Creatio Academy

Mobile application customization

Version 8.0

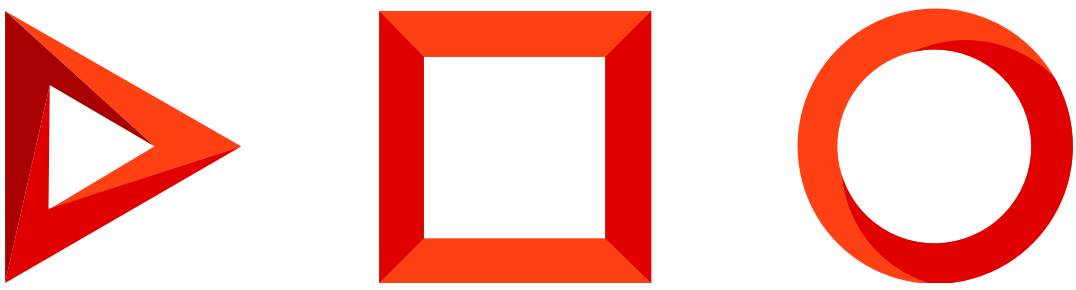


Table of Contents

Mobile application manifest

Advanced

The mobile application manifest describes the structure of the mobile app, its objects and connections between them. The base version of the Mobile Creatio app is described in the manifest located in the `MobileApplicationManifestDefaultWorkplace` 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 `MobileApplicationManifestFieldForceWorkplace`.

Mobile application manifest structure

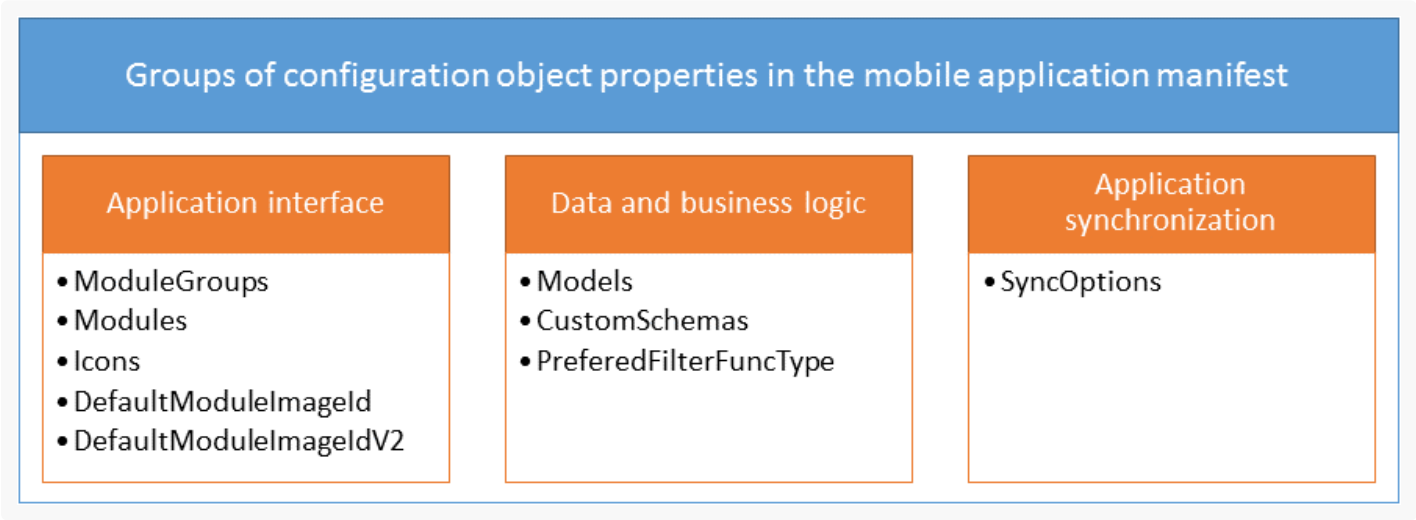
The **mobile application manifest** is a configuration object whose properties describe the structure of the mobile app. Table contains names and descriptions of the mobile application manifest.

Manifest configuration object properties

Property	Purpose
`ModuleGroups`	Describes the properties of the mobile app modules.
`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:

- **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.
- **Data and business logic properties** contain properties where imported data and custom logic is described.
- **Application synchronization properties** contain a single property for synchronization with the primary application.



Application interface

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.

Access modifiers of a page

The mobile application version 7.11.0 or higher has the ability to configure access modificators 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 code to the schema which name contains `ModuleConfig`:

Setting the access in the read only mode

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

Or for the standard detail:

Setting the access in the read only mode for the standard detail

```
Terrasoft.sdk.Details.setChangeModes("UsrClaim", "StandardDetailName", [Terrasoft.ChangeModes.Re
```

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 modificators could be combined. For example, the code could be used to disable deleting and enable creating and modifying the records:

Access modificators combining

```
Terrasoft.sdk.Module.setChangeModes("UsrClaim", [Terrasoft.ChangeModes.Create, Terrasoft.ChangeM
```

All access modificators are given in the `Terrasoft.ChangeModes` enumeration.

Control elements

You can add the following control elements to the section page:

- `Terrasoft.ColumnSet` column groups;
- `Terrasoft.EmbeddedDetail` embedded details;
- standard details;
- inheritor components of the `Terrasoft.RecordPanelItem` class.

Use the mobile application wizard to add the first three types of control elements.

To add an inheritor component of the `Terrasoft.RecordPanelItem` class to the section page:

1. Extend the `Terrasoft.RecordPanelItem` class with a custom class. Define the component configuration object and the functionality methods in the custom class.
2. Create a section settings schema (`Mobile[Section]ModuleConfig`). Using the `addPanelItem()` method of the `Terrasoft.sdk.RecordPage` class in the schema, implement adding the created component to the section page.
3. Add the new custom schemas to the mobile application manifest.

Application data and business logic

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.

Application synchronization

The conditional property group of the manifest configuration object contains a single property used to synchronize data with the main application.

Set up the mobile application menu

Advanced

Example. Setting up the mobile application menu with two groups — the main group and the [*Sales*] group.

Example implementation

The `ModuleGroups` property

```
// 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
    }
}
```

Set up the start page and menu sections in the mobile application

Advanced

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.

Example implementation

The `Modules` property

```
// 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": "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
    }
```

```
}
```

Set up the model configuration

Advanced

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.

Example implementation

The `Models` property

```
// 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"
        ]
    }
}
```

Use the substring search function for data search

Advanced

Example. Use the substring search function for data search.

Example implementation

The `PreferedFilterFuncType` property

```
// 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.

Load model data upon synchronization

Advanced

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.

Example implementation

The `SyncOptions` property

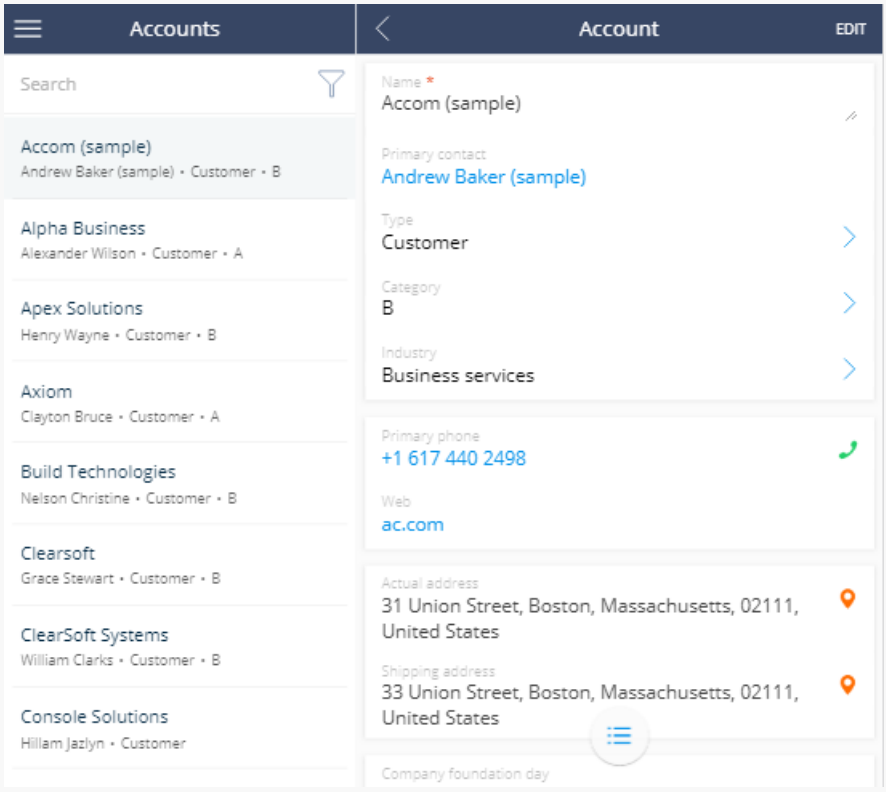
```
// Synchronization settings
"SyncOptions": {
    // The number of pages imported in the same thread.
    "ImportPageSize": 100,
    // The number of import threads.
    "PagesInImportTransaction": 5,
    // Imported system settings array.
    "SysSettingsImportConfig": [
        "SchedulerDisplayTimingStart", "PrimaryCulture", "PrimaryCurrency", "MobileApplicationMo
    ],
    // Imported system lookups array.
    "SysLookupsImportConfig": [
        "ActivityCategory", "ActivityPriority", "ActivityResult", "ActivityResultCategory", "Act
    // 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", "Activity
            ]
        },
        // The ActivityType model is loaded in full.
        {
            "Name": "ActivityType",
```

