



Development Guide

ACCELERATE 

Table of Contents

Creatio development guide	14
Getting started	15
How to start the development	15
Introduction	15-16
Development process	16
Introduction	16-18
Recommended development sequence	18-20
Development rules	20-21
Development environment	21
Introduction	21-22
Deploying the Creatio on-site application	22-23
Deploying the Creatio cloud application	23-24
Version control system settings	24
Create repository in SVN server	24-27
Platform description	28
Creatio architecture	28
Main application	28-37
Components and services	37
Mobile application	37-40
Portal	40-45
Process modeling (Studio Creatio, free edition)	45-46
Global Search Service	46-48
Bulk duplicate search service	48-50
Website event tracking service	50-51
Database enrichment service	51-53
Machine Learning service	53-54
Phone integration service	54-56
Sync Engine synchronization service	56-58
Exchange Listener synchronization service	58-59
Static content bundling service	59-61

Bulk email service	61-62
Developing solutions in Creatio	62
Introduction	62-65
Low-code and no-code development	65-66
Front-end (JS)	66-71
Back-end (C#)	71-73
Integrations	73-75
Developer tools	75
Introduction	76
For solving simple customization tasks	76
Introduction	76-77
Development tools. Built-in IDE	77
Introduction	77-83
The [Configuration] section. The [Data] tab	83-86
Source code and metadata viewport	86-88
Designers of configuration items	88-89
Workspace of the Object Designer	89-90
Module designer	90-93
Source code designer	93-95
Process designer for embedded business processes	95-98
User task designer workspace	98-99
Workspace of image list designer	99-100
Creating the entity schema	100-108
Creating a custom client module schema	108-112
Creating the [Source code] schema	112-114
Development tools. Packages	114
Introduction	114
Package structure and contents	114-116
Package dependencies	116-121
Package [Custom]	121-123
Creating a package for development	123-126
Binding data to packages	126-131

Version control system. Built-in IDE	131-132
Committing a package to repository	132-133
Installing packages from repository	133-137
Updating package from repository	137-138
Delivery tools. Built-in IDE	138
Packages export and import	138-145
Schema export and import	145-146
Transferring changes using SVN	146-148
Debugging tools	148
Client code debugging	148-152
IsDebug mode	152-155
For solving complex customization tasks and server-side development	155-156
Introduction	156
Development tools. IDE Microsoft Visual Studio	156
Introduction	156-160
IDE settings for development	160-164
Working with the server side source code	164
Overview	165-169
Developing the configuration server code in the user project	169-177
Working with the client code	177
Introduction	177-180
Automatic displaying of changes	180-184
File content	184
Packages file content	184-187
Localization of the file content	187-189
Client static content in the file system	189-192
How to use TypeScript when developing custom functions	192-197
Creating an Angular component to use in Creatio	197-202
Development tools. SQL	202
How to work with PostgreSQL	202-204
Version control systems	204
Subversion	204-205

Introduction	205-207
Creating a package in the file system development mode	207-215
Installing an SVN package in the file system development mode	215-221
Binding an existing package to SVN	221-231
Updating and committing changes to the SVN from the file system	231-234
Creation of the package and switching to the file system development mode	234-240
Git	240
Working with Git	241-245
Testing tools. NUnit	245-250
Logging tools	250-251
Logging in Creatio. NLog	251-253
Logging in Creatio. Log4net	253-255
Server code debugging	255-263
Delivery tools	263
WorkspaceConsole utility	263
Settings	263-265
Parameters	265-270
Exporting packages from database	270-272
Saving packages to the database	272-275
Saving SVN packages	275-277
Transfer changes	277-280
Server-side development	280
Overview	280
Replacement class object factory	280-286
Repositories	286-295
Repositories	295-304
Configuration web-services	304
Creating a configuration service	304-306
Creating anonymous web service	306-308
Calling configuration services with ServiceHelper	308-309
Working with database	309

Working with database	309-311
Root schema. Building paths to columns	311-314
Multi-row data insert. The Insert class	314-315
Multithreading when working with the database. Using the DBExecutor	315-316
Localizable configuration resources	316
Introduction	316-319
Localizable resource structure and use	319-322
Localization tables	322-323
Bound data structure	323-324
Entity event layer	324-327
Developing the source code in the file content (project package)	327-329
Class Description	329-330
The Select class	330-336
The EntitySchemaQuery class	336-346
The Insert class	346-348
The InsertSelect class	348-350
The Update class	350-353
The Delete class	353-355
The Entity class	355-364
The EntityMapper class	364-367
The QueryFunction class	367-385
The EntitySchemaQueryFunction class	385-400
Examples	400
Creating replacement classes in packages	400-404
Configuration web-services	404-405
Creating a configuration service	405-407
Creating anonymous web service	407-410
Calling configuration services with ServiceHelper	410-414
Calling configuration services using Postman	414-418
CRUD operations	418
Retrieving data from the database	418-423

Retrieving data based on user permissions	423-426
Adding data	426-428
Adding data using subqueries	428-429
Modifying data	429-430
Deleting data	430-431
Working with database entities	431-434
Work with localizable resources	434
Enabling multi-language in an object schema	434-436
Reading multilingual data with EntitySchemaQuery	436-438
Working with the localized data via Entity	438-442
Localizing views	442-447
Client-side development	447
Application logical levels	447
Introduction	447-450
Configuration architectural elements	450-454
Modules	454
AMD concept. Module definition	454-457
Modular development principles in Creatio	457-463
Module types and their specificities	463-467
Module message exchange. Sandbox component	467
Module message exchange	467-473
Bidirectional messages	473-477
Loading and unloading modules	477-481
Client view model schemas	481
Introduction	481-482
Mixins. The "mixins" property	482-488
Attributes. The "attributes" property	488-489
Messages. The "messages" property	489-491
Properties. The "properties" property	491-492
Methods. The "methods" property	492-493
Rules. The "rules" property	493
Business rules. The businessRules property	493-494

Modules. The "modules" property	494-495
The "diff" array	495-497
The "diff" array. Alias mechanism	497-500
Schema formatting requirements	500-504
CRUD-operation implementation on client	504-505
The EntitySchemaQuery class. Building of paths to columns	505-506
The EntitySchemaQuery class. Adding columns to a query	506-510
The EntitySchemaQuery class. Getting query result	510-511
The EntitySchemaQuery class. Filters handling	511-515
WebSocket messages transferring mechanism. ClientMessageBridge	515
ClientMessageBridge. Message history save mechanism	515-516
ClientMessageBridge. API description	516-518
ClientMessageBridge. The client-side WebSocket message handler	518-523
Frequently used client-side classes	523
The DataManager class	523-527
The SourceCodeEditMixin class	527-531
Interface controls	531
Basic interface controls	531
Controls. Introduction	531-532
Main menu	532-533
Sections	533
Introduction	534-535
Section lists	535-538
Section analytics	538-539
Section actions	539-540
Filters	540-542
Tags	542-543
Record edit page	543-544
Details	544
Introduction	544-547
Details	547-558
Mini-page	558-559

