject<GetEmailEntitiesResponse>(response.Content);

                  // List of found contact information.
                  return responseInfo.TextEntities;
       }
 }
```

Bpm'online lending

Terrasoft.Configuration.EntityMapper class

Terrasoft.Configuration.EntityMapper class

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>	*esq* – instance of the	The instance of the EntitySchemaQuery	Sets columns for selection from

columns)	EntitySchemaQuery class *Columns* – a dictionary of names of the mapped columns.	class.	the database.
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.

DataManager class description and use cases

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.

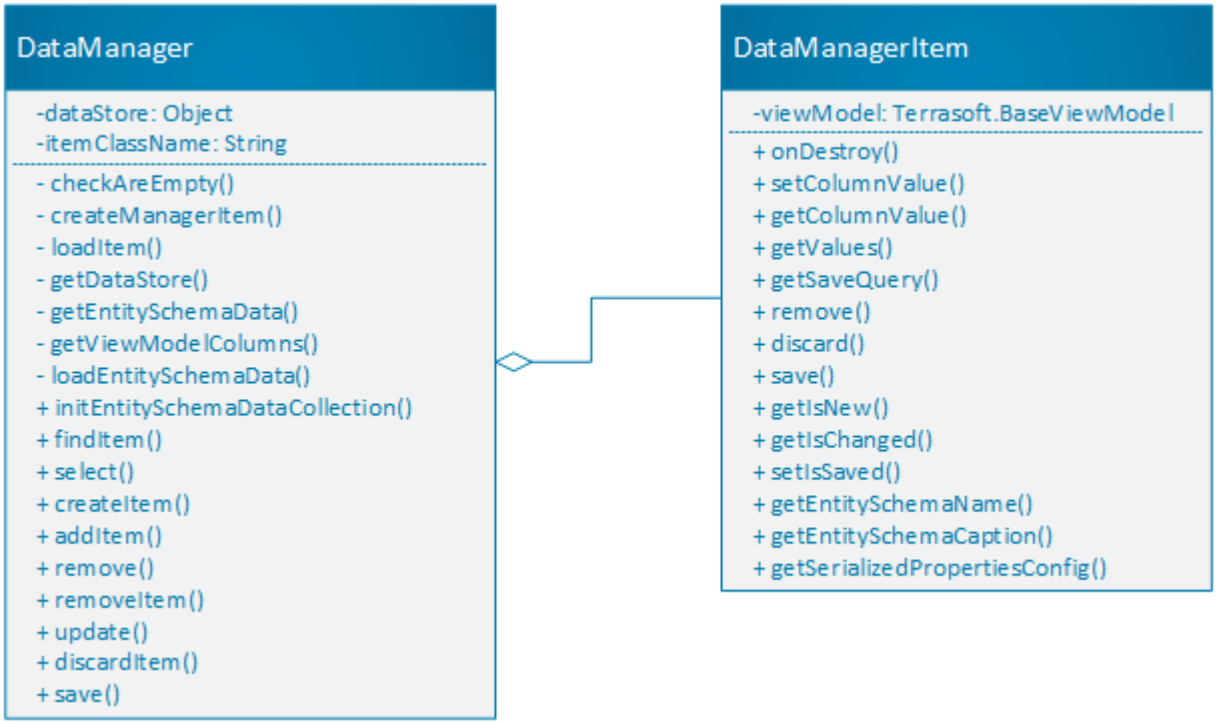
The *DataManager* class is a singleton available through the *Terrasoft* global object. This class provides the *dataStore* repository. The contents of one or more database tables can be loaded into the repository. Example:

```
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 disabling functionality

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.

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

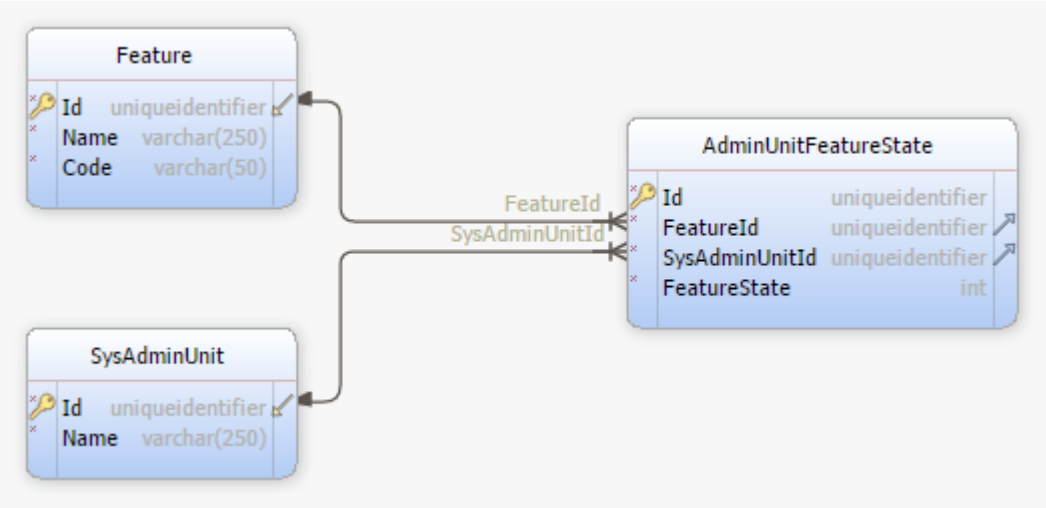
Name	Type	Description
Id	*uniqueidentifier*	Unique Id of the record
Name	*varchar(250)*	Functionality name.
Code	*varchar(50)*	Functionality code.

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 (optional)	*Number*	The value of the denomination (multiplicity) 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 own macros for a MS Word printable**" article.

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 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.

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.

Development recommendations for Right-To-Left mode

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%);
```

```
    }
}
```