

bpmonline



# **Bpm'online mobile**

## **Development Guide**



Simplify the future

## Table of Contents

Getting started with the bpm'online mobile platform	2
Mobile app architecture	2-6
How to start the development	7
Mobile application debugging	7-11
Platform description	12
Mobile application manifest	12-13
Manifest. Application interface properties	13-16
Manifest. Data and business logic properties	16-19
Manifest. Application synchronization properties	19-23
Batch mode export	23-24
Page life cycle in mobile application	24-27
Mobile application background update	27-28
Getting the settings and data from the [Dashboards] section	28-30
Resolving synchronization conflicts automatically	30-31
Mobile SDK	32
List SDK	32-34
Bpm'online mobile development cases	35
Adding a standard detail to the section in mobile application	35-40
Access modifiers of the page in the mobile application	40-41

## Getting started with the bpm'online mobile platform



### Mobile app architecture

Architecture, general schema and modes of the bpm'online mobile application.

## Mobile app architecture

### Difficulty level



## 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.

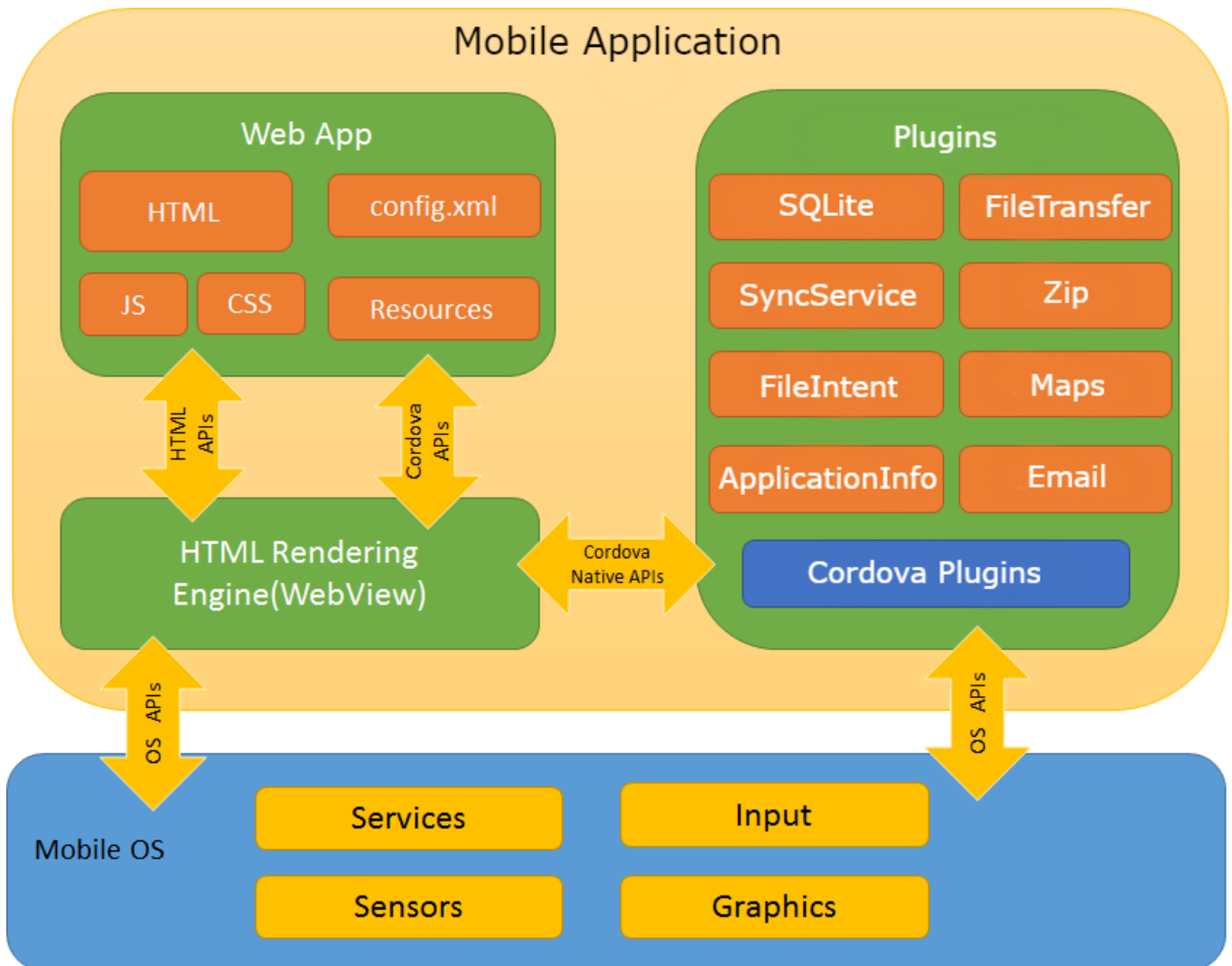
*Hybrid application* – a web application “encased” in a native shell. Hybrid applications are installed from the online shop (just like the native ones) and have access to the same functions of the mobile device, but are developed using HTML5, CSS and JavaScript. Unlike the native applications, hybrid 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 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 the 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

Internet connection is required.

Users work with bpm'online server directly.

Synchronization is required only upon configuration changes (adding and deleting columns, changing business logic).

### Offline

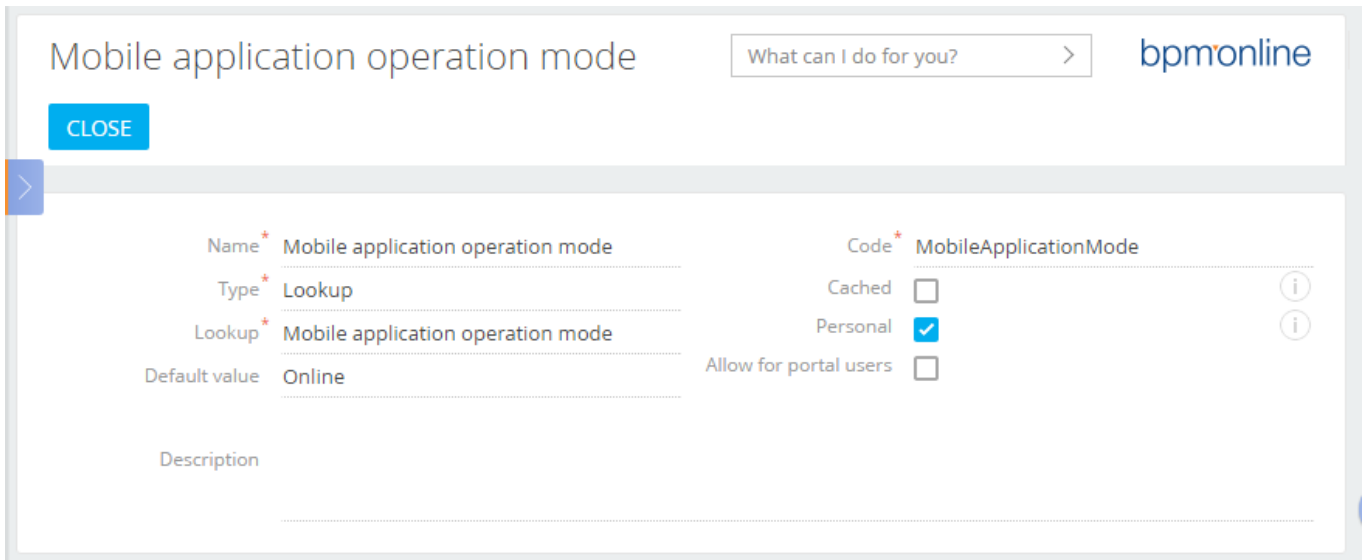
Internet connection is not required. Needed only for initial import and synchronization.

The data are saved locally, on the mobile device.

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.