```
            "SyncColumns": []
        }
    ]
}
```

Display the page in full screen mode on tablets

Easy

If you view Creatio mobile application's section page on a tablet, the section list will be displayed on the left by default.



Example. Activate full screen mode on tablets.

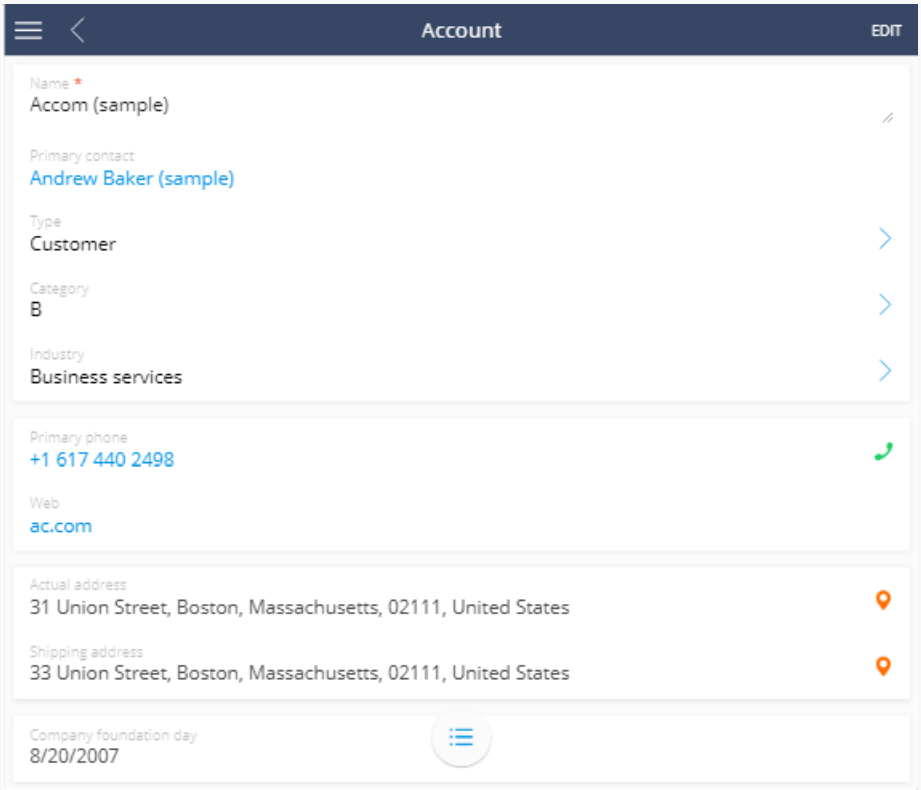
Implement the example

Add the `TabletViewMode` property with the “`SinglePage`” value to the mobile application manifest to display the section page in full screen mode.

The `TabletViewMode` property

```
{
    "TabletViewMode": "SinglePage",
    "CustomSchemas": [],
    "SyncOptions": {
        "SysSettingsImportConfig": [],
        "ModelDataImportConfig": []
    },
    "Modules": {},
    "Models": {}
}
```

After you save the schema and restart the mobile application, the tablet will display the section page in full screen mode.



Add a standard detail with columns

Advanced

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 Mobile Creatio 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.

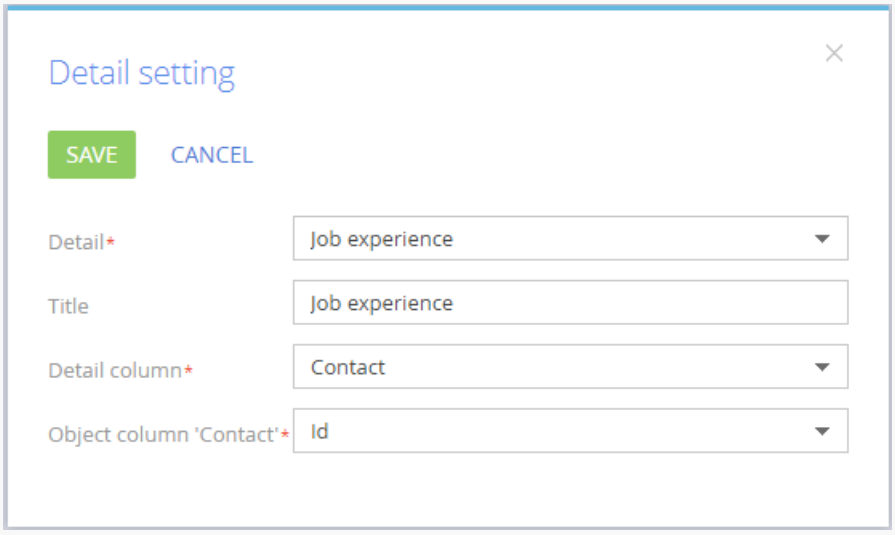
Example. Add the [*Job experience*] detail on the edit page of the [*Contacts*] section of mobile application. Display the [*Job title*] column as primary column.

Example implementation algorithm

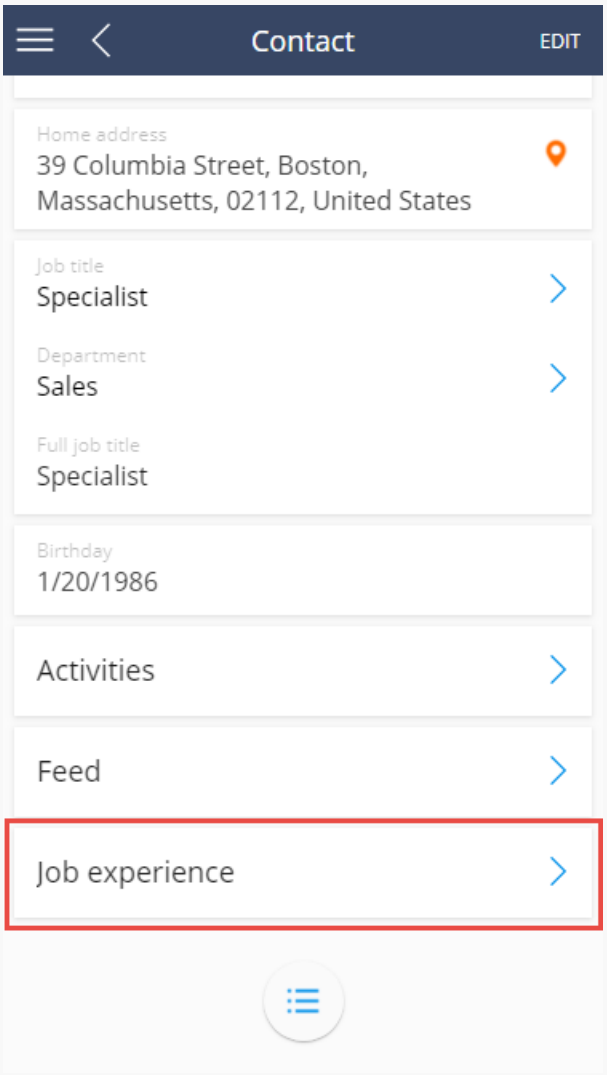
1. Add the [*Job experience*] detail via the mobile application wizard

Use the mobile application wizard to add a detail on the record edit page. To do this:

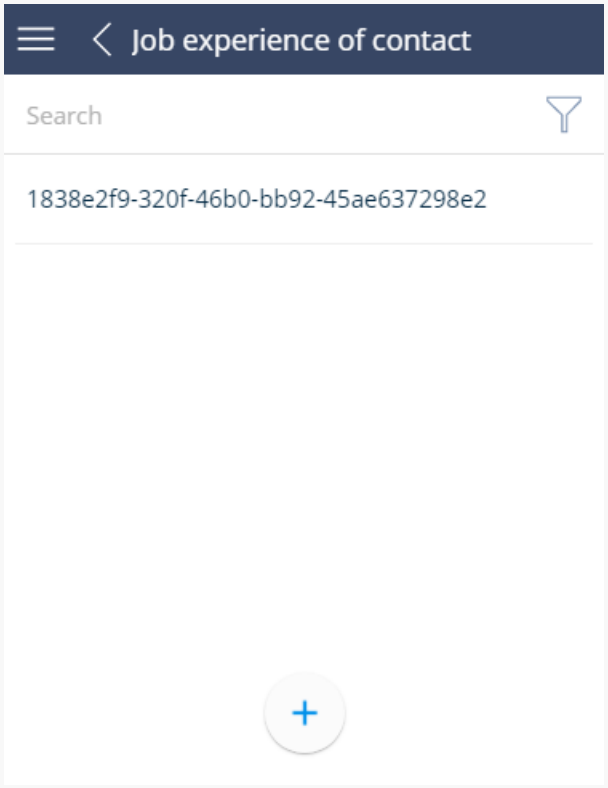
1. Open the necessary workplace (for example [*Main workplace*]) and click the [*Set up sections*].
2. Select the [*Contacts*] section and click the [*Details setup*] button.
3. Set up the [*Job experience*] detail.



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



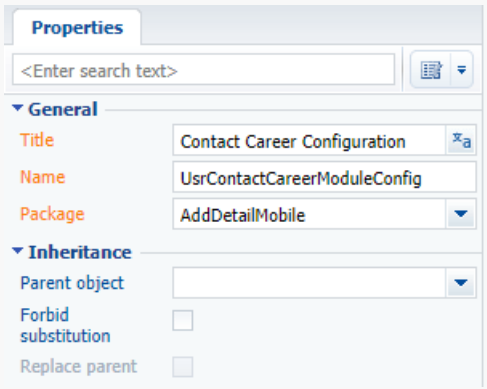
If the [*Job experience*] detail object is not a section object of the Mobile Creatio application, the detail will display the value of the [*Contact*] primary column (id of the connected record of the contact).



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:

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



Add the source code to the module schema:

`UsrContactCareerModuleConfig`

```
// 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 shoul 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:

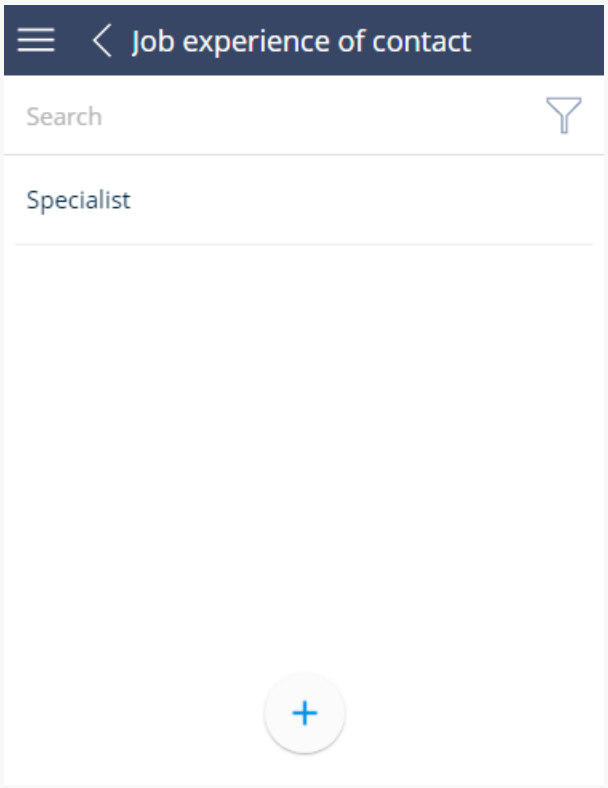
1. Open the schema of the mobile application manifest (`MobileApplicationManifestDefaultWorkplace`) in the custom module designer. This schema is created in the custom package by the mobile application wizard.
2. Add the `UsrContactCareerModuleConfig` module to the `PagesExtensions` section of the `ContactCareer` model.

ContactCareer

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

3. Save the schema of the mobile application manifest.

As a result, the [*Job experience*] detail will display records by the [*Job title*] column.



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.

Add a custom widget to the mobile application

Advanced

The Mobile Creatio 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. 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 Creatio 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.

Example. Add a custom widget to the dashboards page of the mobile application.