Modal windows	559-560
Communication panel	560-561
Command line	561-562
Action dashboard	562
Dashboard widgets	562-563
Introduction	563-566
Charts	566-567
Metrics	567-568
Gauge	568-569
Lists	569-570
Web-page	570-571
Sales pipeline	571-572
The [Timeline] tab	572
Introduction	572-575
Creating the [Timeline] tab tiles bound to custom section	575-585
The [Connected entity profile] control	585-594
Interface control tools	594
Locking edit page fields	594-597
Feature Toggle. Mechanism of enabling and disabling functions	597-600
User services and tools	600
Report setup	600-601
Introduction	601-603
Setting up reports in Creatio	603-607
Setting up the report	607-616
Setting up the report with an image	616-626
Phone integration	626
Introduction	626-630
Oktell	630-633
Webitel	633-634
Asterisk	634-636
Creatio marketing	636
Campaign elements	636-638

Sync Engine synchronization mechanism	638
Creatio synchronization with external storages	638-645
Synchronizing metadata in Creatio	645-648
Synchronizing tasks with MS Exchange	648-650
Synchronizing email with MS Exchange	650-652
Synchronizing contacts with MS Exchange	652-655
Synchronizing appointments with MS Exchange	655-657
Scheduler setup	657
Recommendations on scheduler setup	657-660
Quartz policies for the processing of overdue tasks	660-662
Self-service Portal	662
Introduction	662-663
PortalMessagePublisherExtensions mixin. Portal messages in SectionActionsDashboard	663-665
Restricting access to web services for portal users	665-668
Machine learning service	668
Introduction	668-673
Creating data queries for the machine learning model	673-676
Connecting a custom web-service to the machine learning functionality	676-685
Sending emails	685
Sending emails from existing account	685-698
Sending emails using the explicit account credentials	698-718
Contact data enrichment from emails	718-722
Static content bundling service	722-726
Class libraries and REST API	727
Integrations and external API	727
Introduction	727-728
Choosing the method of integration with Creatio	728-735
Authentication of external requests	735-739
Integration via OData protocol	739
Creatio integration via the OData 3 protocol	739-752
Creatio integration via the OData 4 protocol	752-763

DataService	763-764
The ProcessEngineService.svc web service	764-766
Integration examples with Creatio	766
Integration tools	766-767
Executing OData queries using Fiddler	767-776
Working with requests in Postman	776-782
Working with request collections in Postman	782-788
OData	788
OData integration examples	788
Working with Creatio objects over the OData protocol WCF-client	788-793
Working with Stream data	793-796
Working with batch-requests	796-806
DataService	806
DataService. Adding records	806-808
DataService. Reading records	808-813
DataService. Data filtering	813-816
DataService. Using macros	816-818
DataService. Updating records	818-820
DataService. Deleting records	820-821
DataService. Batch queries	821-822
Creatio development cases	823
Section business logic	823
Creating a new section	823-826
Adding an action to the list	826
Introduction	826-828
How to add a section action: handling the selection of a single record	828-831
How to add a section action: handling the selection of several records	831-834
Handling the selection of several records. Examples	834-840
How to add a button to a section	840-844
How to highlight a record in the list in color	844-846
Adding quick filter block to a section	846-849
Deleting a section	849

Page configuration	849-850
Introduction	850-851
Setting the edit page fields using business rules	851
Introduction	851-854
The FILTRATION rule use case	854-858
The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition	859-863
The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition	863-865
The BINDPARAMETER rule. How to make a field required based on a specific condition	865-869
Business rules created via wizards	869-871
Adding an action to the edit page	871-876
Control elements	876
Adding a new field to the edit page	876-881
Adding a button to the edit page	881-882
Introduction	882-883
How to add a button to an edit page in the new record add mode	883-887
How to add the button on the edit page in the combined mode	887-892
How to add a field with an image to the edit page	892-900
How to add the color select button to the edit page	900-903
How to add multi-currency field	903-910
How to add custom logic to the existing controls	910-914
Adding calculated fields	914-918
How to set a default value for a field	919-923
How to add the field validation	923-929
Using filtration for lookup fields. Examples	929-933
Adding an action panel	933-936
Adding a new channel to the action panel	936-942
Displaying contact's time zone	942-946
How to display the difference between dates on edit page fields	946-947
How to block fields of the edit page	947-949
Work with details	949-950

Introduction	950
Creating a detail in wizards	950-954
Adding an edit page detail	954-963
Adding a detail with an editable list	963-971
Creating a detail with selection from lookup	971-979
Adding multiple records to a detail	979-982
Creating a custom detail with fields	982-987
Advanced settings of a custom detail with fields	987-992
Adding the [Attachments] detail	992-1001
Displaying additional columns on the [Attachments] tab	1001- 1003
How to hide menu commands of the detail with list	1003- 1005
Deleting a detail	1005
Business processes	1005
How to add auto-numbering to the edit page field	1005- 1014
Process launch from a client module	1014- 1019
Creating custom [User task] process element	1019- 1028
How to customize notifications for the [User task] process element	1028- 1034
How to run Creatio processes via web service	1034- 1042
How to save the record without closing the edit page which is opened by the business process	1042- 1048
Typical customizations	1048
Creating pop-up summaries (mini pages)	1048- 1056
Adding pop-up summaries (mini pages) to a module	1056- 1060
Creating a pop-up summary (mini page) for adding records	1060- 1065
Adding pop-up hints	1065- 1073

How to modify sales pipeline calculations	1073- 1077
How to enable additional filtering in a sales pipeline	1077- 1079
Adding a custom dashboard widget	1079- 1085
Using the Terrasoft.AlignableContainer custom control	1085- 1093
Adding a duplicate search rule	1093- 1096
Adding a rule for duplicates search when saving a record	1096- 1101
Junk case custom filtering	1102- 1103
How to display custom implementation of approving in the section wizard	1103
How to create custom reminders and notifications	1103-1112
Adding multi-language email templates to a custom section	1112-1118
Working with email threads	1118-1120
Integration of third-party sites via iframe	1120- 1122
Analytics	1122
Basic macros in the MS Word printables	1122-1131
How to create macros for a custom report in Word	1131-1146
Sales Creatio customization	1146
How to change the calculation for the "Closed" column in the [Forecasts] section.	1146-1151
Configuration of the editable columns on the product selection page	1151-1154
Service Creatio customization	1154
Adding a new rule for calculating case deadline	1154-1159
Adding the macro handler to the email template	1159-1162
How to hide feed area in the agent desktop	1162-1163
Adding floating icons for internal case feed posts	1163-1166
Financial Services Creatio customization	1166-1167
How to create custom verification action page	1167-1170
Using the EntityMapper schema	1170-1175

Marketing Creatio customization	1175-1176
Adding a custom campaign element	1176-1187
Configuring campaign elements for working with triggers	1187- 1188
Adding a custom transition (flow) to a new campaign element	1188- 1202
Web-to-Case	1202
Introduction	1202- 1204
Creating Web-to-Case landing pages	1204- 1209
Web-To-Object. Integration via landings and web-forms	1209- 1211
Setting up web forms for a custom object	1211-1222
Prediction	1222- 1223
How to implement custom prediction model	1223- 1226

Creatio development guide

- **Getting started**
- **Platform description**
- **Class libraries and REST API**
- **Creatio development cases**

Getting started

Contents

- **How to start the development**

How to start the development

Contents

- **Introduction**
- **Development process**
- **Development environment**
- **Version control system settings**

How to start the development

Beginner

Easy

Medium

Advanced

Keep the following sequence of action.