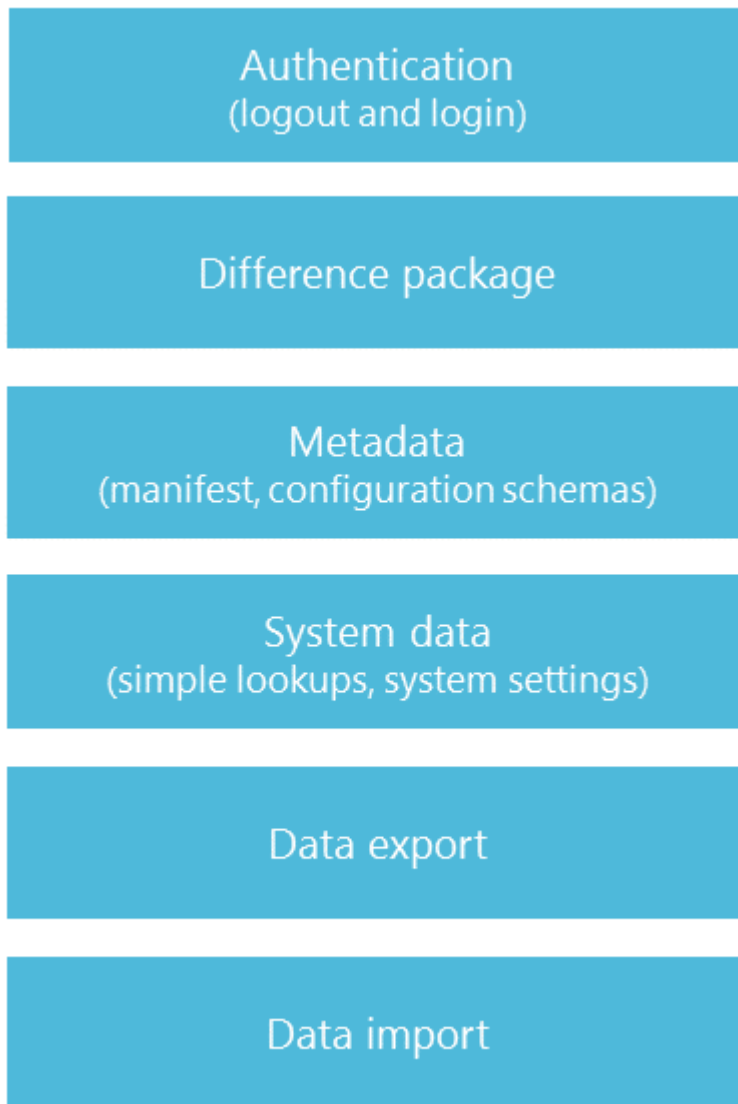
Fig. 3 The [Mobile application operation mode] system setting



## 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*.

## How to start the development

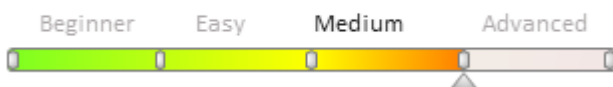


### Mobile application debugging

During the process of developing custom solutions for the bpm'online mobile application, you need to repeatedly perform application debug. More information about debugging the application code via browser development tools can be found in this article.

## Mobile application debugging

### Difficulty level



## 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:

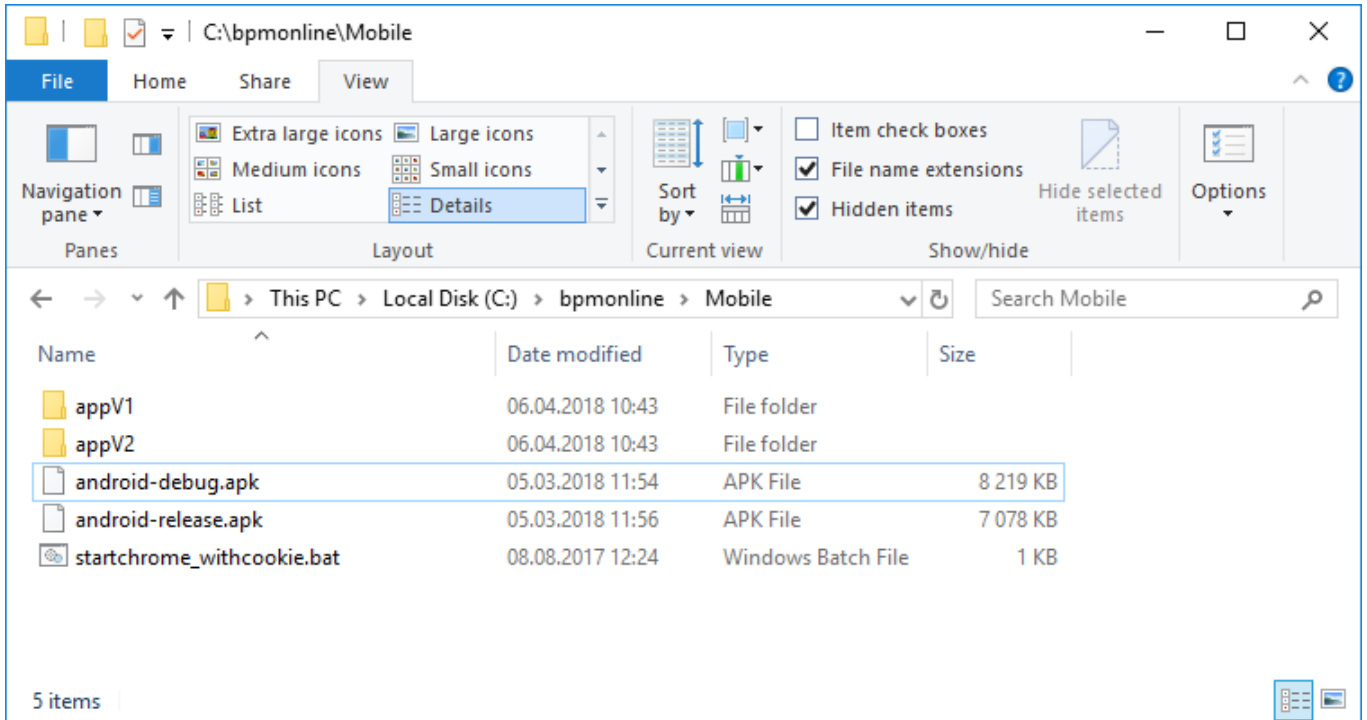
1. Get the files necessary for debugging the mobile application.
2. Launch the *startchrome.bat*.
3. Enter the debug mode for mobile devices in the Google Chrome.
4. Make the necessary settings and synchronize the mobile application bpm'online.

## 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





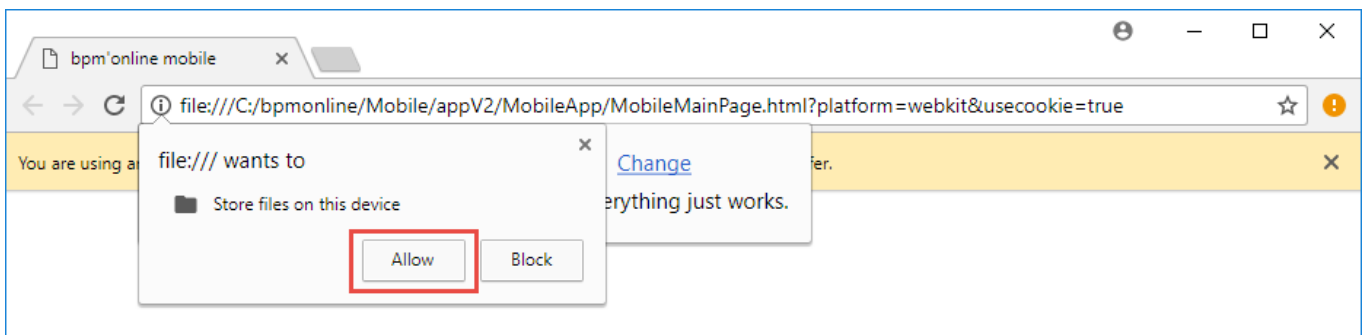
## Launch of the startchrome\_withcookie.bat

**ATTENTION**  
Close Google Chrome application before you launch *startchrome\_withcookie.bat*.

The *startchrome\_withcookie.bat* is located in the root folder of unpacked archive. The Google Chrome will launch after executing the *startchrome.bat*.