Example 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.

CustomDashboardItem

```
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 userConnectio
            : 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.

UsrMobileCustomDashboardItem

```
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 `Terrasoft.DashboardItemClassName` enumeration

Add the source code to the module created on a previous step.

CustomDashboardItem

```
Terrasoft.DashboardItemClassName.CustomDashboardItem = "Terrasoft.configuration.controls.CustomD
```

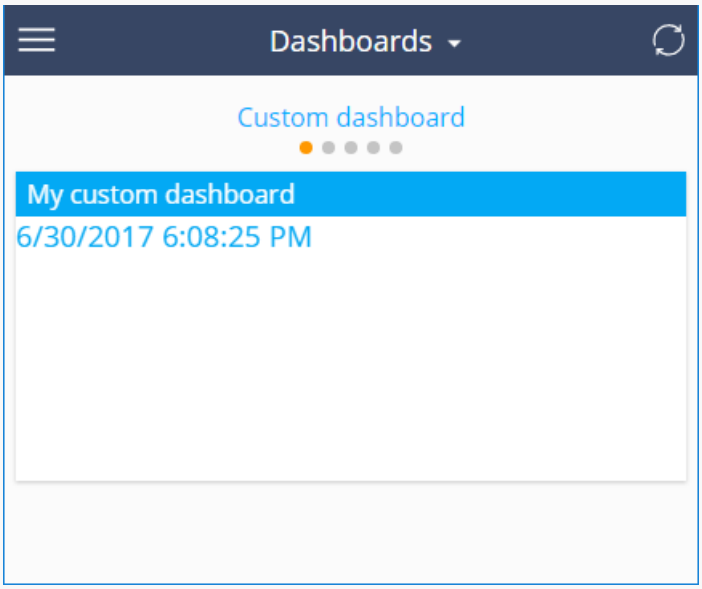
2.3. Add a name of a new custom schema to the mobile application manifest

In the mobile application manifest file, add the name of the created module schema to the `CustomSchemas` array.

CustomSchemas

```
{
    "SyncOptions": {
        ...
    },
    "CustomSchemas": ["UsrMobileCustomDashboardItem"],
    "Modules": {...},
    "Models": {...}
}
```

After saving all changes, the widget will be displayed in the [*Dashboars*] section of the mobile application.



Attention. Add the dashboard widget to the main Creatio application. The mobile application will be synchronized with the main application and the widget will be displayed there.

Add a button to display the name of the contact

Advanced

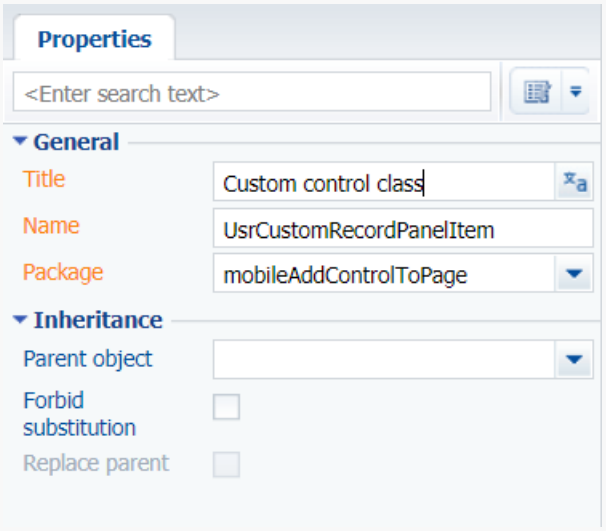
Example. Add a button to the edit page of the [*Contacts*] section of the mobile application. Clicking on the button must trigger a message with the full name of the contact.

Example implementation algorithm

1. Create a custom `Terrasoft.RecordPanelItem` inheritor class

Use the [*Configuration*] section to create a custom module in the custom package with the following properties:

- [*Title*] – “`Custom control class`”.
- [*Name*] – “`UsrCustomRecordPanelItem`”.



Add the source code to the method:

CustomRecordPanelItem

```
Ext.define("Terrasoft.controls.CustomRecordPanelItem", {
    extend: "Terrasoft.RecordPanelItem",
    xtype: "cftestrecordpanelitem",
    config: {
        items: [
            {
                xtype: "container",
                layout: "hbox",
                items: [
                    {
                        xtype: "button",
                        id: "clickMeButton",
                        text: "Full name",
                        flex: 1
                    }
                ]
            }
        ]
    },
    initialize: function() {
        var clickMeButton = Ext.getCmp("clickMeButton");
        clickMeButton.element.on("tap", this.onClickMeButtonClick, this);
    },
    onClickMeButtonClick: function() {
        var record = this.getRecord();
        Terrasoft.MessageBox.showMessage(record.getPrimaryDisplayColumnValue());
    }
});
```

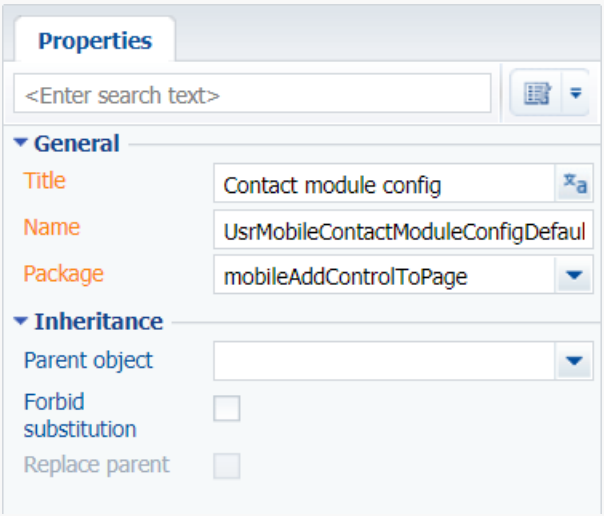
The class defines a configuration object for the created control element and two following methods:

- `initialize()` – button click event handler method.
- `onClickMeButtonClick()` – the method initializes the created element and binds button click events to the handler method.

2. Create module schema in which to configure the section page

Use the [*Configuration*] section to create a custom module in the custom package with the following properties:

- [*Title*] – "`Contact module config`".
- [*Name*] – "`UsrMobileContactModuleConfigDefaultWorkplace`".



Add the source code to the module schema:

`UsrMobileContactModuleConfigDefaultWorkplace`

```
Terrasoft.sdk.RecordPage.addPanelItem("Contact", {
    xtype: "cftestrecordpanelitem",
    position: 1,
    componentConfig: {
    }
});
```

The `addPanelItem()` method of the `Terrasoft.sdk.RecordPage` class is called at this point. The method adds the created control element to the section page.

3. Connect the module schemas in the mobile application manifest

To apply section page settings implemented in the `UsrMobileContactModuleConfigDefaultWorkplace` module:

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.
2. Add the `UsrCustomRecordPanelItem` module to the `CustomSchemas` section, and the `UsrContactCareerModuleConfig` module to the `PagesExtensions` section of the `Contact` model:

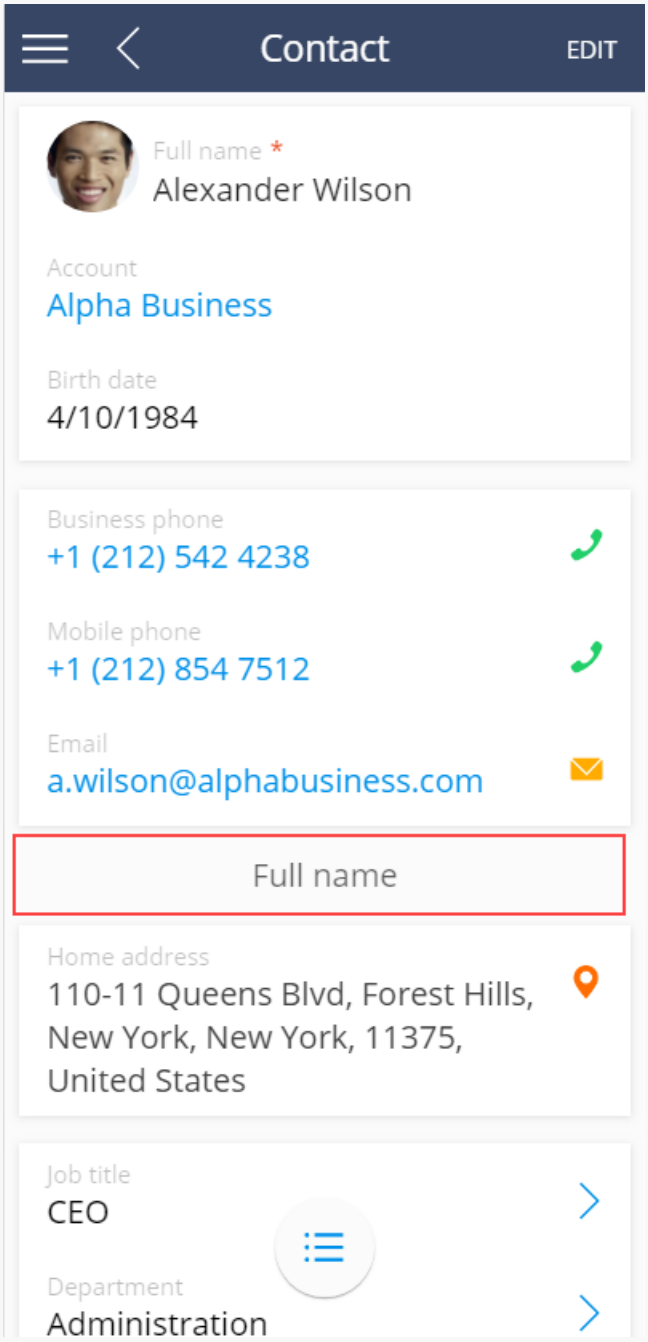
Modules adding

```
{
    "CustomSchemas": [
        "UsrCustomRecordPanelItem.js"
    ],
    "SyncOptions": {},
    "Modules": {},
    "Models": {
        "Contact": {
            "RequiredModels": [],
            "ModelExtensions": [],
            "PagesExtensions": [
                "UsrMobileContactModuleConfigDefaultWorkplace.js"
            ]
        }
    }
}
```