1. Set up the development process.

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

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

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

To develop complex project solutions, use the recommendations provided in the "[Project Life Cycle Methodology](#)" documentation.

2. Select and configure development environment

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

To use specific development tools (for example, Visual Studio), you will need to deploy application on-site. Please refer to "**Deploying the Creatio on-site application**" for any details.

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

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

3. Configure SVN storage (optional)

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

More information about the version control system can be found in the "[Create repository in SVN server](#)" and "[Working with SVN in the file system](#)" articles.

4. Create a custom package for developing new functionality

The Creatio functionality is implemented in configuration elements – schemas. A set of schemas that implement some functionality is combined in a package. More information about purpose and structure of Creatio packages can be found in the "[Package structure and contents](#)", "[Package dependencies](#)" and "[Package \[Custom\]](#)" articles.

Create a new custom package to develop new functionality. Please refer to the "[Creating a package for development](#)" article for any details.

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

For more information about creating a package in the development in file system mode, refer to the "[Creating a package in the file system development mode](#)" article.

5. Create schemas that implement the functionality

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

Templates of development of new functionality are described in the "[Creatio development cases](#)" article.

6. Transfer modifications to test and production environments

After the development is completed, the modifications must be transferred to the pre-production environment for the testing. If the testing was successful, transfer the modifications to the production environment. For more information refer to the "[Transferring changes between the working environments \(on-line documentation\)](#)" article.

See also

- [Development process organization](#)
- [Organizing a development environment](#)
- [Recommended development sequence](#)
- [Development rules](#)
- [Deploying the Creatio on-site application](#)
- [Deploying the Creatio cloud application](#)
- [Create repository in SVN server](#)
- [Working with packages](#)
- [Transferring changes between the working environments \(on-line documentation\)](#)
- [Creating a custom client module schema](#)
- [Creating the entity schema](#)
- [Creating the \[Source code\] schema](#)

Development process

Contents

- [Introduction](#)
- [Recommended development sequence](#)
- [Development rules](#)

Development process organization

Beginner

Easy

Medium

Advanced

Overview

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

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

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

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

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

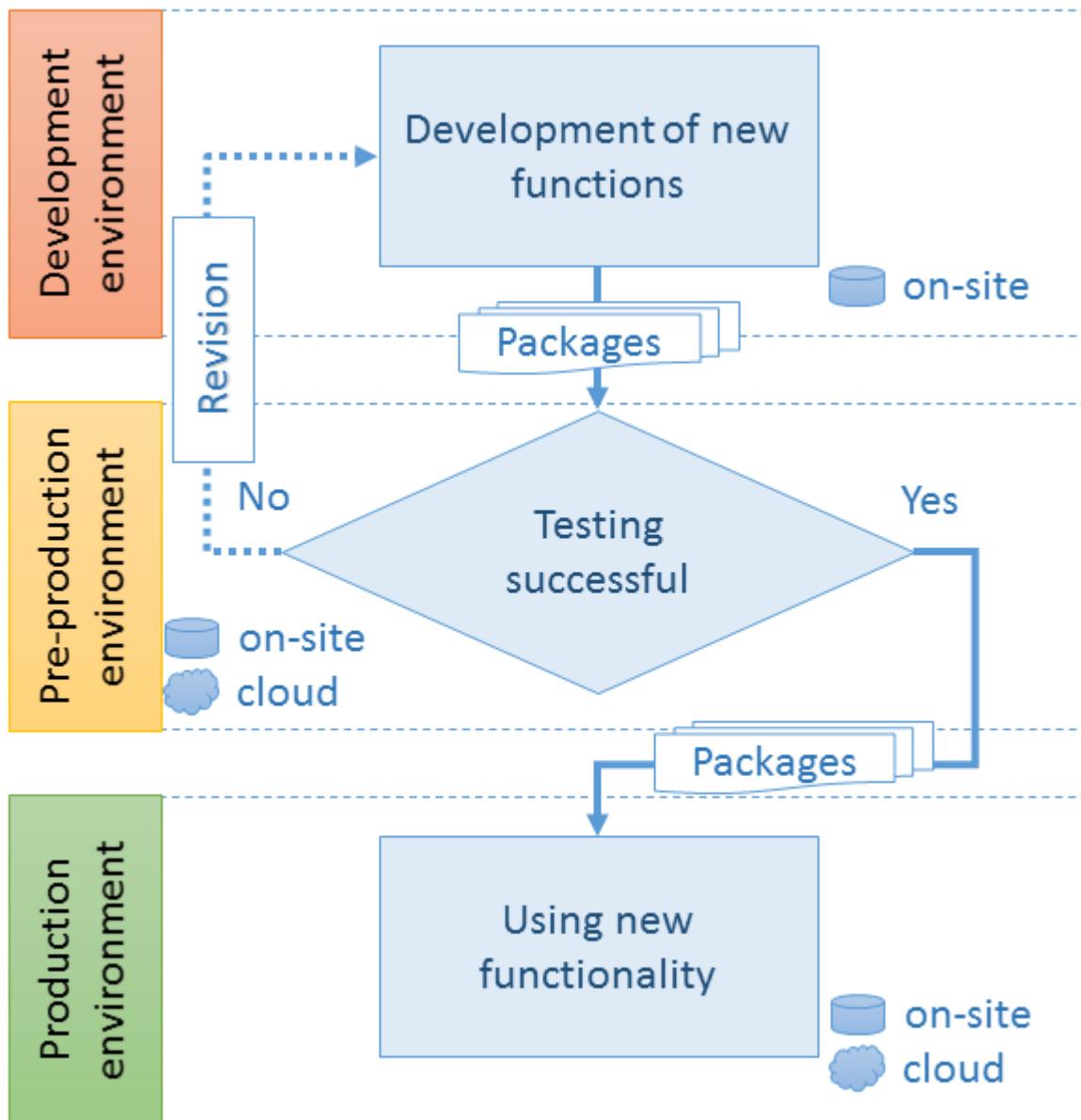
For more information about deployment options, please refer to the "**Deploying the Creatio on-site application**" and "**Deploying the Creatio cloud application**".