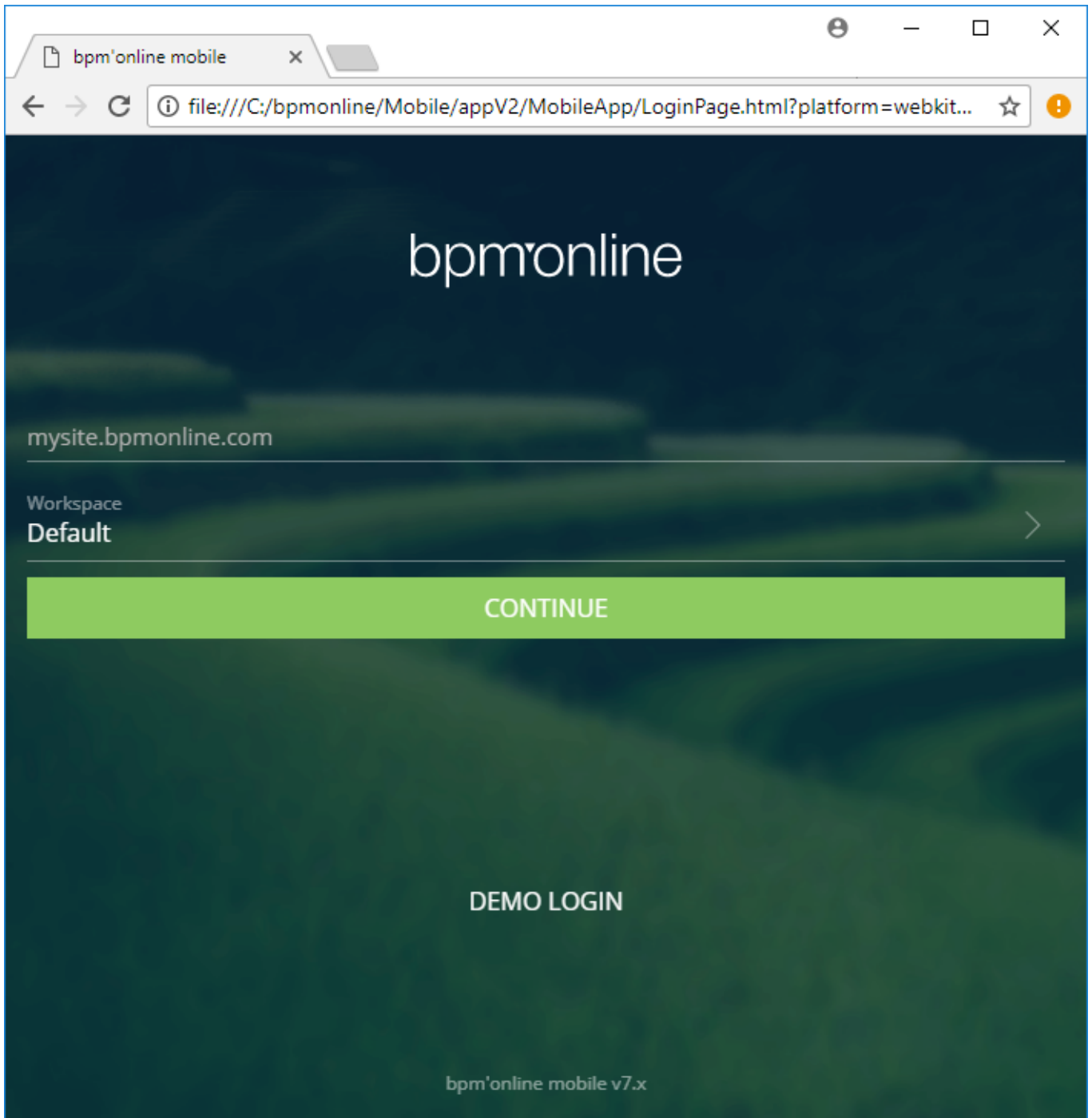
When you first start the browser with the *startchrome.bat*, an information window will appear, warning you about saving the files to the file system (Fig. 2). Allow the saving of the files. Close the warning about the *--disable-web-security* unsupported file (Fig. 2).

Fig. 2. Information window with warning



The execution of *startchrome\_withcookie.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 [Continue] button (Fig. 4). After that, enter user name and password (Fig. 5).