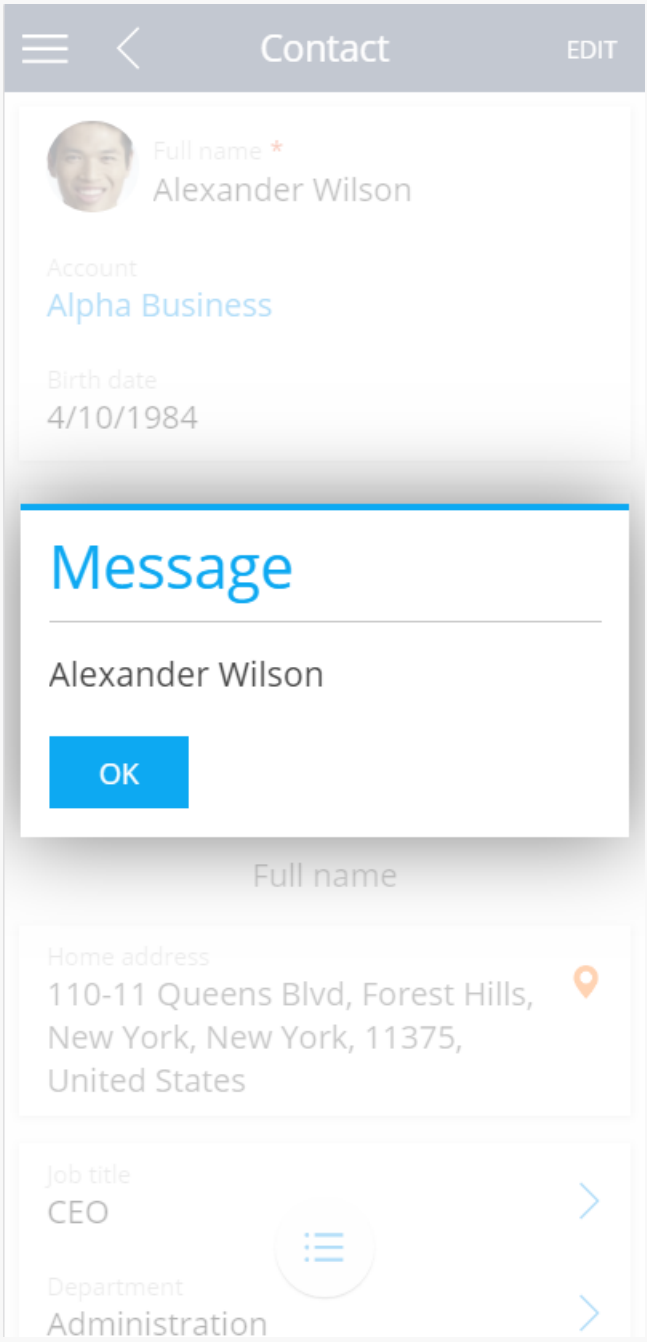
3. Save the schema of the mobile application manifest.

As a result, the contact page will have a control element. A click on the element will trigger a message with the full name of the contact.

Case result. Adding a button



Case result. Click the button



ModuleGroups property JS

Advanced

Application module groups. Uses for for the menu group setup. 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.

The configuration object property

`Position`

Group position in the main menu. Strats with 0.

Modules property JS

Advanced

A mobile application module is an application section. Each module in the `Modules` configuration object manifest describes a configuration object with properties given in table. The name of the configuration section object must match the name of the model that provides section data.

Configuration object properties

`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.

`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`.

Icons property JS

Advanced

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.

Configuration object properties

`ImageListId`

Image list ID.

`ImageId`

Custom image ID from the `ImageListId` list.

Use of custom images

```
// 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 JS

Advanced

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.

Installation of unique image identifiers

```
// Custom interface V1 default image ID.
```

```
"DefaultModuleImageId": "423d3be8-de6b-4f15-a81b-ed454b6d03e3",
// Custom interface V2 default image ID.
"DefaultModuleImageIdV2": "1c92d522-965f-43e0-97ab-2a7b101c03d4"
```

Models property

Advanced

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.

Configuration object properties

`Grid`

Model list page schema name. The page will be generated automatically with the following name: `Mobile[Model_name][Page_type]Page`.

`Preview`

Preview page schema name for model element. The page will be generated automatically with the following name: `Mobile[Model_name][Page_type]Page`.

`Edit`

Edit page schema name for model element. The page will be generated automatically with the following name: `Mobile[Model_name][Page_type]Page`.

`RequiredModels`

Names of the models that the current model depends on. 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. An array of schemas, where additional model settings are implemented (adding business rules, events, default values, etc.).

`PagesExtensions`

Model page extensions. An array of schemas where additional settings for various page types are implemented (adding details, setting titles, etc.).

PreferedFilterFuncType property

Advanced

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 `Terrasoft.FilterFunctions` enumeration. The list of filtering functions is available in table.

Filtering functions (`Terrasoft.FilterFunctions`)

`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.

CustomSchemas property JS

Advanced

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.

Connect additional custom schemas for registering actions and utilities

```
// Connect additional custom schemas.
"CustomSchemas": [
    // Custom action registration schema.
    "MobileActionCheckIn",
    // Custom utility schema.
    "CustomMobileUtilities"
]
```

SyncOptions property JS

Advanced

Describes the options for configuring data synchronization. Contains the configuration object with properties presented in table.

The configuration object properties for the synchronization setup

---

`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 is added to the `ModelDataImportConfig` array.

The configuration object properties for the synchronization model setup

`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.

Filter model configuration object properties

`type`

Filter type. Set by the enumeration value `Terrasoft.FilterTypes`. Optional property. By default `Terrasoft.FilterTypes.Simple`.

Possible values (`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`.

Possible values (`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.

Possible values (`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.

Possible values (`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`.

Possible values (`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`.

Possible values (`Terrasoft.ComparisonType`)

`Equal`	equal
`LessOrEqual`	less or equal
`NotEqual`	not equal
`Greater`	greater
`GreaterOrEqual`	greater or equal
`Less`	less

The `SyncOptions.ModelDataImportConfig.QueryFilter` property

Available in the application starting with version 7.12.1 and in the 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 service.

Attention. Data import via the DataService service is available only for the Android and iOS platforms. The OData 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.

Example of the exists filter

```
{
  "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,
                                "macrosType": 2
                             }
                          }
                       }
                    }
                 }
              }
           }
        }
     ]
  }
}
```

Mobile application list

Mobile application list

Advanced

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

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

Set up the section list

Advanced

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.

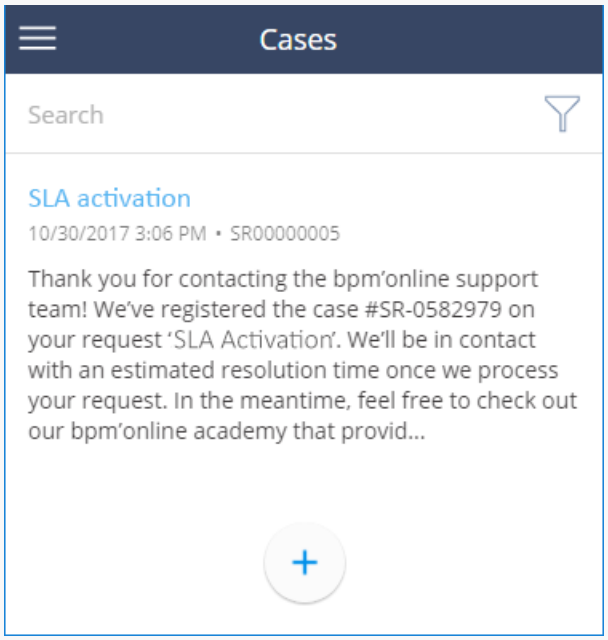
Example implementation

Use the following source code to configure the list.

Configure the list of the[*Cases*] section

```
// 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:



GridPage class JS

Advanced

Class uses to configure list layout, sorting, search logic, etc.

Methods

`setPrimaryColumn(modelName, column)`

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

Parameters

`modelName`	model name
`column`	column name

Example of call

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

`setSubtitleColumns(modelName, columns)`

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

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"]);
```

Example of call (option 2)

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

}]);
```

`setGroupColumns(modelName, columns)`

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

Parameters

`modelName`	model name
`columns`	can array of columns or column configuration objects

Example of call (option 1)

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

Example of call (option 2)

```
Terrasoft.sdk.GridPage.setGroupColumns("Case", [{
    name: "Symptoms",
    isMultiline: true,
```

```
        label: "CaseGridSymptomsColumnLabel",
        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.

Business rules in mobile application

 Advanced

Business rules represent a Creatio mechanism that enables setting up the behavior of record edit page fields. You can use business rules to, e.g., set up visible or required fields, make fields enabled, etc.

Attention. Business rules work only on record edit and view pages.

Adding business rules to a page is performed via the `Terrasoft.sdk.Model.addBusinessRule(name, config)` method, where

- `name` – is the name of the model, bound to the edit page, e.g., “`Contact`”.
- `config` – is the object defining business rule properties. The list of properties depends on a specific business rule type.

In the mobile application you can add business rule that implements custom logic (custom business rule). The `Terrasoft.RuleTypes.Custom` method is provided for this type of business rules.

Filter values

Advanced

Example 1

Example. Filter values in a column by condition.

When selecting a value in the [*Product*] lookup column, only the products containing the `true` value in the [*Active*] column of the [*Product in invoice*] detail are available.

Example implementation

Filtration example

```
Terrasoft.sdk.Model.addBusinessRule("InvoiceProduct", {
    ruleType: Terrasoft.RuleTypes.Filtration,
    events: [Terrasoft.BusinessRuleEvents.Load],
    triggeredByColumns: ["Product"],
    filters: Ext.create("Terrasoft.Filter", {
        modelName: "Product",
        property: "Active",
        value: true
    })
});
```

Example 2

Example. Filter values in a column by the value in another column.

The [*Contact*] field on the record edit page of the [*Invoices*] section should be filtered based on the [*Account*] field value.

Example implementation

Filtration example

```
Terrasoft.sdk.Model.addBusinessRule("Invoice", {
    ruleType: Terrasoft.RuleTypes.Filtration,
    events: [Terrasoft.BusinessRuleEvents.Load, Terrasoft.BusinessRuleEvents.ValueChanged],
    triggeredByColumns: ["Account"],
    filteredColumn: "Contact",
    filters: Ext.create("Terrasoft.Filter", {
        property: "Account"
    })
});
```

Example 3

Example. Add and delete filtration by custom logic.

Example implementation

Filtration example

```
Terrasoft.sdk.Model.addBusinessRule("Activity", {
    name: "ActivityResultByAllowedResultFilterRule",
    position: 1,
    ruleType: Terrasoft.RuleTypes.Custom,
    triggeredByColumns: ["Result"],
    events: [Terrasoft.BusinessRuleEvents.ValueChanged, Terrasoft.BusinessRuleEvents.Load],
    executeFn: function(record, rule, column, customData, callbackConfig) {
        var allowedResult = record.get("AllowedResult");
        var filterName = "ActivityResultByAllowedResultFilter";
        if (!Ext.isEmpty(allowedResult)) {
            var allowedResultIds =  Ext.JSON.decode(allowedResult, true);
            var resultIdsAreCorrect = true;
            for (var i = 0, ln = allowedResultIds.length; i < ln; i++) {
                var item = allowedResultIds[i];
                if (!Terrasoft.util.isGuid(item)) {
                    resultIdsAreCorrect = false;
                    break;
                }
            }
            if (resultIdsAreCorrect) {
                var filter = Ext.create("Terrasoft.Filter", {
                    name: filterName,
                    property: "Id",
```