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

The general development process is shown in Fig. 1.

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

Fig. 1. General workflow of the development process



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

Recommended development sequence

Beginner Easy Medium Advanced

Introduction

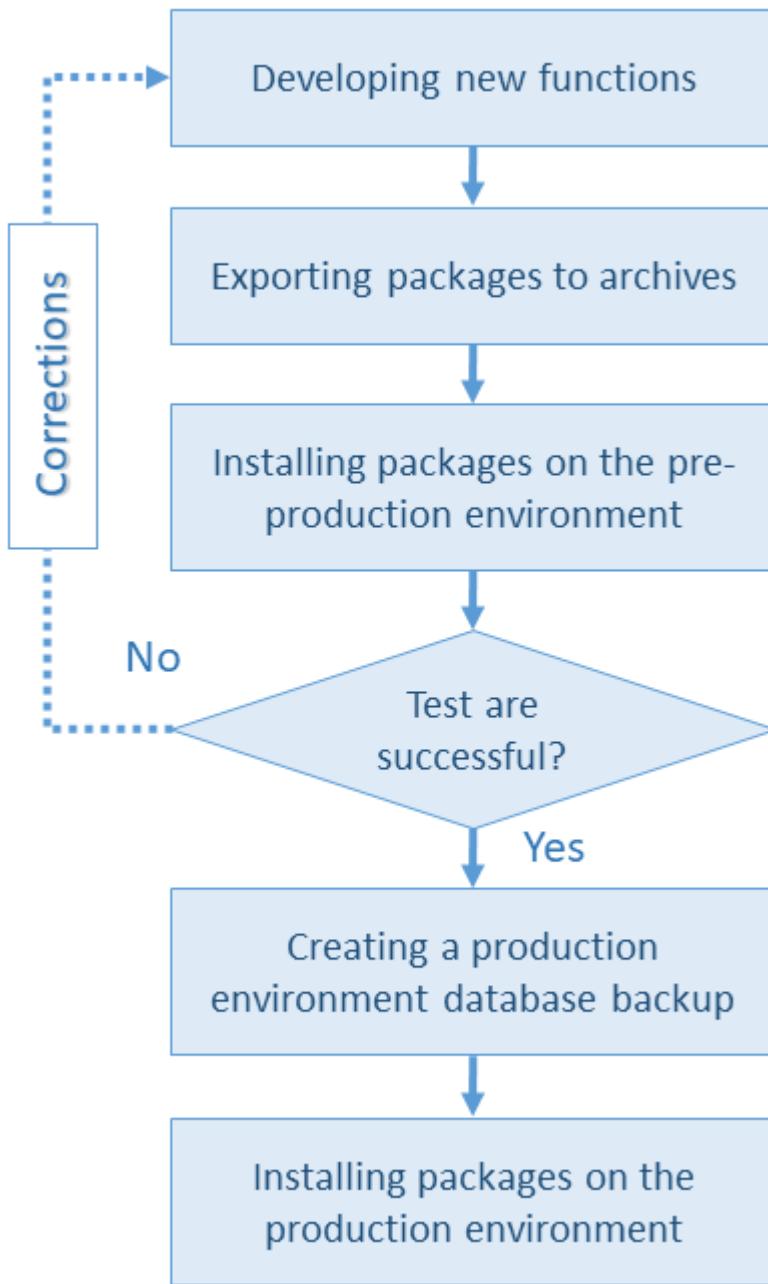
Development of complex functionality requires proper **organization of the development processes**. There are three general options for development environment deployment:

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

Development sequence

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

Fig. 1. General sequence of development



1. Development of new functions

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

Using SVN is not recommended for transferring changes to the production environment. Transferring changes with SVN can only be used in the development environment.

2. Exporting packages to archives

Two options for uploading packages into archives:

- From the [Configuration] section (see the “[Transferring changes using packages export and](#)

import” article).

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

3. Installing the packages on the pre-production environment

There are two options for installing packages to application:

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

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

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

4. Creating production database backup

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

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

5. Installing the packages on the production environment

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

Development rules

Beginner

Easy

Medium

Advanced

Introduction

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

Minimal required developer skills

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

Recommended developer skills

Over a year of C#, JavaScript and T-SQL (PL-SQL) programming experience. Expertise in [WCF](#) and [OData](#) technologies, as well as [Sencha Ext.JS](#) framework and [RequireJS](#) library.

Development rules and recommendations

Using a development environment

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

Developing in a configuration

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

Developing in a custom package

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

Using the SVN

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

Identification of the solution provider

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

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

Using the extending and replacing modules and schemas

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

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

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

Using localizable strings

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

Data backup

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

Development environment

Contents

- **Introduction**
- **Deploying the Creatio on-site application**
- **Deploying the Creatio cloud application**

Organizing a development environment

Beginner

Easy

Medium

Advanced

Overview

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

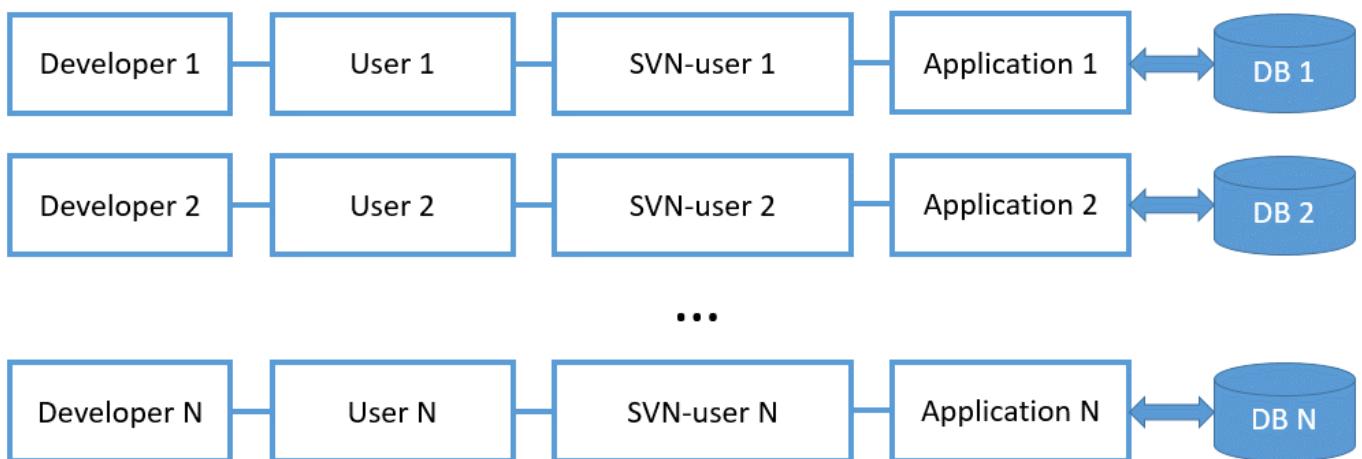
The production environment database must never be used for development. Development-related activities in the production environment are strictly forbidden.