Fig. 4. Settings page of the local mobile application

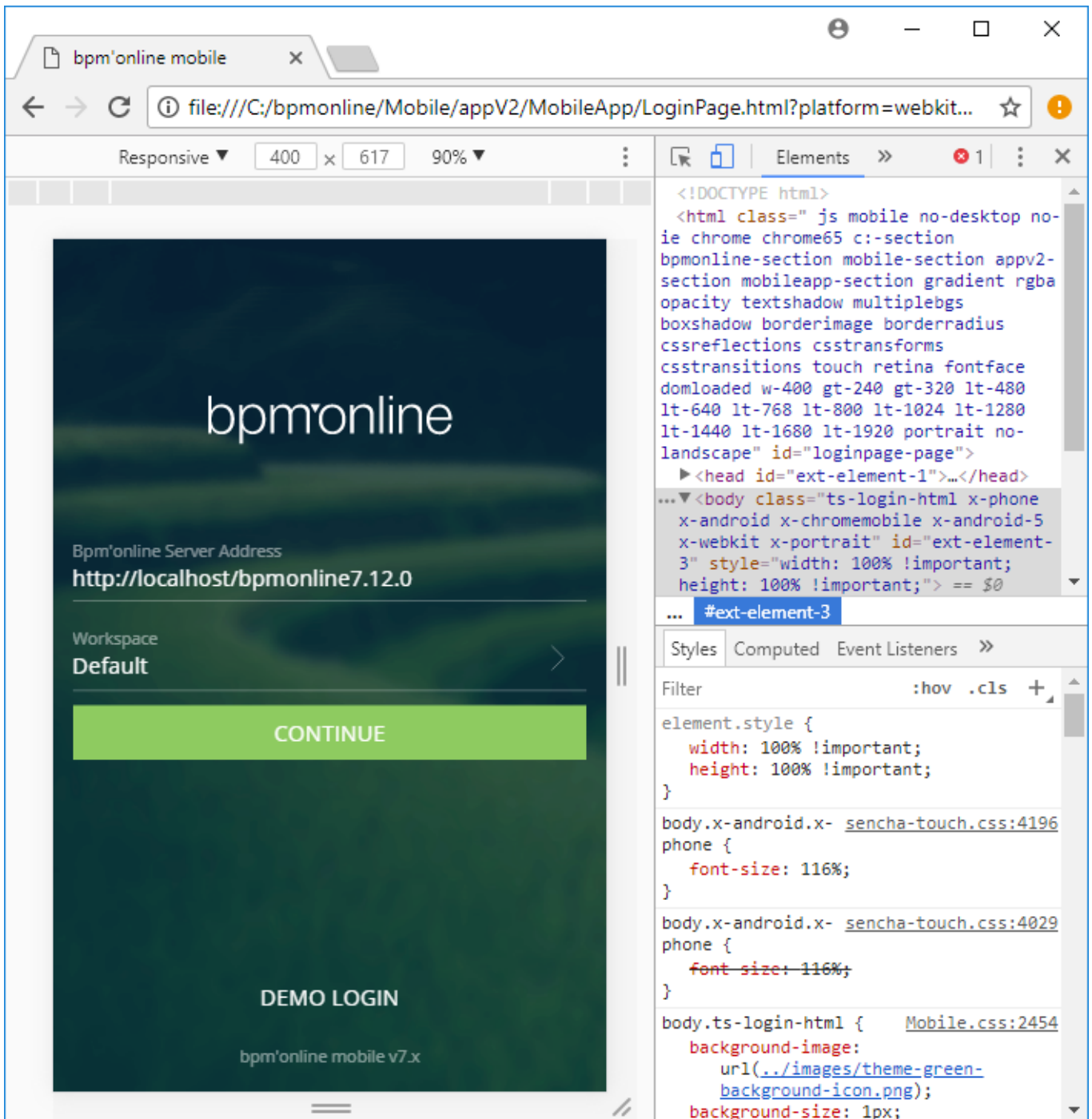
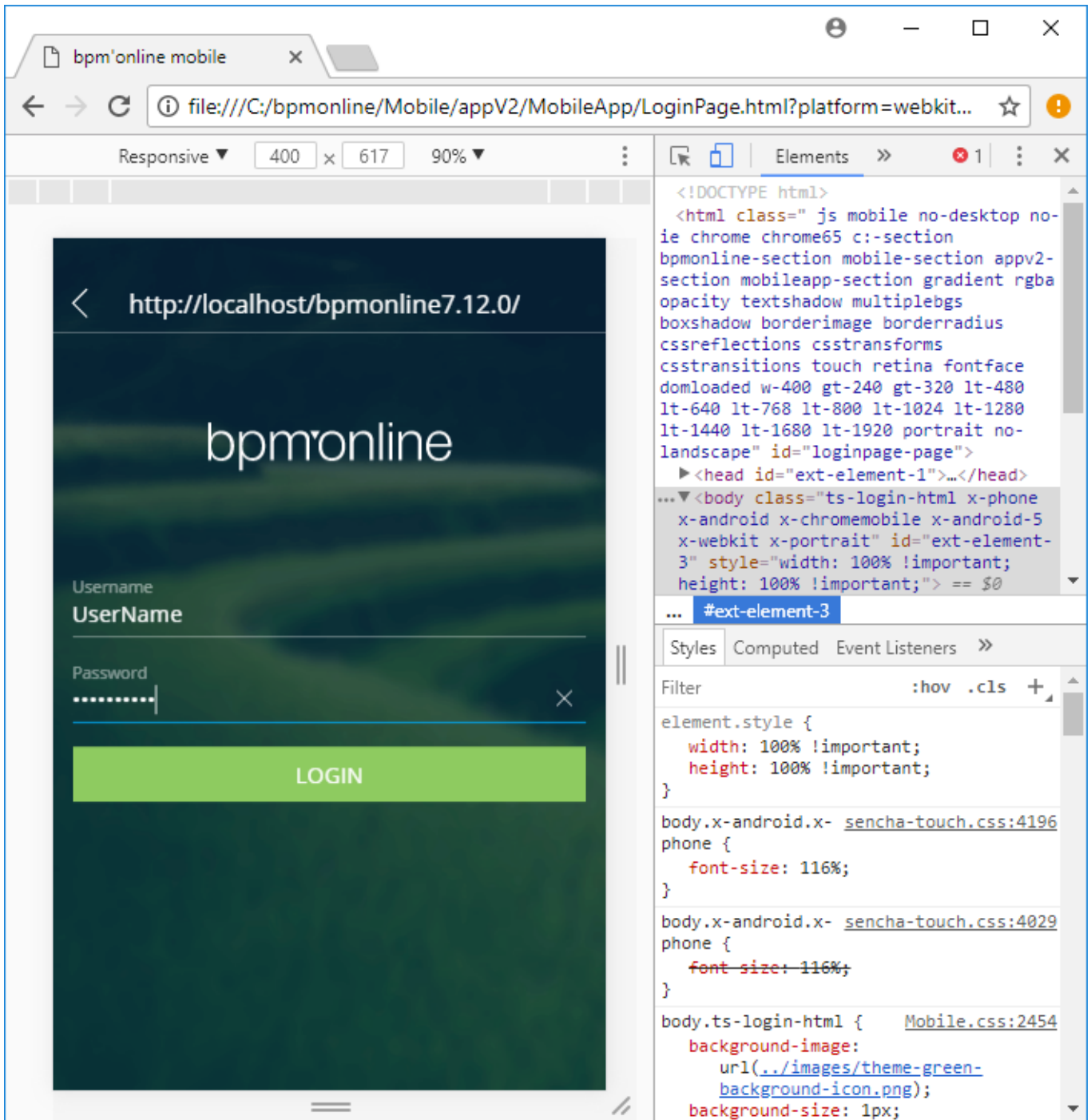


Fig. 5. Login page



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.

## Platform description



### Mobile application manifest

The mobile application manifest describes the structure of the mobile app, its objects and connections between them.

- **Manifest. Application interface properties**
- **Manifest. Data and business logic properties**
- **Manifest. Application synchronization properties**
- **Batch mode export**



### Page life cycle in mobile application

Each page in the mobile application has several stages during navigation process (opening, closing, unloading, returning to page, etc.). The time passed from loading a page, to unloading it from the mobile device memory is called a page life cycle.



### Mobile application background update

The bpm'online mobile application implements a synchronization mechanism for the application structure, which can work automatically in the background.



### Getting the settings and data from the [Dashboards] section

Getting the settings and the dashboards data is implemented in the *AnalyticsService* service and in the *AnalyticsServiceUtils* utility in the *Platform* package.



### Resolving synchronization conflicts automatically

During the synchronization of a mobile app working in the offline mode, the transferred data sometimes cannot be saved.

## Mobile application manifest

Difficulty level



## 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.

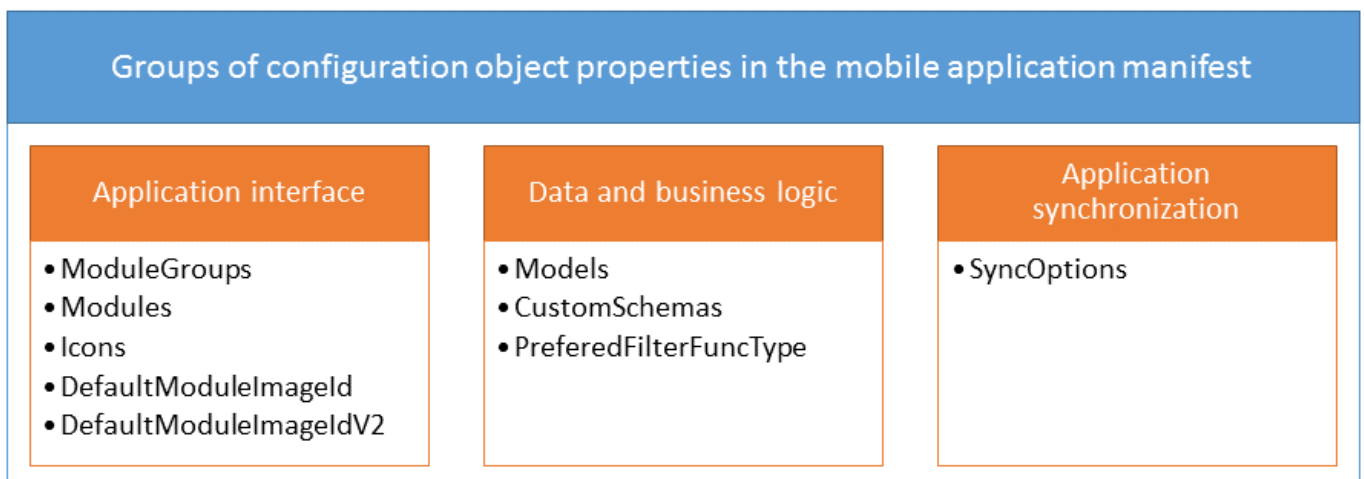
Table 1. Manifest configuration object properties

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**".

Fig. 1. Groups of configuration object properties in the configuration manifest



## Manifest. Application interface properties

Difficulty level



## 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. Starts 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 <i>ModuleGroups</i> 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 <i>Models</i> property of the manifest configuration object.
Position	Section position in the main menu group. Set by a numeric value starting with 0.
Title	Section title. String with the section title localized value name. Section title localized value name should be added to the <i>[LocalizableStrings]</i> 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 ( <i>true</i> — hidden, <i>false</i> — displayed). Optional property. By default — <i>false</i> .