```
                    funcType: Terrasoft.FilterFunctions.In,
                    funcArgs: allowedResultIds
                });
                record.changeProperty("Result", {
                    addFilter: filter
                });
            } else {
                record.changeProperty("Result", {
                    removeFilter: filterName
                });
            }
        } else {
            record.changeProperty("Result", {
                removeFilter: filterName
            });
        }
        Ext.callback(callbackConfig.success, callbackConfig.scope, [true]);
    }
});
```

Select a field by condition

Advanced

Example. Highlight the field with the result of the activity, if its status is "`Completed`", the [*Result*] field is not filled and the [*ProcessElementId*] column has a value.

Example implementation

Highlight the field by condition

```
// Rule for the activity edit page.
Terrasoft.sdk.Model.addBusinessRule("Activity", {
    // The name of the business rule.
    name: "ActivityResultRequiredByStatusFinishedAndProcessElementId",
    // Business rule type: custom.
    ruleType: Terrasoft.RuleTypes.Custom,
    //The rule is initiated by the Status and Result columns.
    triggeredByColumns: ["Status", "Result"],
    // The rule will work before saving the data and after changing the data.
    events: [Terrasoft.BusinessRuleEvents.ValueChanged, Terrasoft.BusinessRuleEvents.Save],
    // Handler function.
    executeFn: function(record, rule, column, customData, callbackConfig) {
        // A flag of the validity of the property and the rule.
```

```
        var isValid = true;
        // The value of the ProcessElementId column.
        var processElementId = record.get("ProcessElementId");
        // If the value is not empty.
        if (processElementId && processElementId !== Terrasoft.GUID_EMPTY) {
            // Set the validity flag.
            isValid = !(record.get("Status.Id") === Terrasoft.Configuration.ActivityStatus.Finis
                Ext.isEmpty(record.get("Result")));
        }
        // Change the properties of the Result column.
        record.changeProperty("Result", {
            // Set the column correctness indicator.
            isValid: {
                value: isValid,
                message: Terrasoft.LS["Sys.RequirementRule.message"]
            }
        });
        // Asynchronous return of values.
        Ext.callback(callbackConfig.success, callbackConfig.scope, [isValid]);
    }
});
```

Reset negative values to 0

Advanced

Example. Implement the logic for dropping negative values to 0.

Example implementation

Reset negative values to 0

```
Terrasoft.sdk.Model.addBusinessRule("Opportunity", {
    name: "OpportunityAmountValidatorRule",
    ruleType: Terrasoft.RuleTypes.Custom,
    triggeredByColumns: ["Amount"],
    events: [Terrasoft.BusinessRuleEvents.ValueChanged, Terrasoft.BusinessRuleEvents.Save],
    executeFn: function(model, rule, column, customData, callbackConfig) {
        var revenue = model.get("Amount");
        if ((revenue < 0) || Ext.isEmpty(revenue)) {
            model.set("Amount", 0, true);
        }
        Ext.callback(callbackConfig.success, callbackConfig.scope);
    }
```

```
});
```

Generate the title of an activity

Advanced

Example. Implement generating the activity header for the FieldForce solution.

Example implementation

Generating the activity header

```
Terrasoft.sdk.Model.addBusinessRule("Activity", {
    name: "FieldForceActivityTitleRule",
    ruleType: Terrasoft.RuleTypes.Custom,
    triggeredByColumns: ["Account", "Type"],
    events: [Terrasoft.BusinessRuleEvents.ValueChanged, Terrasoft.BusinessRuleEvents.Load],
    executeFn: function(record, rule, column, customData, callbackConfig, event) {
        if (event === Terrasoft.BusinessRuleEvents.ValueChanged || record.phantom) {
            var type = record.get("Type");
            var typeId = type ? type.get("Id") : null;
            if (typeId !== Terrasoft.Configuration.ActivityTypes.Visit) {
                Ext.callback(callbackConfig.success, callbackConfig.scope, [true]);
                return;
            }
            var account = record.get("Account");
            var accountName = (account) ? account.getPrimaryDisplayColumnValue() : "";
            var title = Ext.String.format("{0}: {1}", Terrasoft.LocalizableStrings.FieldForceTit
            record.set("Title", title, true);
        }
        Ext.callback(callbackConfig.success, callbackConfig.scope, [true]);
    }
});
```

config object property C#

Advanced

The base business rule

The base business rule is an abstract class, i.e., all business rules should be its inheritors.

The properties of the `config` configuration object that can be used by the inheritors of the business rule.

Configuration object properties

`ruleType`

The type of rule. The value must be included into the `Terrasoft.RuleTypes` enumeration.

`triggeredByColumns`

The column array that triggers the rule.

`message`

A text message displayed under the control element connected with the column in case business rule is not executed. It is necessary for rules that inform a user of warnings.

`name`

A unique name of a business rule. It is necessary if you need to delete a rule by the `Terrasoft.sdk` methods.

`position`

A position of a business rule that defines its order priority in the current queue.<

`events`

An event array, defining the time of running business rules. It should contain values included into the `Terrasoft.BusinessRuleEvents` enumeration.<

Possible values (`Terrasoft.BusinessRuleEvents`)

`Save`	the rule is executed before saving the data
`ValueChanged`	the rule is executed when the data is modified (while editing)
`Load`	the rule is executed when the edit page is opened

The [*Is required*] business rule (`Terrasoft.RuleTypes.Requirement`) C#

Defines whether an edit page field is required.

Configuration object properties

`ruleType`

Should contain the `Terrasoft.RuleTypes.Requirement` value for this rule.

`requireType`

Verification type. The value must be included into the `Terrasoft.RequirementTypes` enumeration. The rule can verify one or all the columns from `triggeredByColumns`.

`triggeredByColumns`

The column array that triggers the rule. If the verification type equals `Terrasoft.RequirementTypes.Simple`, one column in the array should be specified.

Possible values (`Terrasoft.RequirementTypes`)

`Simple`	value verification in one column
`OneOf`	one of the columns specified in the `triggeredByColumns` should be populated

Use case

```
Terrasoft.sdk.Model.addBusinessRule("Contact", {
    ruleType: Terrasoft.RuleTypes.Requirement,
    requireType : Terrasoft.RequirementTypes.OneOf,
    events: [Terrasoft.BusinessRuleEvents.Save],
    triggeredByColumns: ["HomeNumber", "BusinessNumber"],
    columnNames: ["HomeNumber", "BusinessNumber"]
});
```

The [*Visibility*] business rule (`Terrasoft.RuleTypes.Visibility`) C#

You can hide and display fields per condition using this rule.

Configuration object properties

`ruleType`

Should contain the `Terrasoft.RuleTypes.Visibility` value for this rule.

`triggeredByColumns`

The column array that triggers the rule.

`events`

An event array, defining the time of running business rules. It should contain values included into the `Terrasoft.BusinessRuleEvents` enumeration.

`conditionalColumns`

Condition array of business rule execution. Usually, these are specific column values.

`dependentColumnNames`

Column name array that the business rule is applied to.

Use case

```
Terrasoft.sdk.Model.addBusinessRule("Account", {
    ruleType: Terrasoft.RuleTypes.Visibility,
    conditionalColumns: [
        {name: "Type", value: Terrasoft.Configuration.Consts.AccountTypePharmacy}
    ],
    triggeredByColumns: ["Type"],
    dependentColumnNames: ["IsRx", "IsOTC"]
});
```

The fields connected with the `IsRx` and `IsOTC` columns are displayed if the `Type` column contains the value defined by the `Terrasoft.Configuration.Consts.AccountTypePharmacy` invariable.

```
Terrasoft.Configuration.Consts = {
    AccountTypePharmacy: "d12dc11d-8c74-46b7-9198-5a4385428f9a"
};
```

You can use the 'd12dc11d-8c74-46b7-9198-5a4385428f9a' value instead of the invariable.

The [*Enabled/Disabled*] business rule (`Terrasoft.RuleTypes.Activation`) C#

This business rule enables and disables fields for entering values per condition.

Configuration object properties

`ruleType`

Should contain the `Terrasoft.RuleTypes.Activation` value for this rule.<

`triggeredByColumns`

The column array that triggers the rule.

`events`

An event array, defining the time of running business rules. It should contain values included into the `Terrasoft.BusinessRuleEvents` enumeration.

`conditionalColumns`

Condition array of business rule execution. Usually, these are specific column values.

`dependentColumnNames`

Column name array that the business rule is applied to.

Whether a field connected with the `Stock` column is enabled depends on the value in the `IsPresence` column.

Use case

```
Terrasoft.sdk.Model.addBusinessRule("ActivitySKU", {
    ruleType: Terrasoft.RuleTypes.Activation,
    events: [Terrasoft.BusinessRuleEvents.Load, Terrasoft.BusinessRuleEvents.ValueChanged],
    triggeredByColumns: ["IsPresence"],
    conditionalColumns: [
        {name: "IsPresence", value: true}
    ],
    dependentColumnNames: ["Stock"]
});
```

The [*Filtration*] business rule (`Terrasoft.RuleTypes.Filtration`)

This business rule can be used for filtration of lookup columns by condition, or by another column value.

Configuration object properties

`ruleType`

Should contain the `Terrasoft.RuleTypes.Filtration` value for this rule.

`triggeredByColumns`

The column array that triggers the rule.

`events`

An event array, defining the time of running business rules. It should contain values included into the `Terrasoft.BusinessRuleEvents` enumeration.

`filters`

Filter. The property should contain the `Terrasoft.Filter` class instance.

`filteredColumn`

The column used for filtering values.

The [*Mutual Filtration*] business rule (`Terrasoft.RuleTypes.MutualFiltration`) C#

This business rule enables mutual filtering of two lookup fields. Works only with columns with the “one-to-many” relationship, e.g., [*Country*] – [*City*]. Create a separate business rule for every field cluster. For example, for the [*Country*] – [*Region*] – [*City*] and the [*Country*] – [*City*] clusters, create three business rules:

- [*Country*] – [*Region*];
- [*Region*] – [*City*];
- [*Country*] – [*City*].

Configuration object properties

`ruleType`

Should contain the `Terrasoft.RuleTypes.MutualFiltration` value for this rule.

`triggeredByColumns`

The column array that triggers the rule.

`connections`

Object array that configures cluster relationship.

Mutual filtration of the [Country], [Region] and [City] fields