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

Development in on-site application

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

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



Advantages:

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

Disadvantages:

1. Cloud deployment option is unavailable.

Recommendations:

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

Deploying the Creatio on-site application

Beginner

Easy

Medium

Advanced

Introduction

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

To deploy the Creatio application on-site, the server-side and the client-side must meet certain technical requirements. These requirements are described in the "[Server-side system requirements](#)" and "[Client-side system requirements](#)" articles. Complying with certain technical requirements will ensure high system performance.

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

Deploying the Creatio cloud application

Beginner

Easy

Medium

Advanced

Introduction

The standard procedure for deploying the Creatio cloud application is as follows:

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

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

Primary limitations

The use of SQL Agent is restricted

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

The use of DB Mail is restricted

Email notifications must be sent via Creatio platform features.

The use of Extended Stored Procedure is restricted

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

The use of DBMS user names is restricted

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

Modifications to Web.config are restricted

All required parameters must be stored as Creatio system settings.

Binding to server and DBMS IP addresses is restricted

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

Installing additional software is restricted

No additional software can be installed on Creatio servers.

Working with file system is restricted

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

Third-party applications cannot be run on server

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

Database must be deployed on SQL Server 2016

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

The application must support both HTTP and HTTPS protocols

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

The application must work with access rights of a regular user

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

The application must work as a user without a profile

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

Additional recommendations for partners

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

Version control system settings

Contents

- **Create repository in SVN server**

Create repository in SVN server

Beginner

Easy

Medium

Advanced

Introduction

The purpose of version control system in Creatio:

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

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

Creatio supports operation with Subversion control system (SVN) of version 1.7 and higher.

For more details on use of SVN see [documentation](#).

Principles of operation with repositories of version control system

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

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

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

SVN setup

To set up integration with SVN:

1. Install SVN server

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

Use one of the publicly available SVN installers for Windows:

- [VisualSVN](#)
- [CollabNet](#)

You can download the last version of binary files of the SVN server for your operating system [here](#).

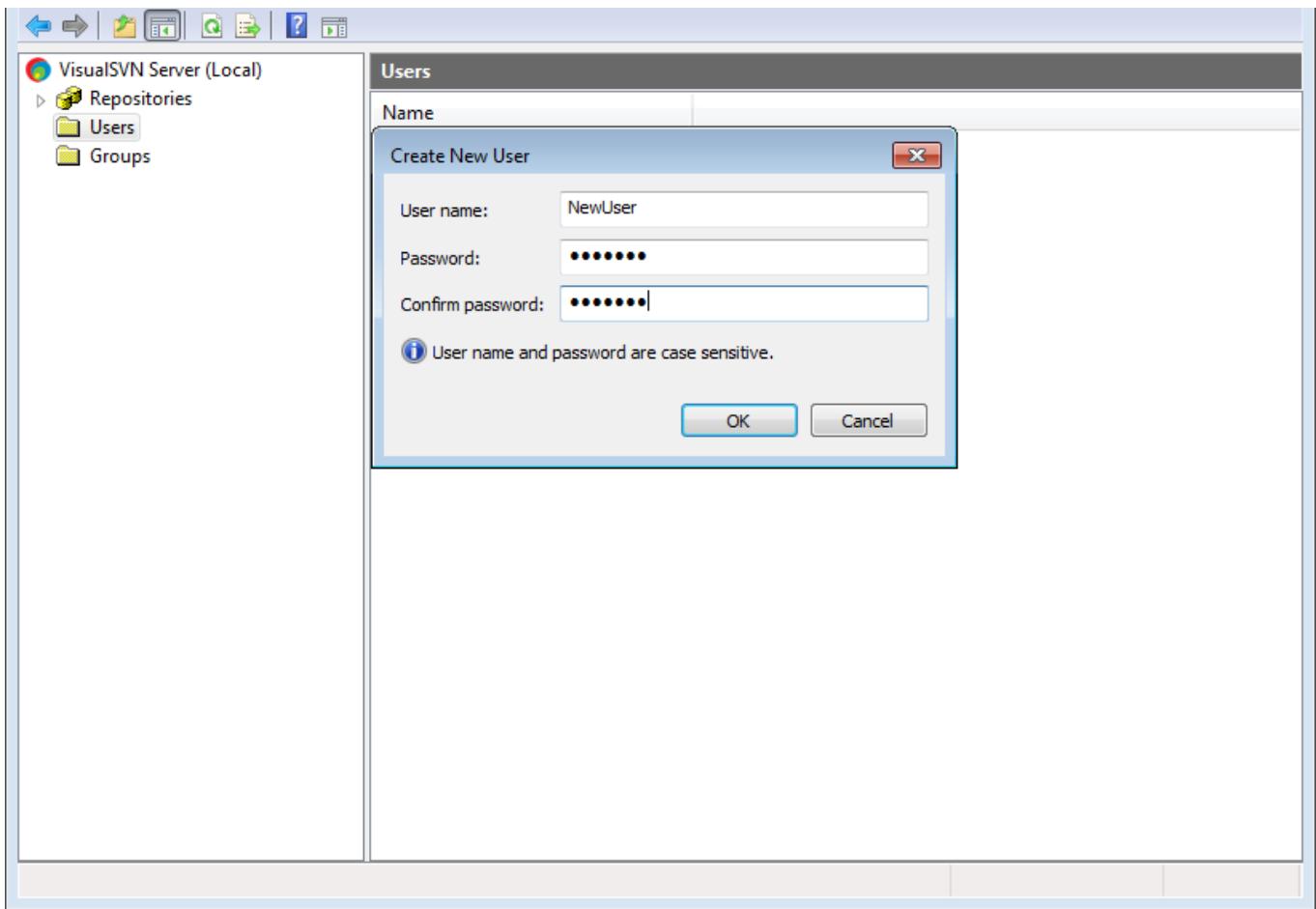
SVN server can function independently or through Apache web-server (it is installed automatically by means of the VisualSVN and CollabNet utilities). In the first case, repositories are accessed through svn:// protocol. In the second case repositories are accessed through the http(s):// protocol.

We recommend using the http(s):// protocol for integration with Creatio.

2. Create a user on the SVN server

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

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



3. Create repository on the SVN server

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

Creatio supports simultaneous operation of several repositories that can be located on different SVN servers.

4. Install SVN client

You can additionally install an SVN client in the developer workplace, for example, [TortoiseSVN](#).

We recommend using TortoiseSVN client version 1.8 and up.

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

List of repositories

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

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

Add	Edit	Delete	Authorize	
Name	Storage address	Active		
CommonStorage	http://svnserver:1050/CommonStorage	<input checked="" type="checkbox"/>		
CustomStorage	http://svnserver:1050/CustomStorage	<input type="checkbox"/>		
ProductTrunk	http://svnserver:1050/Product/trunk	<input checked="" type="checkbox"/>		

Adding a new repository

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

Fig. 3. — New repository card

Name	CustomStorage
Storage address	http://svnserver:1050/CustomStorage
Active	<input checked="" type="checkbox"/>
<input type="button" value="OK"/> <input type="button" value="Cancel"/>	

[Name] — repository name.