**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,
    // 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,
    // Section title.
    "Title": "AccountSectionTitle",
    // Custom image import to section.
    "Icon": {
      // Unique image ID.
      "ImageId": "c046aala-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 <i>ImageListID</i> 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

### Difficulty level



## General provisions

The group of 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 PreferredFilterFuncType 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
<i>SubStringOf</i>	Determines whether a string passed as an argument, is a substring of the <i>property</i> string.
<i>ToUpper</i>	Returns values of the column specified in the <i>property</i> in relation to upper list.
<i>EndsWith</i>	Verifies if the <i>property</i> column value ends with a value passed as argument.
<i>StartsWith</i>	Verifies if the <i>property</i> column value starts with a value passed as argument.
<i>Year</i>	Returns year based on the <i>property</i> column value.
<i>Month</i>	Returns month based on the property column value.
<i>Day</i>	Returns day based on the property column value.
<i>In</i>	Checks if the property column value is within the value range passed as the function argument.
<i>NotIn</i>	Checks in the property column value is outside the value range passed as the function argument.
<i>Like</i>	Determines if the <i>property</i> 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 PreferredFilterFuncType property of the configuration object in the manifest must look like this:

```
// Substring search function is used to search for data.
"PreferredFilterFuncType": "Terrasoft.FilterFunctions.SubStringOf"
```

### ATTENTION

If the function specified as the data filtering function in the PreferredFilterFuncType 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

### Difficulty level



## 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 <i>Models</i> 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 ( <i>CreatedOn</i> , <i>CreatedBy</i> , <i>ModifiedOn</i> , <i>ModifiedBy</i> ) 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 <i>Terrasoft.FilterTypes</i> . Optional property. By default <i>Terrasoft.FilterTypes.Simple</i> . Filter types ( <i>Terrasoft.FilterTypes</i> ):

	<i>Simple</i>	Filter with one condition.
	<i>Group</i>	Group filter with multiple conditions.
logicalOperation	The logical operation for combining a collection of filters (for filters with <i>Terrasoft.FilterTypes.Group</i> type). Set by the enumeration value <i>Terrasoft.FilterLogicalOperations</i> . By default - <i>Terrasoft.FilterLogicalOperations.And</i> . Logical operation types ( <i>Terrasoft.FilterLogicalOperations</i> ):	
	<i>Or</i>	Logical operation OR.
	<i>And</i>	Logical operation AND.
subfilters	A collection of filters applied to a model. Obligatory property for the filter type <i>Terrasoft.FilterTypes.Group</i> . The filters are interconnected by the logical operation set in the <i>logicalOperation</i> property. Each filter is a configuration filter object.	
property	Filtered column model name. Obligatory property for the filter type <i>Terrasoft.FilterTypes.Simple</i> .	
valueIsMacrosType	The checkbox that defines whether the filtered value is a macro. Optional property can be: <i>true</i> if the filter uses a macro, and <i>false</i> if it doesn't.	
value	Value of the column filtration set in the <i>property</i> property. Obligatory property for the filter type <i>Terrasoft.FilterTypes.Simple</i> . Can be set directly by the filter value (including <i>null</i> ) or a macro (the <i>valueIsMacrosType</i> property must be set to <i>true</i> ). Macros that can be used as the property value are contained in the <i>Terrasoft.ValueMacros</i> enumeration. Value macros ( <i>Terrasoft.ValueMacros</i> ):	
	<i>CurrentUserContactId</i>	Current user ID.
	<i>CurrentDate</i>	Current date.
	<i>CurrentDateTime</i>	Current date and time.
	<i>CurrentDateEnd</i>	Current date end.
	<i>CurrentUserContactName</i>	Current contact name.
	<i>CurrentUserContact</i>	Current contact name and ID.
	<i>SysSettings</i>	System setting value. The system setting name is included in the <i>macrosParams</i> property.
	<i>CurrentTime</i>	Current time.
	<i>CurrentUserAccount</i>	Current account name and ID.
	<i>GenerateUId</i>	Generated ID.
macrosParams	Values transitioned to macros as parameters. Optional property. This property is now used only for the <i>Terrasoft.ValueMacros.SysSettings</i> macro.	
isNot	Applied to the negation operator filter. Optional property. Takes the <i>true</i> value if the the negation operator is applied to the filter, otherwise — <i>false</i> .	
funcType	Function type applied to the model column set in the <i>property</i> property. Optional property. Takes values from the <i>Terrasoft.FilterFunctions</i> enumeration. Argument values for the filtration functions are set in the <i>funcArgs</i> property. The value to compare the result of the function is specified by the <i>value</i> property. Filtration functions ( <i>Terrasoft.FilterFunctions</i> ):	
	<i>SubStringOf</i>	Determines whether the string passed in as an argument is a substring of the <i>property</i> column.
	<i>ToUpper</i>	Changes the column value set in the <i>property</i> to uppercase.
	<i>EndsWith</i>	Checks whether the value in the <i>property</i> column ends with the

		value set as an argument.
	<i>StartsWith</i>	Checks if the value of the <i>property</i> column starts with the value set as an argument.
	<i>Year</i>	Returns the year value according to the <i>property</i> column.
	<i>Month</i>	Returns the month value according to the <i>property</i> column.
	<i>Day</i>	Returns the day value according to the <i>property</i> column.
	<i>In</i>	Checks the occurrence of the value of the column property in the range of values that is passed as argument to the function.
	<i>NotIn</i>	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.
	<i>Like</i>	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 <i>funcType</i> property. The order of the values in the array <i>funcArgs</i> must match the order of parameters of the <i>funcType</i> 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 <i>Terrasoft.FilterOperation</i> enumeration. By default — <i>Terrasoft.FilterOperation.General</i> .	
	Filtration operations ( <i>Terrasoft.FilterOperation</i> ):	
	<i>General</i>	Standard filtration.
	<i>Any</i>	Filtration by the <i>exists</i> filter.
compareType	Filter comparison operation type. Optional parameter. Takes values from the <i>Terrasoft.ComparisonType</i> enumeration. By default — <i>Terrasoft.ComparisonType.Equal</i> .	
	Comparison operations ( <i>Terrasoft.ComparisonType</i> ):	
	<i>Equal</i>	Equal.
	<i>LessOrEqual</i>	Less or equal.
	<i>NotEqual</i>	Not equal.
	<i>Greater</i>	Greater.
	<i>GreaterOrEqual</i>	Greater or equal.
	<i>Less</i>	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", "OppContactLoyalty", "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": []
    }
    ]
}

```

## The SyncOptions.ModelDataImportConfig.QueryFilter property

Available in the bpm'online application starting with version 7.12.1 and in the bpm'online mobile application starting with version 7.12.3.

The *QueryFilter* synchronization property enables to configure data filtering of the specific model when importing via the [DataService](#) service. Previously, the *SyncFilter* property was used to filter data and the import was performed via the [OData \(EntityDataService\)](#).

 ATTENTION

Data import via the DataService service is available only for the Android and iOS platforms. The OData (EntityDataService) is used for the Windows platform.

The *QueryFilter* filter is a set of parameters in the form of JSON object that are sent in the request to the DataService service. Description of the DataService parameters can be found in the "[DataService. Data filtering](#)" development guide article.

Example of the exists filter is available below:

```
{
  "SyncOptions": {
    "ModelDataImportConfig": [
      {
        "Name": "ActivityParticipant",
        "QueryFilter": {
          "logicalOperation": 0,
          "filterType": 6,
          "rootSchemaName": "ActivityParticipant",
          "items": {
            "ActivityFilter": {
              "filterType": 5,
              "leftExpression": {
                "expressionType": 0,
                "columnPath": "Activity.[ActivityParticipant:Activity].Id"
              },
              "subFilters": {
                "logicalOperation": 0,
                "filterType": 6,
                "rootSchemaName": "ActivityParticipant",
                "items": {
                  "ParticipantFilter": {
                    "filterType": 1,
                    "comparisonType": 3,
                    "leftExpression": {
                      "expressionType": 0,
                      "columnPath": "Participant"
                    },
                    "rightExpression": {
                      "expressionType": 1,
                      "functionType": 1,
                      "macroType": 2
                    }
                  }
                }
              }
            }
          }
        }
      }
    ]
  }
}
```

Batch mode export



### Difficulty level



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.

## Page life cycle in mobile application

### Difficulty level



## Introduction

Each page in the mobile application has several stages during navigation process (opening, closing, unloading, returning to page, etc.). The time passed from loading a page, to unloading it from the mobile device memory is called a page life cycle.

For each stage of page life cycle provided the corresponding page event. Use page events to expand functionality. Main page events:

- initialization of the view
- completion of class initialization
- page loading
- data uploading
- page closing.

Understanding the page life cycle stages enables you to enlarge the logic of the pages.

## Life cycle stages

### ATTENTION

Only one page can be displayed on the mobile phone screen. Tablet PC can display one page in a portrait orientation and two pages in a landscape orientation. Due to this, the page life cycle differs for the phone and the tablet.

### Page opening

At first opening of the page, all scripts necessary for the work of the page are being loaded. After that the controller is being initialized and the view is created.

Sequence of page opening event generation:

1. *initializeView* – view initialization.
2. *pageLoadComplete* – event of completion of the page loading.
3. *launch* – initiates data loading.

## Page closing

When the page is closed, its view is deleted from the document object model (DOM) and controller is being deleted from the device memory.

Page is closed in the following cases:

- The [Back] button is pressed. In this case the last page is being deleted.
- New section was opened. In this case all pages that were opened before are being deleted.

*pageUnloadComplete* – the event of page closing completion.

## Page unloading

Unloading is performed after the passing to another page in the same section. The current page becomes inactive. It can stay visible on the device screen. For example, if you open a view page from the list on the tablet PC, the list page will stay visible. In the same case on the phone, the list page will not be visible but will stay in the memory. This is the difference between unloading and closing a page.

*pageUnloadComplete* – the event of page unload (coincides with the event of the page closing).

## Return to the page

Return to the unloaded page is performed by pressing the [Back] button.

*pageLoadComplete* – returning to the page event.

### ATTENTION

Only one instance of the page can be used in the application. If you consistently open two identical pages and return to the first page, the *launch* event handler will be executed again. This should be taken into account in the development.

## Life cycle event handlers

Page controller classes are inherited from the *Terrasoft.controller.BaseConfigurationPage* class that provides methods of handling the life cycle events.

### **initializeView(view)**

Method is called after the page view in the DOM is being created (but was not rendered). On this stage you can subscribe to the events of the view classes and perform additional actions with DOM.

### **pageLoadComplete(isLaunch)**

Provides extension of the logic that is executed at the page load and return. The *true* value of the *isLaunch* parameter indicates that the page is being loaded for the first time.

### **launch()**

Called only when the page is opened. The method initiates the loading of data. If you need to load additional data, use the *launch()* method.

### **pageUnloadComplete()**

Provides extension of the logic that is executed at the page unload and closure.

## Page navigation

The *Terrasoft.PageNavigator* class manages the life cycle of the pages. The class enables opening and closing of the pages, updating of the irrelevant data and storing the page history.

### **forward(openingPageConfig)**

Method opens page according to the properties of the *openingPageConfig* configuration parameter object. Main properties of this object are listed in the Table 1.

Table 1. The *openingPageConfig* object properties

Property	Description
<i>controllerName</i>	Name of the controller class.
<i>viewXType</i>	View type according to xtype.
<i>type</i>	Page type from the <i>Terrasoft.core.enums.PageType</i> enumeration.
<i>modelName</i>	Name of the page model.
<i>pageSchemaName</i>	Name of the page schema in configuration.
<i>isStartPage</i>	Flag indicating that the page is a start page. If previously the pages have been opened, they will be closed.
<i>isStartRecord</i>	Flag indicating that the view/edit page should be the first after the list. If there are other opened pages after the list, they will be closed.
<i>recordId</i>	Id of the record of the page being opened.
<i>detailConfig</i>	Settings of the standard detail.

## backward()

The method is closing the page.

## markPreviousPagesAsDirty(operationConfig)

Method marks all previous pages as irrelevant. After returning to previous pages, the *refreshDirtyData()* method is called for each page. The method re-loads the data or updates the data basing on the *operationConfig* object.

## refreshPreviousPages(operationConfig, currentPageHistoryItem)

Method re-loads data for all previous pages and updates the data basing on the *operationConfig* object. If the value is set for the *currentPageHistoryItem* parameter, the method performs the same actions for the previous pages.

## refreshAllPages(operationConfig, excludedPageHistoryItems)

Method re-loads data for all pages or updates the data basing on the *operationConfig* object. If the *excludedPageHistoryItems* parameter is set, the method does not update the specified pages.

# Navigation with routes

## Routing

Routing is used for managing visual components: pages, pickers, etc. The route has 3 states:

1. *Load* – opens a current route.
2. *Unload* – closes current route on return.
3. *Reload* – restores the previous route on return.

The *Terrasoft.Router* class is used for routing and it's main methods are *add()*, *route()* and *back()*.

## add(name, config)

Adds a route. Parameters:

- *name* – Unique name of the route. In case of re-adding, the latest route will override the previous one.
- *config* – describes names of the functions that handle route states. Handlers of the route states are set in the *handlers* property.

Use case:

```
Terrasoft.Router.add("record", {
  handlers: {
    load: "loadPage",
    reload: "reloadPage",
    unload: "unloadLastPage"
  }
});
```

## route(name, scope, args, config)

Starts the route. Parameters:

- *name* – name of the route.
- *scope* – context of the function of the state handlers.
- *args* – parameters of the functions of the state handlers.
- *config* – additional route parameters.

Use case:

```
var mainPageController = Terrasoft.util.getMainController();
Terrasoft.Router.route("record", mainPageController, [{pageSchemaName:
"MobileActivityGridPage"}]);
```

## back()

Closes current route and restores previous.

# Mobile application background update

### Difficulty level



## 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

Name *	Update checks frequency	Code *	MobileAppCheckUpdatePeriod
Type *	Integer	Cached	<input checked="" type="checkbox"/>
Default value	10	Personal	<input type="checkbox"/>
		Allow for portal users	<input type="checkbox"/>
	Value in hours		
Description			

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.

## Getting the settings and data from the [Dashboards] section

Difficulty level



## 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 contains 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 a 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

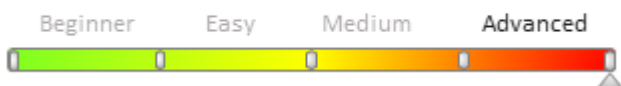
```
{
  "dashboardId": "a71d5c04-dff7-4892-90e5-9e7cc2246915",
  "itemName": "Chart4",
  "timeZoneOffset": 120
}
```

## 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": []
}
```

## Resolving synchronization conflicts automatically

### Difficulty level



## Introduction

During the synchronization of a mobile app working in the offline mode, the transferred data sometimes cannot be saved. This happens if:

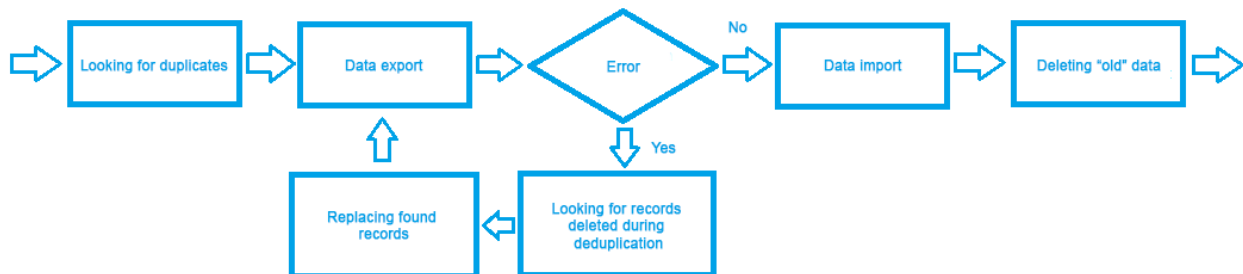
- A record was merged with a duplicate, and therefore does not exist.
- A record was deleted.