```
Terrasoft.sdk.Model.addBusinessRule("ContactAddress", {
    ruleType: Terrasoft.RuleTypes.MutualFiltration,
    triggeredByColumns: ["City", "Region", "Country"],
    connections: [
        {
            parent: "Country",
            child: "City"
        },
        {
            parent: "Country",
            child: "Region"
        },
        {
            parent: "Region",
            child: "City"
        }
    ]
});
```

Mutual filtration of the [Contact], [Account] fields

```
Terrasoft.sdk.Model.addBusinessRule("Activity", {
    ruleType: Terrasoft.RuleTypes.MutualFiltration,
    triggeredByColumns: ["Contact", "Account"],
    connections: [
        {
            parent: "Contact",
            child: "Account",
            connectedBy: "PrimaryContact"
        }
    ]
});
```

The [*Regular expression*] business rule (`Terrasoft.RuleTypes.RegExp`) C#

Verifies the conformity of the column value with the regular expression.

Configuration object properties

`ruleType`

Should contain the `Terrasoft.RuleTypes.RegExp` value for this rule.

`RegExp`

Regular expression whose conformity with all the `triggeredByColumns` array columns is verified.

`triggeredByColumns`

The column array that triggers the rule.

Use case

```
Terrasoft.sdk.Model.addBusinessRule("Contact", {
    ruleType: Terrasoft.RuleTypes.RegExp,
    regExp : /^([0-9\(\)\/\+ \-]*)$/
    triggeredByColumns: ["HomeNumber", "BusinessNumber"]
});
```

Custom business rules

When adding a custom business rule via the `Terrasoft.sdk.Model.addBusinessRule(name, config)` method you can use properties of the `config` configuration object of the base business rule. In addition, the `executeFn` property is also provided.

Configuration object properties

`ruleType`

Rule type. For the custom rules it should contain the `Terrasoft.RuleTypes.Custom` value.

`triggeredByColumns`

Array of columns which initiates trigging of the business rule.

`events`

Array of events determining the start time of the business rule. It should contain values from the

`Terrasoft.BusinessRuleEvents` enumeration. Default value: `Terrasoft.BusinessRuleEvents.ValueChanged`.

Possible values (`Terrasoft.BusinessRuleEvents`)

`Save`	the rule trigs before saving the data
`ValueChanged`	the rule trigs after changing the data (at modification)
`Load`	the rule trigs when the edit page is opened

`executeFn`

A handler function that contains the user logic for executing the business rule.

Properties of the `executeFn` handler function

Handler function signature

```
executeFn: function(record, rule, checkColumnName, customData, callbackConfig, event) {    }
```

Parameters

`record`	a record for which the business rule is executed
`rule`	an instance of the current business rule
`checkColumnName`	a column name that calls business-rules firing
`customData`	an object that is shared between all rules. Not used. Left for compatibility with previous versions
`callbackConfig`	a configuration object of the `Ext.callback` asynchronous callback
`event`	an event that triggered the business rul.

After the completion of function operation it is necessary to call either the `callbackConfig.success` or `callbackConfig.failure`.

Use cases options

```
Ext.callback(callbackConfig.success, callbackConfig.scope, [result]);
Ext.callback(callbackConfig.failure, callbackConfig.scope, [exception]);
```

Where:

- `result` – the returned boolean value obtained when the function is executed (`true` / `false`).
- `exception` – the exception of the `Terrasoft.Exception` type, which occurred in the handler function.

Methods

In the source code of the handler function, you can use the following methods of the model passed in the `record` parameter:

`get(columnName)`

Po get the value of a record column. The `columnName` argument should contain the column name.

`set(columnName, value, fireEventConfig)`

To set the value of the record column.

Parameters

`columnName`	the name of the column
`value`	the value assigned to the column
`fireEventConfig`	a configuration object to set the properties that are passed to the column modification event

---

`changeProperty(columnName, propertyConfig)`

For changing column properties except its value. The `columnName` argument should contain the column name and the `propertyConfig` object that sets the column properties.

The `propertyConfig` object properties

`disabled`	activity of the column. If `true`, the control associated with the column will be inactive and disabled for operation
`readOnly`	“Read only” flag. If `true`, the control associated with the column will be available only for reading. If `false` – the access for reading and writing
`hidden`	column visibility. If `true`, the control associated with the column will be hidden. If `false` – the control will be displayed
`addFilter`	add filter. If the property is specified, it should have a filter of the `Terrasoft.Filter` type that will be added to the column filtration. Property is used only for lookup fields
`removeFilter`	remove the filter. If the property is specified, it should have a name of the filter that will be removed from the column filtration. Property is used only for lookup fields
`isValid`	flag of column validity. If the property is specified, it will change the validity flag of the control associated with the column. If the column is invalid, then this can mean canceling of saving the record, and can also lead to the determining the record as invalid

Example of changing the properties (but not the values) of the `Owner` column

```
record.changeProperty("Owner", {
    disabled: false,
    readOnly: false,
    hidden: false,
    addFilter: {
        property: "IsChief",
        value: true
    },
    isValid: {
        value: false,
        message: LocalizableStrings["Owner_should_be_a_chief_only"]
    }
 });
```

Getting the settings and data from the [Dashboard] section

Advanced

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

Sample requests to the AnalyticsService service

Advanced

Request headers

```
Accept:application/json
```

Methods

`GetDashboardViewConfig()`

Request

```
POST /0/rest/AnalyticsService/GetDashboardViewConfig

{
    "id": "a71d5c04-dff7-4892-90e5-9e7cc2246915"
}
```

Response

```
{
  "items": [
    {
      "layout": {
        "column": 0,
        "row": 0,
        "colSpan": 12,
        "rowSpan": 5
      },
      "name": "Chart4",
      "itemType": 4,
      "widgetType": "Chart"
    }
  ]
}
```

GetDashboardData()

Request

```
POST /0/rest/AnalyticsService/GetDashboardData

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

Response

```
{
    "items": [
        {
            "name": "Indicator1",
            "caption": "Average time for activity",
            "widgetType": "Indicator",
            "style": "widget-green",
            "data": 2
        }
    ]
}
```

GetDashboardItemData()

Request

```
POST /0/rest/AnalyticsService/GetDashboardItemData

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

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": []
}
```


AnalyticsService class C#

Advanced

Class implements the functionality of getting settings and the dashboards data.

Methods

`Stream GetDashboardViewConfig(Guid id)`

Returns the settings of a view and widgets on the dashboards tab by the dashboard page Id.

`Stream GetDashboardData(Guid id, int timeZoneOffset)`

Returns the data from all widgets on the dashboards tab by the dashboard page Id.

`Stream GetDashboardItemData(Guid dashboardId, string itemName, int timeZoneOffset)`

Returns data from athe specific widget by the dashboard page Id and the widget name.

`timeZoneOffset` – the time zone offset (in minutes) from the UTC. Dashboards data will be received using this time zone.

Mobile portal

Beginner

Mobile portal (mobile application for portal users) is a mobile workplace. The **purpose** of the mobile portal is to enable the mobile portal users to create cases and communicate with customer support.

A mobile portal has **configurable**:

- mobile portal user workplace
- case list
- case page
- page that adds cases

Add base package schema to the custom package

If you are yet to perform the setup using the Mobile Application Wizard, the base package schema might not be available in the custom package.

To **add base package schema to the custom package**:

1. Click to open the System Designer.
2. Go to the [*System setup*] block → [*Mobile application wizard*].
3. Open the [*Portal*] workplace in the section list.
4. Click [*Set up sections*] on the toolbar.
5. Select the [*Cases*] section in the section list and click [*Page setup*].
6. Save the settings of the [*Cases*] section page.
7. Save the settings of the [*Mobile application wizard*] section.

Set up the workplace of a mobile portal user

You can set up the workplace of a mobile portal user in the following **ways**:

- Add a new workplace.
- Hide a workplace.

- Delete a workplace.

Add a workplace of a mobile portal user

To **check if the [*Portal*] workplace** is available:

1. Click to open the System Designer.
2. Go to the [*System setup*] block → [*Mobile application wizard*].

The [*Portal*] workplace is in the [*Mobile application wizard*] section. By default, all mobile portal users can access the workplace.

If the [*Portal*] workplace is not available in the [*Mobile application wizard*] section, add the workplace.

To **add a workplace of a mobile portal user**:

1. Make sure that your Creatio application includes the mobile portal functionality.
2. Click to open the System Designer.
3. Go to the [*System setup*] block → [*Mobile application wizard*].
4. Click [*New workplace*] in the [*Mobile application wizard*] section toolbar.
5. Fill out the **workplace properties**.
   - Set [*Name*] to the workplace name.
   - Set [*Code*] to "`Portal`."
6. Configure the access permissions to the workplace for users or user groups on the [*Roles*] detail. Learn more in user documentation: Object operation permissions.
7. Click [*Set up sections*] on the toolbar. By default, the workplace of a mobile portal user includes the [*Cases*] section.
8. Save the settings of the [*Mobile application wizard*] section.

As a result, Creatio will add a workplace of a mobile portal user.

Learn more about adding a workplace to a mobile application in user documentation: Set up mobile app workplaces.

Hide the workplace of a mobile portal user

1. Click to open the System Designer.
2. Go to the [*System setup*] block → [*Mobile application wizard*].
3. Open the [*Portal*] workplace in the section list.
4. Delete users or user groups of the [*Portal*] workplace. To do this, click and select [*Delete*] on the [*Roles*] detail.

As a result, Creatio will hide the workplace of a mobile portal user.

Delete the workplace of a mobile portal user

1. Click to open the System Designer.
2. Go to the [*System setup*] block → [*Mobile application wizard*].
3. Select the [*Portal*] workplace in the section list and click [*Delete*].

As a result, Creatio will delete the workplace of a mobile portal user.

Set up the case list

You can set up the case list of a mobile portal in the following **ways**:

- Add a column to the case list.
- Make a case list column searchable.
- Hide the column title from the case list.
- Change the case sorting order in the list.

Add a column to the case list

1. Click to open the System Designer.
2. Go to the [*System setup*] block → [*Mobile application wizard*].
3. Open the [*Portal*] workplace in the section list.
4. Click [*Set up sections*] on the toolbar.
5. Select the [*Cases*] section in the section list and click [*List setup*].
6. Click the [*New column*] button in the [*Subtitle*] or [*Additional columns*] block and select the required column.
7. Save the list settings in the [*Cases*] section.
8. Save the settings of the [*Mobile application wizard*] section.

As a result, Creatio will add a column to the case list. However, the column will not be searchable. To **make the case list column searchable**, follow the instructions in different section: Add a searchable column to the case list.

Learn more about adding a column to the section list in user documentation: Set up mobile application section list .

Make a case list column searchable

[*Number*] and [*Description*] columns of the case list are searchable. You can make other columns searchable as well. To do this, add the columns to the `MobileCaseGridPageSettingsPortal` schema.