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

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

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

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

Platform description

Contents

- **Creatio architecture**
- **Developer tools**
- **Server-side development**
- **Client-side development**
- **Interface controls**
- **User services and tools**

Creatio architecture

Contents

- **Main application**
- **Components and services**
- **Developing solutions in Creatio**

Main application

Beginner

Easy

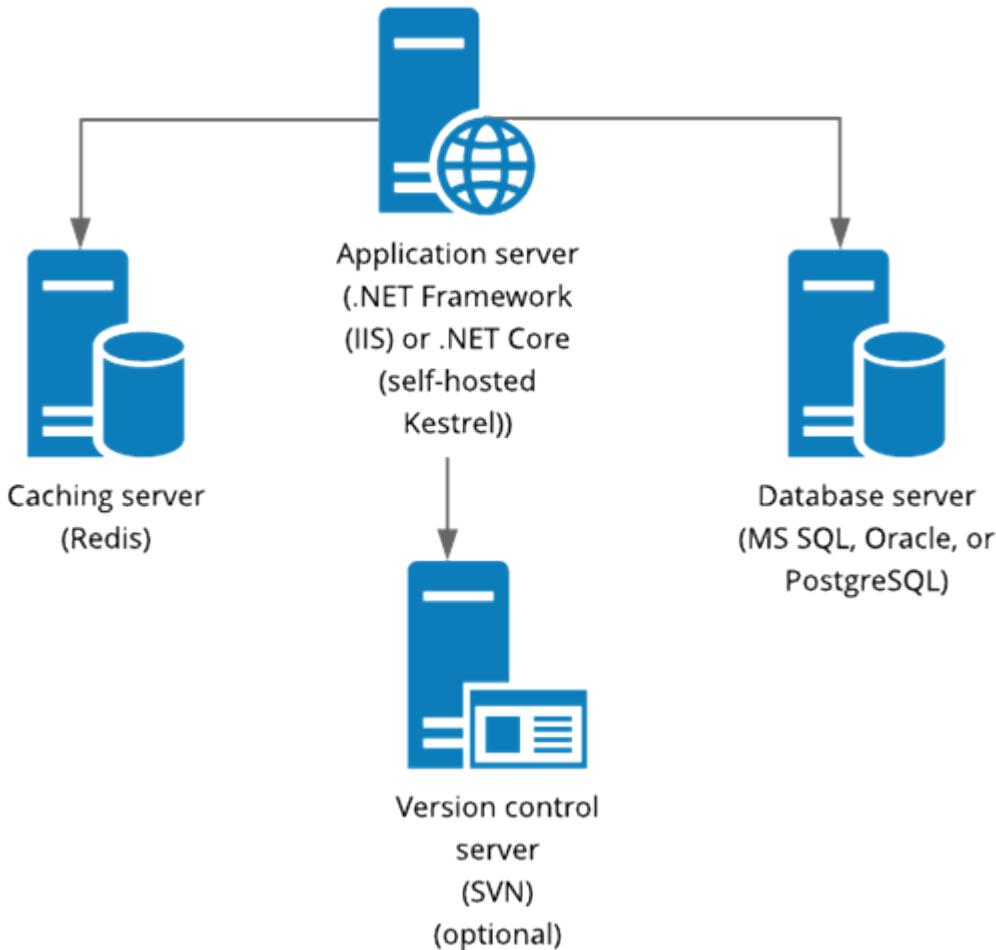
Medium

Advanced

Application infrastructure

The architecture of the main Creatio application is presented in Figure 1.

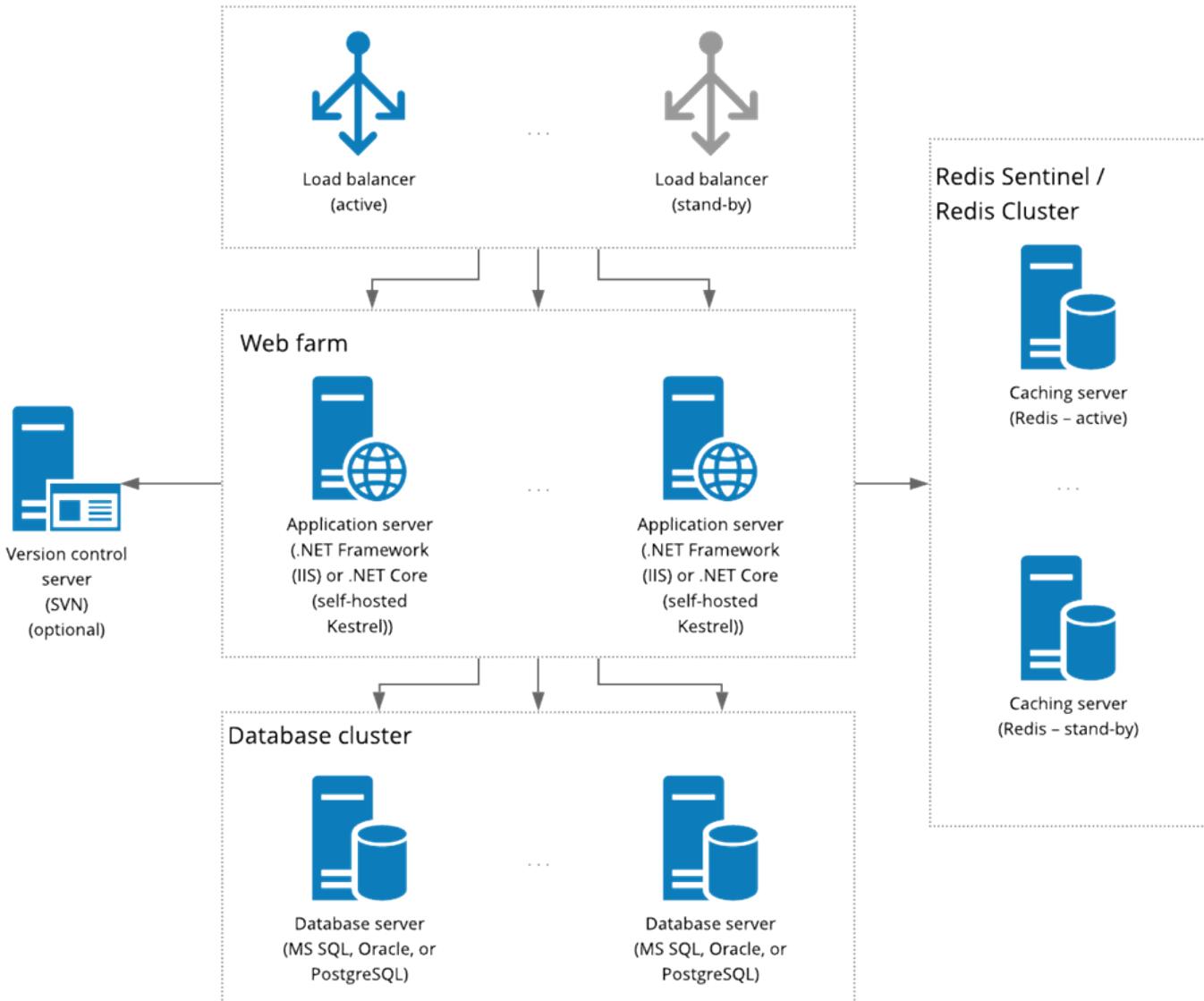
Fig. 1. The architecture of the main Creatio application



You can enhance the performance of large-scale Creatio projects through [horizontal scaling](#). When using horizontal scaling, the architecture of the main Creatio application (Fig. 2) includes the following components.