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.



## Mobile SDK



### List SDK

Classes, methods and properties of the bpm'online mobile application list.

## List SDK

### Difficulty level



#### ATTENTION

This article is relevant for mobile application version 7.11.1 or higher.

## Introduction

List SDK is a tool that enables to configure list layout, sorting, search logic, etc. It is implemented on the *Terrasoft.sdk.GridPage*.

## Terrasoft.sdk.GridPage methods

### setPrimaryColumn()

Sets the primary display column. Configures the displaying of a title of the list record.

#### Method signature

```
setPrimaryColumn(modelName, column)
```

#### Parameters

*modelName* – model name.

*Column* – column name.

#### Example of call

```
Terrasoft.sdk.GridPage.setPrimaryColumn("Case", "Subject");
```

### setSubtitleColumns()

Sets the columns displayed under the title. Sets the subtitle display as a list of columns with a separator.

#### Method signature

```
setSubtitleColumns(modelName, columns)
```

#### Parameters

*modelName* – model name.

*columns* – an array of columns or column configuration objects.

#### Example of call

#### Option 1

```
Terrasoft.sdk.GridPage.setSubtitleColumns("Case", ["RegisteredOn", "Number"]);
```

#### Option 2

```
Terrasoft.sdk.GridPage.setSubtitleColumns("Case", ["RegisteredOn", { name: "Number",  
convertFunction: function(values) { return values.Number; } }]);
```

### setGroupColumns()

Sets a group with columns that are displayed vertically. Configures displaying the group of columns.

#### Method signature

```
setGroupColumns(modelName, columns)
```

#### Parameters

*modelName* – model name.

*columns* – an array of columns or column configuration objects.

#### Example of call

##### Option 1

```
Terrasoft.sdk.GridPage.setGroupColumns("Case", ["Symptoms"])
```

##### Option 2

```
Terrasoft.sdk.GridPage.setGroupColumns("Case", [  
{  
name: "Symptoms",  
isMultiline: true, //Display as multi-line field  
label: "CaseGridSymptomsColumnLabel", //Name of the localized string  
convertFunction: function(values) {  
return values.Symptoms;  
}  
}]);
```

### setImageColumn()

Sets the image column.

### setOrderByColumns()

Sets the list sorting.

### setSearchColumn()

Sets the search column.

### setSearchColumns()

Sets the search columns.

### setSearchPlaceholder()

Sets the hint text in the search field.

### setTitle()

Sets the title of the list page.

## Example

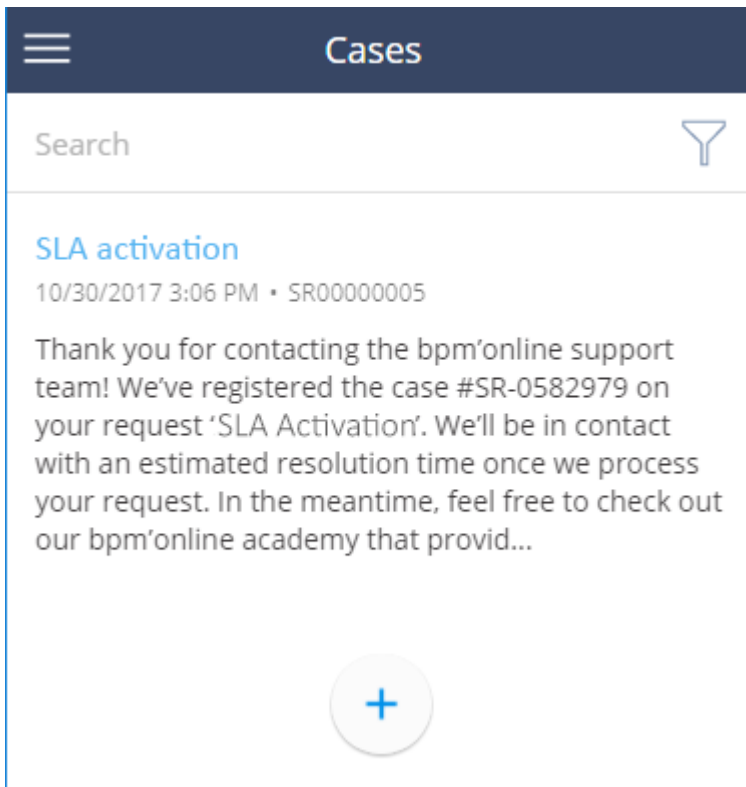
Configure the [Cases] section list to display the title with the case subject, subtitle with the registration date and case number and the case description as the multi-line field.

Use the following source code to configure the list:

```
// Configure the primary column with the case subject.
Terrasoft.sdk.GridPage.setPrimaryColumn("Case", "Subject");
// Setting the subtitle with the registration date and the case number.
Terrasoft.sdk.GridPage.setSubtitleColumns("Case", ["RegisteredOn", "Number"]);
// Adding a multi-line field with the description.
Terrasoft.sdk.GridPage.setGroupColumns("Case", [
{
name: "Symptoms",
isMultiline: true
}]);
```

As a result, the list will be displayed as shown on Fig. 1.

Fig. 1. Configured list of cases



## Bpm'online mobile development cases



### **Adding a standard detail to the section in mobile application**

Use the Mobile application wizard to add a detail to the section of mobile application. If the detail object is not a section object of the bpm'online mobile application, the detail will display the id of the connected section record instead of record values. Configure the schema of the detail page to display values.



### **Access modifiers of the page in the mobile application**

The mobile application version 7.11.0 has the ability to configure access modifiers of section or standard detail. For example, you can disable modifying, adding and deleting records for all users in the section.

## Adding a standard detail to the section in mobile application

### **Introduction**

Use the Mobile application wizard to add a detail to the section of mobile application. The setting up a detail via mobile application wizard is described in the [“How to set up a standard detail”](#) article.

If the detail object is not a section object of the bpm'online mobile application, the detail will display the id of the connected section record instead of record values. Configure the schema of the detail page to display values.

### **Case description**

Add the [Job experience] detail on the edit page of the [Contacts] section of mobile application. Display the [Job title] column as primary column.

### **Source code**

You can download the package with case implementation using the following [link](#).

### **Case implementation algorithm**

#### **1. Add the [Job experience] detail via the mobile application wizard**

Use the [mobile application wizard](#) to add a detail on the record edit page. To do this:

- 1.1. Open the necessary workplace (for example [Main workplace]) and click the [Set up sections].
- 1.2. Select the [Contacts] section and click the [Details setup] button.
- 1.3. Set up the [Job experience] detail (Fig.1).