To **make a case list column searchable**:

1. Go to the [*Configuration*] section.
2. Open the `MobileCaseGridPageSettingsPortal` schema in the custom package. If you are yet to set up the case list using the Mobile Application Wizard, the `MobileCaseGridPageSettingsPortal` schema will not be available in the custom package. To **add** the `MobileCaseGridPageSettingsPortal` schema **to the custom package**, follow

the instructions in a different section: Add a base package schema to the custom package.

3. Make a **case list column searchable**.

   For Creatio version **8.0.2 and later**

   For Creatio version **7.18.4-8.0.1**

4. Click [*Save*] on the Module Designer's toolbar.

As a result, [*Number*], [*Description*], and [*Subject*] columns of the case list will be searchable.

Hide the column title from the case list

1. Go to the [*Configuration*] section.
2. Open the `MobileCaseGridPageSettingsPortal` schema in the custom package. If you are yet to set up the case list using the Mobile Application Wizard, the `MobileCaseGridPageSettingsPortal` schema will not be available in the custom package. To **add** the `MobileCaseGridPageSettingsPortal` schema **to the custom package**, follow the instructions in a different section: Add base package schema to the custom package.
3. Hide the **column title from the case list**.

   For Creatio version **8.0.2 and later**

   For Creatio version **7.18.4-8.0.1**

4. Click [*Save*] on the Module Designer's toolbar.

As a result, Creatio will hide the column title from the case list.

Change the case sorting order in the list

1. Go to the [*Configuration*] section.
2. Open the `MobileCaseGridPageSettingsPortal` schema in the custom package. If you are yet to set up the case list using the Mobile Application Wizard, the `MobileCaseGridPageSettingsPortal` schema will not be available in the custom package. To **add** the `MobileCaseGridPageSettingsPortal` schema **to the custom package**, follow the instructions in a different section: Add base package schema to the custom package.
3. Change the **sorting order in the case list**.

   For Creatio version **8.0.2 and later**

   For Creatio version **7.18.4-8.0.1**

4. Click [*Save*] on the Module Designer's toolbar.

As a result, Creatio will display cases sorted in the specified order in the list.

Set up the case page

Set up the case page to add a column to the [*Details*] tab.

To **add a column to the [*Details*] tab** of the case page:

1. Click to open the System Designer.
2. Go to the [*System setup*] block → [*Mobile application wizard*].
3. Open the [*Portal*] workplace in the section list.
4. Click [*Set up sections*] on the toolbar.
5. Select the [*Cases*] section in the section list and click [*Page setup*].
6. Click [*New column*] in the [*General information*] block and select the [*Number*] column.
7. Save the settings of the [*Cases*] section page.
8. Save the settings of the [*Mobile application wizard*] section.

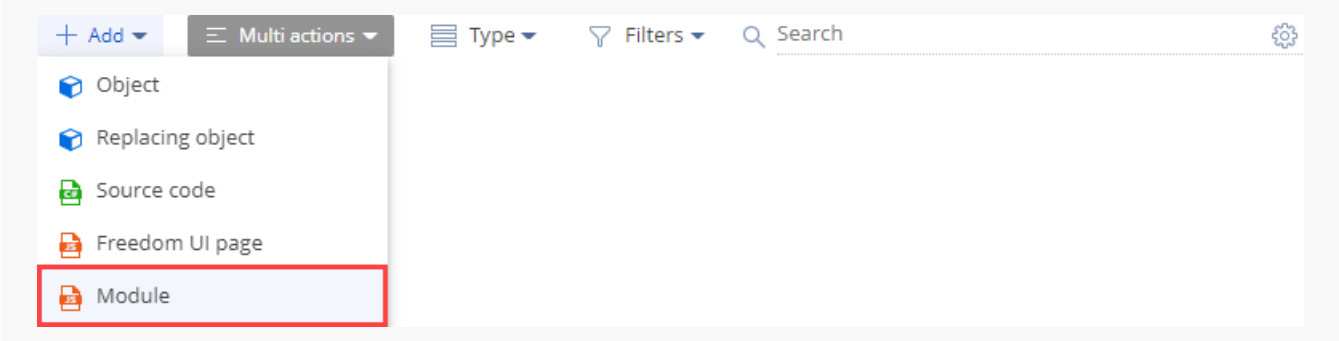
Note. Columns of the [*Details*] tab are read-only.

Set up the page that adds cases

You can add a column to the page that adds cases.

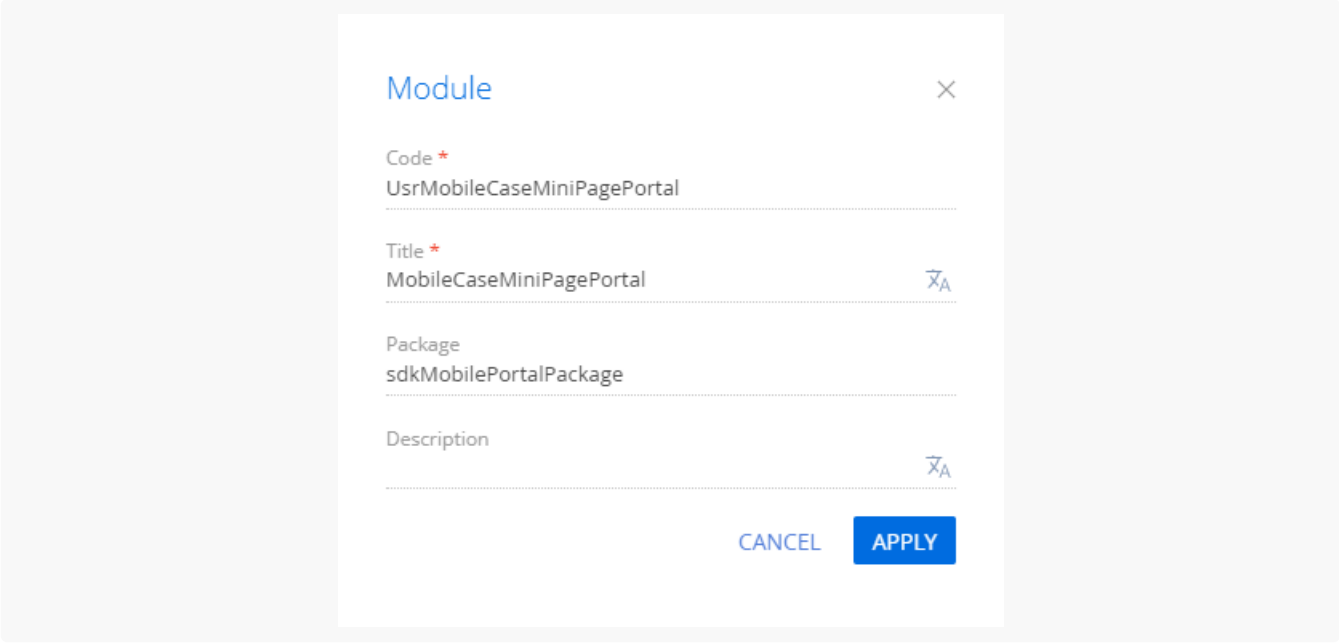
To **add a column to the page that adds cases**:

1. Create a **schema of the case page module**.
   a. Go to the [*Configuration*] section.
   b. Open the `MobileCaseMiniPagePortal` schema of the `CaseMobile` package and copy its contents.
   c. Select a custom package to add the schema.
   d. Click [*Add*] → [*Module*] on the section list toolbar.



   e. Fill out the **schema properties**.
      - Enter the schema name in the [*Code*] property. Required. The name must start with the prefix specified in the [*Prefix for object name*] (`SchemaNamePrefix` code) system setting, `Usr` by default. Can contain Latin characters and digits. When you create a configuration element schema, Creatio adds the prefix specified in the [*Prefix for object name*] (`SchemaNamePrefix` code) system setting to the current field automatically. Creatio checks whether the prefix exists and matches the system setting when you save the schema properties. If the prefix does not exist or does not match, Creatio sends a corresponding user notification.

- Enter the localizable schema title in the [*Title*] property. Required. The title of the configuration element schema is generated automatically and matches the value of the [*Code*] property without a prefix.



h. Add the copied contents of the `MobileCaseMiniPagePortal` schema of the `CaseMobile` package to the custom module.

i. Move the localized strings of the `MobileCaseMiniPagePortal` schema of the `CaseMobile` package to the custom module.

j. Add a **column**.

For Creatio version **8.0.2 and later**

For Creatio version **7.18.4-8.0.1**

k. Click [*Save*] on the Module Designer's toolbar.

2. **Register** the earlier created `UsrMobileCaseMiniPagePortal` custom schema in the portal workplace manifest.

a. Open the `MobileApplicationManifestPortal` schema in the custom package. If you are yet to set up the app using the Mobile Application Wizard, the `MobileApplicationManifestPortal` schema will not be available in the custom package. To **add** the `MobileApplicationManifestPortal` schema **to the custom package**, follow the instructions in a different section: Add base package schema to the custom package.

b. Register the **schema**.

a. Specify the schema used to add the schema record of the `Case` object in the `Modules` property.

b. Specify the schema used to extend the schema of the `Case` object in the `Models` property.

The example below registers the `UsrMobileCaseMiniPagePortal` schema.

Example of the `Modules` and `Models` property setup

```
{
    ...
    "Modules": {
        "Case": {
            ...
            "screens": {
                ...
                "add": {
                    "schemaName": "UsrMobileCaseMiniPagePortal"
                }
                ...
            }
            ...
        }
        ...
    },
    "Models": {
        "Case": {
            ...
            "PagesExtensions": [
                "UsrMobileCaseMiniPagePortal"
            ]
        }
        ...
    }
    ...
}
```

3. Click [*Save*] on the Module Designer's toolbar.

Brand and publish mobile apps built on Mobile Creatio

Easy

You can use your logos and names to brand a mobile app built on Mobile Creatio using the **SDKConsole utility**.

Set up the SDKConsole utility

In general, the procedure comprises the following **steps**:

1. Perform the preliminary setup.
2. Install and set up the SDKConsole utility.
3. Run the SDKConsole utility.

1. Perform the preliminary setup

1. **Ensure you can publish the app**. You must be enrolled into Apple Developer program to publish your app on iOS and have a Google Play developer account to publish the app on Android. Learn more on Apple and Android websites: Apple Developer Program, Google Play Console.
2. **Enable Firebase Cloud Messaging to send push notifications**.

   a. Sign in to https://firebase.google.com/.

   b. Click [*Go to console*] in the top right.

   c. Create a project in the console.

   d. Add your Android and/or iOS app to the project.

   e. Download the config files for the app and save them for later. Specify the path to these files in the utility settings using the `google_service_info_file` property.

   f. Retrieve the server API key. To do this, click ⚙ in the top left of the Firebase project dashboard → [*Project Settings*] → [*Cloud Messaging*] tab → [*Project credentials*] → [*Server key*].

   g. Save the server API key to the `[PushNotificationService]` table of the Creatio database. For example, you can do it using the following SQL query.