- [Load balancers](#).
The load balancer may be either hardware or software. To work in fault-tolerant mode, use the HTTP/HTTPS traffic balancer that supports the WebSocket protocol. Learn more about setting up the HAProxy load balancer in the "[Set up a web-farm for Creatio application server](#)" block of articles.
- [Web farm](#) (multiple application servers).
A Redis or Redis Sentinel caching server / a Redis cluster (multiple Redis caching servers). Learn more about the Redis Sentinel mechanism in the "[Set up Redis Sentinel](#)" article.
- A database server or a database cluster (multiple database servers).
- Version control system server (optional). Learn more about setting up a version control system server in the "[Set up version control for a development environment](#)" article.

Fig. 2. The architecture of the main Creatio application when using a web-farm, a Redis cluster, and a database cluster



You can enable additional features by setting up **components and services** in the main Creatio application.

Application server

The application server performs the main computing work of the system.

The application on the [.NET Framework](#) platform runs under [Internet Information Services \(IIS\)](#). Windows OS, starting from Windows 7, is required for the operation of Creatio application servers on the .NET Framework platform. The .NET Framework Creatio application server consists of the application loader (WebAppLoader) and the Creatio configuration part (WebApp).

The purpose of the application loader is executing the service functions of Creatio and then redirecting users to the main Creatio application.

The application loader accounts for the following:

- User authorization.
- License verification and user authentication.
- Running the scheduler, which enables the execution of background tasks based on a schedule.

On the file system level, the loader is located in the root folder of the application.

After the application loader processes the authentication query, the users can work with the configuration part. The configuration part is an application that implements a specific configuration in Creatio and accounts for the business logic. On the file system level, the configuration part is located in the *Terrasoft.WebApp* folder.

The application on the [.NET Core](#) platform runs under [Kestrel](#). OS X, the last official version, is required for the operation of Creatio application servers on the .NET Core platform. The Creatio application on .NET Core is monolithic and handles the function of both the application loader and the configuration part. You can learn more about Creatio products of the .NET Core platform in the “[Creatio on the .NET Core platform](#)” article.

The database server

The database stores the following data:

- User data.
- Data required for Creatio operation.
- Configuration settings that define the functions of the product.

You can use the following [database management systems \(DBMS\)](#):

- MS SQL Server 2012 SP3 or higher.
- Oracle DBMS 11g Release 2 and 12c (when deploying on-site).
- PostgreSQL 11. PostgreSQL setup files are available for download at [postgresql.org](#).

The setup options for Creatio products are available in Table 1.

Table 1. The setup options for Creatio products

Creatio products	DBMS			
	MS SQL	Oracle	PostgreSQL	PostgreSQL (.NET Core)
Marketing	+		+	+
Sales Enterprise	+		+	+
Sales Commerce	+		+	
Sales Team	+		+	
Service Enterprise	+		+	+
Customer Center	+		+	
Studio	+		+	+
Lending	+		+	
Bank Customer Journey	+		+	
Bank Sales	+		+	
Sales Enterprise & Marketing & Service Enterprise	+	+	+	+
Sales Enterprise & Marketing & Customer Center	+		+	
Sales Commerce & Marketing & Customer Center	+		+	
Sales Team & Marketing	+		+	
Sales Team & Marketing & Customer Center	+		+	+
Bank Sales & Bank Customer Journey & Lending & Marketing	+	+	+	+

Redis caching server

The Redis caching server is responsible for the following tasks:

- Storage of user and application data (user profile, session data, etc.).
- Storage of cached data
- Data communication between web farms.

For these tasks, data repository technology is implemented in Creatio's architecture. The object model of classes, being a unified API for access from application to data, located in an external repository forms the base of this technology. Creatio uses Redis as the external repository.

Creatio repositories focus on the execution of service functions or arrangements of data handling but they also can be used in configuration business logic for solving user tasks.

The data model of Redis is based on "key-object" pairs. Redis supports access only with a unique key and can be successfully used for the storage of serialized objects. Data, placed into the repository, is stored as binary serialized objects in Creatio.

Redis supports the following data storage strategies:

- **In-memory only.**
Redis is used simply as a caching layer making a copy of the persistent database available in-memory. No data storage.
- **RDB persistence (default).**
A snapshot is saved to disk at specified intervals (normally once in 1-15 minutes) based on the time the previous copy has been created and the number of changed keys.
- **AOF persistence.**
Redis logs each write operation synchronously to an append-only log file.
- **Replication.**
You can assign each Redis server with a master server. All changes in the master Redis instances will be reproduced in the replica instances on the slave servers.

The data storage method is determined by configuring the Redis server. At present, Creatio supports RDB persistence but no Redis replication.

More information about the Redis caching server is available in the [Redis documentation](#).

Creatio supports two types of repositories, i.e. data and cache repositories. Learn more about data repositories and caching in the “**Repositories. Types and recommendations on use**” article. The object model is covered in the “**Repositories. Types and recommendations on use**” article.

A data repository is designed for data warehousing of rarely modified long-term data. A cache repository stores operation data.

Individual logical levels of data placement for each repository type are described in Tables 2 and 3.

Table 2. Data repository levels

Level	Details
Request	The query level. The data of this level is available only in the course of current query processing. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Request</i> enumeration value.
Session	The session level. The data of this level is available only in the session of the current user. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Session</i> enumeration value.
Application	The application level. The data is available for the entire application. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Application</i> enumeration value.

Table 3. Cache levels

Level	Details
Session	The session level. The data of this level is available only in the session of the current user. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Session</i> enumeration value.
Workspace	The level of the workspace. The data of this level is available for all users of the same workplace. Corresponds to the <i>Terrasoft.Core.Store.CacheLevel</i> enumeration value.

Application The application level. The data of this level is available for all application users regardless of their workspace. Corresponds to the *Terrasoft.Core.Store.CacheLevel* enumeration value.