Fig. 1. Setting up the [Job experience] detail

Detail setting

SAVE CANCEL

Detail\* Job experience

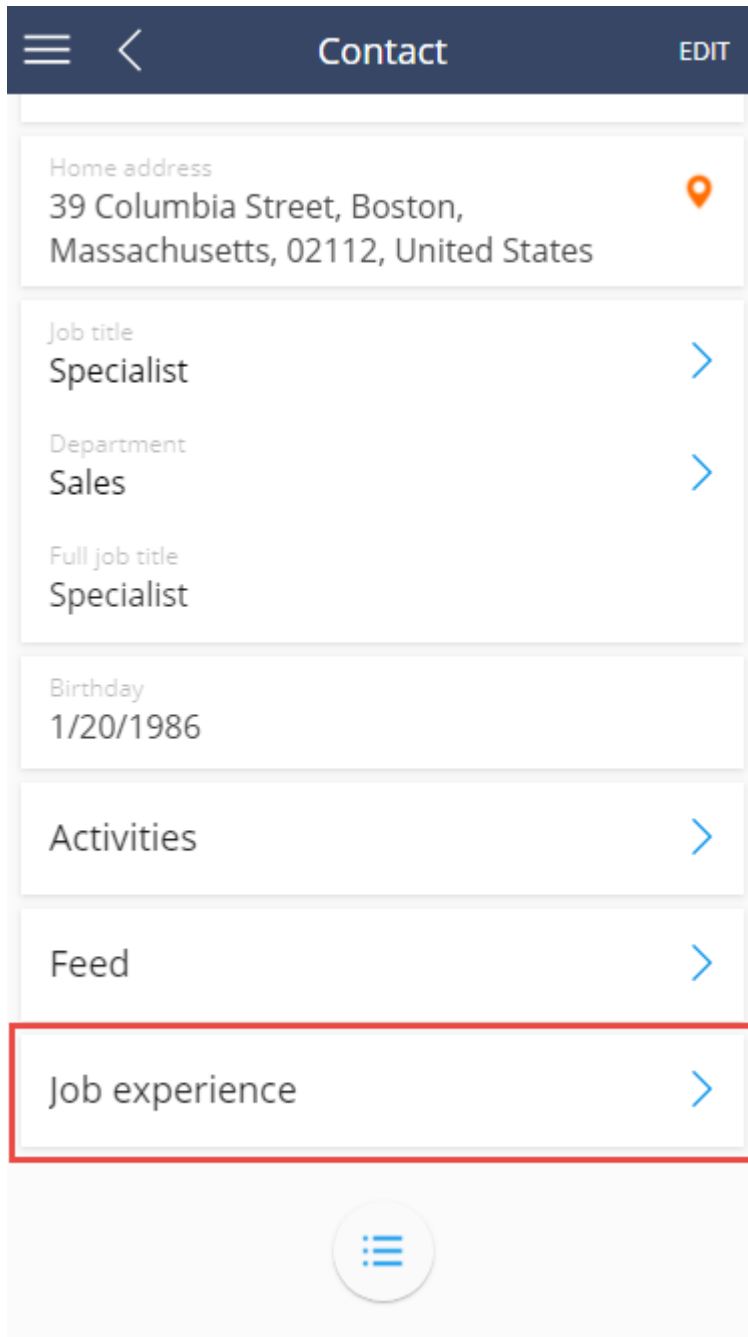
Title Job experience

Detail column\* Contact

Object column 'Contact'\* Id

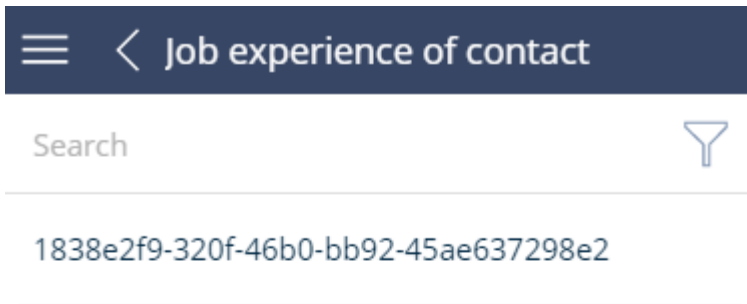
After saving the setup of detail, section and workplace, the [Job experience] detail will be displayed in the mobile application (Fig. 2).

Fig. 2. [Job experience] detail on the [Contacts] section record page



If the [Job experience] detail object is not a section object of the bpm'online mobile application, the detail will display the value of the [Contact] primary column (id of the connected record of the contact).

Fig. 3. Displaying id of the connected record of the contact



## 2. Create module schema in which to configure the detail list

Use the [\[Configuration\] section](#) to [create custom module](#) in the custom package with following properties (Fig. 4):

- [Title] – "Contact Career Configuration".
- [Name] – "UsrContactCareerModuleConfig".

Fig. 4. Properties of the module schema

Properties	
<input type="text" value="&lt;Enter search text&gt;"/>	
<div style="border: 1px solid #ccc; padding: 5px;"> <p><b>General</b></p> <p><b>Title</b> <input type="text" value="Contact Career Configuration"/></p> <p><b>Name</b> <input type="text" value="UsrContactCareerModuleConfig"/></p> <p><b>Package</b> <input type="text" value="AddDetailMobile"/></p> <p><b>Inheritance</b></p> <p>Parent object <input type="text"/></p> <p>Forbid substitution <input type="checkbox"/></p> <p>Replace parent <input type="checkbox"/></p> </div>	

Add the following source code to the module schema:

```
// Setting the [Job title] column as primary column.
Terrasoft.sdk.GridPage.setPrimaryColumn("ContactCareer", "JobTitle");
```

```
// Adding the [Job title] column to the primary column collection.
Terrasoft.sdk.RecordPage.addColumn("ContactCareer", {
    name: "JobTitle",
    position: 1
}, "primaryColumnSet");
// Delete the [Contact] previous primary column from the primary column collection.
Terrasoft.sdk.RecordPage.removeColumn("ContactCareer", "Contact",
"primaryColumnSet");
```

In this code:

- "ContactCareer" – name of the table that corresponds to the detail (as a rule it matches the name of the detail object).
- "Job Title" – name of the column that should be displayed on the page.

### 3. Connect the module schema in the mobile application manifest

To apply list settings performed in the *UsrContactCareerModuleConfig* module, perform following:

- 3.1. Open the schema of the mobile application manifest (*MobileApplicationManifestDefaultWorkplace*) in the custom module designer. This schema is created in the custom package by the mobile application wizard (see the **"How to add a custom section to a mobile application (on-line documentation)"** article).
- 3.2. Add the *UsrContactCareerModuleConfig* module to the *PagesExtensions* section of the *ContactCareer* model:

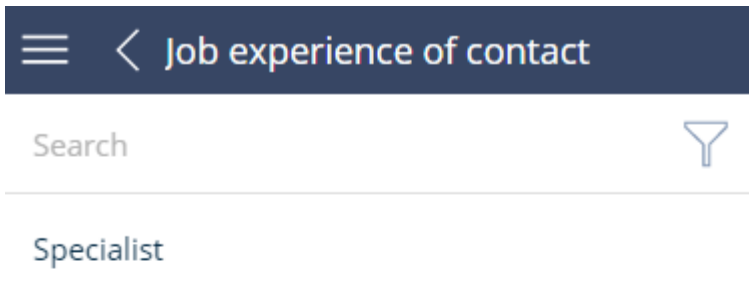
```
{
  "SyncOptions": {
    ...
  },
  "Modules": {},
  "Models": {
    "ContactCareer": {
      "RequiredModels": [
        ...
      ],
      "ModelExtensions": [],
      "PagesExtensions": [
        ...
        "UsrContactCareerModuleConfig"
      ]
    },
    ...
  }
}
```

- 3.3. Save the schema of the mobile application manifest

As a result, the [Job experience] detail will display records by the [Job title] column (Fig. 5).

Fig. 5. Case result





#### ATTENTION

To display the columns after set up clean the mobile application cache. You may need to compile the application using the corresponding action in the [Configuration] section.

## Access modifiers of the page in the mobile application

### Difficulty level



The mobile application version 7.11.0 or higher has the ability to configure access modifiers of section or standard detail. For example, you can disable modifying, adding and deleting records for all users in the section.

To set the access in the read only mode, add the following code to the schema which name contains "ModuleConfig":

```
Terrasoft.sdk.Module.setChangeModes("UsrClaim", [Terrasoft.ChangeModes.Read]);
```

Or for the standard detail:

```
Terrasoft.sdk.Details.setChangeModes("UsrClaim", "StandardDetailName", [Terrasoft.ChangeModes.Read]);
```

As a result the adding button will be disabled on the list page and the modifying button will be disabled on the view page. The [Add], [Delete], [Add record to the embedded detail], etc. buttons will be also disabled on the view page.

Access modifiers could be combined. For example, the following code could be used to disable deleting and

enable creating and modifying the records:

```
Terrasoft.sdk.Module.setChangeModes("UsrClaim", [Terrasoft.ChangeModes.Create,  
Terrasoft.ChangeModes.Update]);
```

All access modifiers are given in the *Terrasoft.ChangeModes* enumeration.