   SQL query

   ```
   UPDATE PushNotificationService SET Settings = '{"url":"https://fcm.googleapis.com/fcm/","ap
   ```

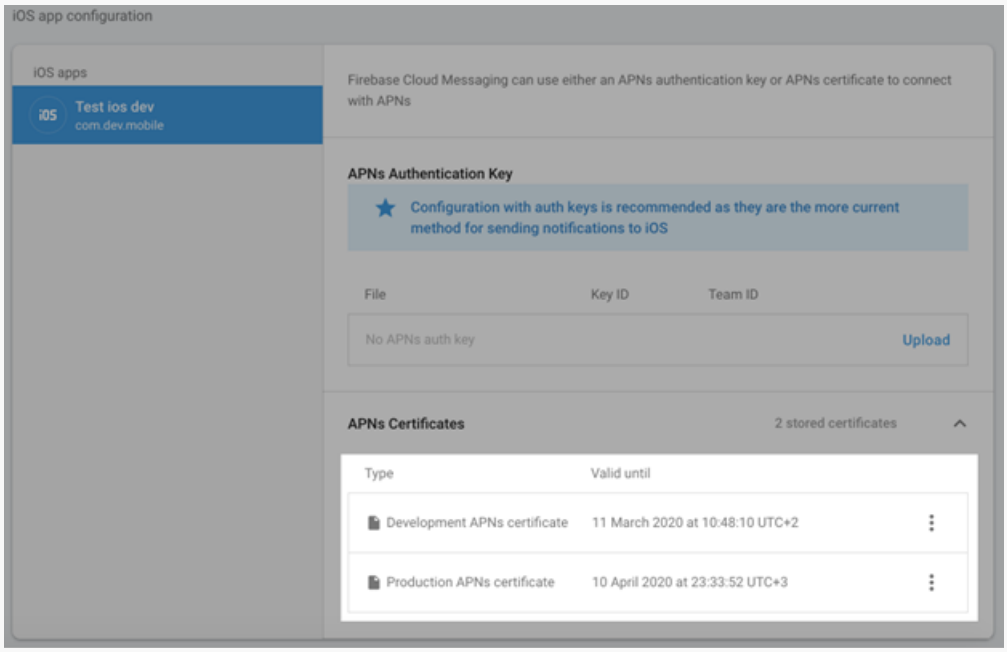
   `Some_Api_Key` is your API key.
3. Download **and install Java development kit version 8**.
4. **Register your app with App Store Connect (iOS only)**. To do this, follow the procedure in the official Apple documentation.
5. **Install the APNs certificates into Firebase (iOS only)**. iOS projects require you to install the development and production APNs certificates. To do this:

   a. Open the certificate configuration page: Certificate.

   b. Add and install development and production `Apple Push Notification service SSL` certificates into your Mac.

   You also need to generate, download and install provisioning profiles.

   Upload the *.p12 files exported from [*Keychain Access*] in the Firebase settings on the [*Cloud Messaging*] tab.



6. **Install J2ObjC (iOS only)**. The native functionality of the Mobile Creatio application is partly written in Java. The J2ObjC utility for Java code to Objective-C translation is required for shared use on iOS. Learn more about the utility in the official Google documentation. To **install J2ObjC**:

   a. Download the J2ObjC release version archive (2.0.5) to a Mac.

   b. Unpack the archive into your user home directory (`MacintoshHD/Users/MyUser /`). The unpacked archive must contain the `dist` directory.

   c. Rename the unpacked archive directory to `j2objc` .

2. Install and set up the SDKConsole utility

1. **Contact Creatio support** (support@creatio.com) and specify the email address that is or will be associated with your GitLab account. The support team will send you a signup link. After the signup, you will be able to access the SDKConsole project.
2. **Sign up for GitLab**. Open the link you received from the Creatio support and follow the instructions. If you sign in with a third-party service, make sure that you have a password set up for your GitLab account. To do this, click the profile icon in the top right → [*Edit profile*] → [*Password*].

3. **Install the SDKConsole utility**.

   Install the SDKConsole utility **on Mac**

   Install the SDKConsole utility **on Windows**

4. **Update the SDKConsole utility**.

   a. Back up the `SDK.config` file that contains the user settings. That way you will not have to reconfigure the utility.

   b. Download the utility archive from the Git repository.

c. Unpack the archive into your utility folder.

d. Move the `SDK.config` backup to the utility folder.

3. Run the SDKConsole utility

1. **Configure the SDKConsole utility**. Before you run the utility, configure the settings in the `SDK.config` file. Make sure that you do not use a single backslash (`\`) in your file paths. Learn more about the utility settings in a separate article: SDKConsole utility settings.

Example of the `SDK.config` file

```
{
    "name": "Creatio beta",
    "web_resources_path": "res/web",
    "tasks": ["prepare", "build", "deploy"],
    "use_extended_logging": true,
    "server_url": "https://mysite.creatio.com/",
    "iOS": {
        "repository_path": "https://gitlab.com/bpmonlinemobileteam/ios.git",
        "source_path": "",
        "google_service_info_file": "",
        "launch_storyboard_image_path": "res//LaunchStoryboard.png",
        "app_identifier": "com.myapp.mobile",
        "app_icon_path": "../res/AppIcon.png",
        "version_number": "7.13.9",
        "build_number": "2"
        "app_store_login": "some@gmail.com",
        "certificate_path": "/Users/your_user_dir/ios_distribution.cer",
        "certificate_password": "private_key_password_of_certificate",
        "apple_2FA_specific_password": "apple_specific_password",
        "testflight_changelog": "My what's new"
    },
    "Android": {
        "build_type": "debug",
        "repository_path": "https://gitlab.com/bpmonlinemobileteam/android.git",
        "source_path": "",
        "google_service_info_file": "",
        "package_name": "com.myapp.mobile",
        "version_number": "1.1.1",
        "build_number": 2,
        "native_resources_path": "res/android/res",
        "key_file": "C:/hybrid/platforms/android/androidappkey",
        "store_password": "android_app_distribution_password",
        "key_alias": "some_key_alias",
        "key_password": "key_password"
    }
}
```

2. **Run the SDKConsole utility**.

   Run the SDKConsole utility **on Mac**

   Run the SDKConsole utility **on Windows**

Description of the solutions for typical errors

View the solutions for typical errors in the table below.

The solutions for typical errors

Error description	
The `Unable to determine Android SDK directory` error on Mac	1. Open Terminal. 2. Run the following commands at the terminal: `echo "export ANDROID_HOME=~/Library/Android/sdk;export PATH=${PATH}:$AN`
The `Couldn't find the specified scheme 'bpm'online'. Please make sure that the scheme is shared…` error when running the build for an iOS project on Mac	Re-run the build. If this does not help, take the following steps: 1. Open the iOS project (`BPMonlineMobile.xcworkspace`) in XCode. 2. Select the current build scheme. If the `[bpm'online]` scheme is not selected, select it 3. Click the current scheme once more and select [*Edit scheme…*] in the list. 4. Select the [*Shared*] checkbox. 5. Close XCode. 6. Run the `./build` command at the Terminal.
Remove the `permission denied` message for my build in the Mac Terminal	Run the following command at the terminal: `chmod -R +x build`

The `function fs.copyFileSync is undefined` error during the build on Mac	Run the following commands at the Terminal:
	`brew link --overwrite node`
	`brew postinstall node`

The `Unable to determine Android SDK directory` error on Windows	Specify the `ANDROID_HOME` environment variable where you need to provide the path to th

SDKConsole utility parameters CLI

Beginner

`name`

Name of your application.

`web_resources_path`

Path to the directory that contains the resources used in the app, i. e., the logo and background on the login page.

`tasks`

Actions the utility executes. This is a string array where you can specify a combination of the tasks.

Available values

`prepare`	Preparation/rebranding of your iOS/Android project. This step makes all the necessary changes. You will get a finished project you can publish in AppStore and Google Play.
`build`	Build the project. You will get an assembled * .ipa iOS app file and\or * .apk Android app file.
`deploy`	Publish the app to TestFlight. iOS only.

`use_extended_logging`

Show detailed logs in the terminal when the utility is running. The recommended value is `true`. If set to `false`, the terminal displays only the currently executed step without details.

`server_url`

Default server. The server URL will be automatically specified on the login page when you log in to the app for the first time.

`repository_path`

Path to the GitLab repository that hosts the original Android/iOS project.

`source_path`

Path to the local Windows/Mac directory where the original Android/iOS project is located. If you specify this parameter the utility uses it in place of the `repository_path` parameter.

`google_service_info_file`

Path to the `GoogleService-Info.plist` (iOS) or `google-services.json` (Android) file required to connect to the Firebase push notification service.

`version_number`

App version in the following format: `0.0.1`.

`build_number`

Build number (string). Always update the build number before you perform the `deploy` task.

`launch_storyboard_image_path`

Path to the image displayed when the application starts (2732x2732 px). iOS only.

`app_identifier`

A unique app ID, for example, `com.myapp.mobile`. This is the Bundle ID specified when you registered the app in App Store Connect. iOS only.

`app_icon_path`

Path to the app icon (1024x1024 px). This is a master image the utility uses to generate the required icons for current iOS devices. iOS only.

`app_store_login`

Account (Apple ID) required to connect to App Store Connect / TestFlight. iOS only.

`certificate_path`

Path to the distribution certificate required when publishing to TestFlight. iOS only.

`certificate_password`

Certificate password. To restore the password, contact the certificate author. iOS only.

`apple_2FA_specific_password`

Specific password. iOS only.

Currently, all Apple accounts support two-factor authentication. To enable third-party services to connect to Apple services, the `app-specific passwords` were added. To get a specific password:

1. Open the https://appleid.apple.com/#!&page=signin URL while signed in to your Apple account.
2. Open the [*Security*] section → [*Generate password...*] command.
3. Follow the instructions to get a new password generated.

`testflight_changelog`

Description of the published changes to TestFlight (what's new). The description is published for the primary app language set in App Store. iOS only.

`build_type`

Build type. Android only.

Available values

`debug`
`release`
`release-unsigned`

`package_name`

A unique app ID, for example, `com.myapp.mobile`. Android only.

`native_resources_path`

Path to app resources, such as the app icon and the startup image. Structure the contents of this directory similarly to the `res` folder in the Android project. The directory can contain subdirectories that have drawable, drawable-xhdpi, and other icons. Android only.

`key_file`

Path to the key file (keystore) required to sign the app. Learn more about signing apps in the official Android documentation. Android only.

`store_password`

Password for the keystore required to sign the app. Android only.

`key_alias`

The key alias. Android only.

`key_password`

The password of the alias from the `key_alias` parameter in the keystore. Android only.