Such partitioning of repositories is implemented for logical separation of data units in the repository and convenience of further use in the source code. The partitioning accounts for:

- Isolation of data between workspaces and user sessions
- Conditional classification of data
- Control of the data life cycle

All repository and cache data can be located physically on an abstract data storage server. The data in the Request level repository are an exception and are stored in the memory directly.

Creatio uses Redis as the repository server. In common cases, it may be a user repository, accessed through unified interfaces. It is necessary to take into account the fact that repository access operations are resource-intensive since they are associated with serialization/deserialization of data and network communication.

The data repository and cache represent the following possibilities for data handling:

- Access to data by key for reading/recoding.
- Deletion of data from the repository by key.

The key difference between a data repository and the cache is the approach to the lifetime of the contained objects.

Data will be stored in the storage until deleted explicitly. The lifetime of such objects is limited by the query execution time (for the data in a Request level repository) or session existence time (for the data in a Session level repository).

Cached data have an aging period that limits the time a specific cached element remains valid. All items are cleared from the cache regardless of their aging period in the following circumstances:

- Upon the session end (data in the cache and repository of the Session level).
- Upon explicit deletion of the workspace (data in the cache of the Workspace level).

The items of the Application level cache are stored for the entire period of existence of the application and can be deleted only by clearing the external repository.

Data can be cleared from the cache at any time. This may result in situations when a program tries to access the cached data that have already been cleared from the cache by the time the attempt is made. In this case, the calling code should only receive data from the persistent repository and place them in the cache.

Version control system server (SVN)

A version control system server is an optional application component. Use a version control system server when developing custom configurations in parallel with the normal operation of the application. Version control is, however, required for multi-user development.

The version control server accounts for the following functions:

- **Transferring changes between applications during the development.**
Changes are transferred via packages stored as sets of folders and files on the file system level. Learn more about working with packages in the **Development tools. Packages** block of articles.
- **Storing the configuration status as packages of a specific version.**
The version control system stores all configuration elements developed in the packages.

The version control system stores the entire change history in the configuration. Learn more about setting up a version control system in the “**Create repository in SVN server**” article. Learn more about working with a version control server in the “**Version control system. Built-in IDE**” block of articles. Different version control systems are covered in the “**Version control systems**” block of articles.

Deployment options

There are two Creatio deployment options:

- On-site.
- Cloud (deploying the Creatio application on cloud).

On-site deployment

In On-site deployment, all expenditures associated with handling the servers (installation, configuration, maintenance, administration) are charged to the customer.

On-site deployment provides the following advantages:

- Fast and convenient development process.
- Independent development environments. Since development is made in a separate application, other applications cannot be affected.
- Using the version control system for saving and transferring changes.
- The ability to use an IDE and set up [continuous integration](#) processes.

The disadvantage of on-site deployment is that the customer bears the cost of supporting the infrastructure (update, administration, maintenance costs).

To deploy the Creatio application on-site, the server-side and the client-side must meet certain technical requirements. More information about the system requirements for client PCs is available in the "[Client-side system requirements](#)" article. More information about the system requirements for Creatio application servers is available in the "[Server-side system requirements](#)" article.

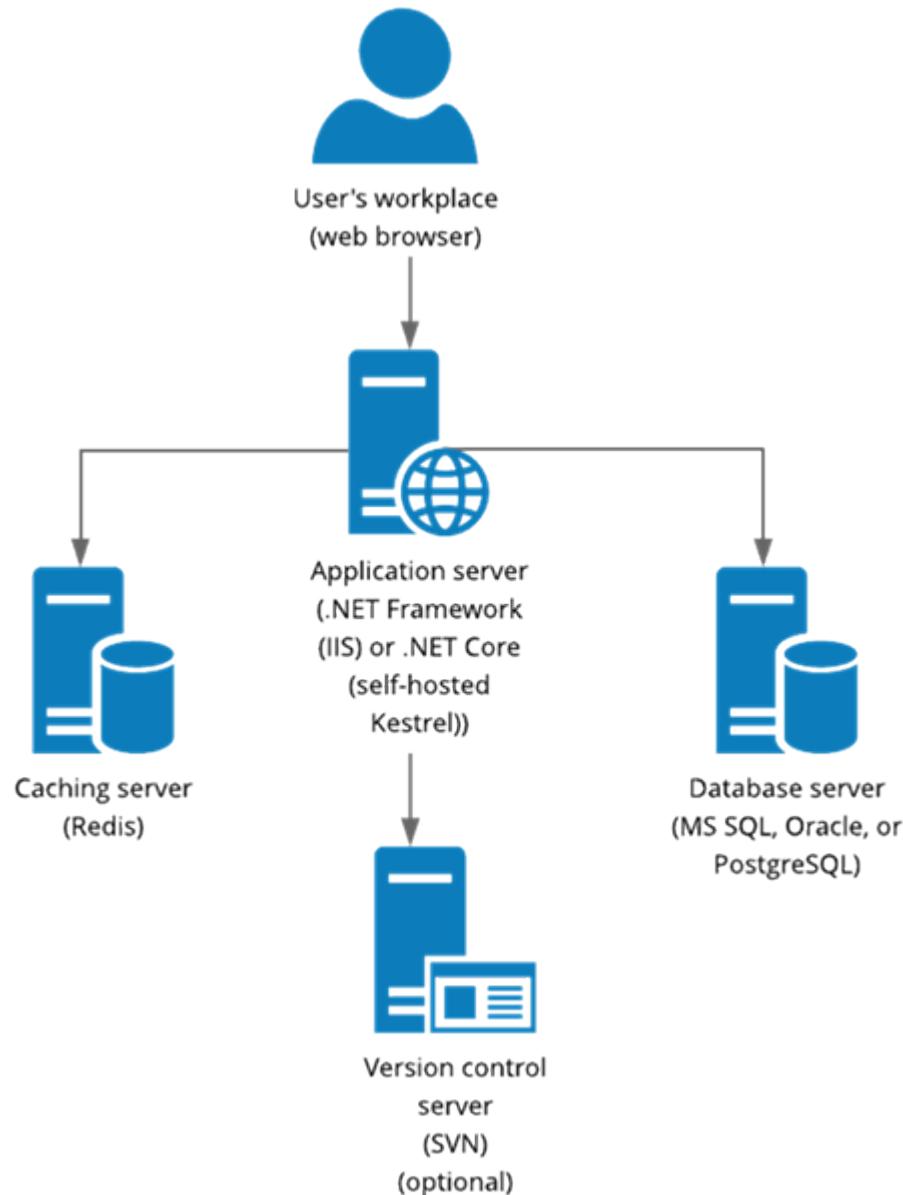
To calculate the parameters of the servers needed to deploy the application and additional components, use the [system requirements calculator](#).

A step-by-step guide for deploying Creatio on-site and setting up the application on Windows platforms is available in the "[Deploy Creatio .NET Framework application on Windows](#)" block of articles. A step-by-step guide for deploying Creatio on-site and setting up the application on Linux platforms is available in the "[Deploy Creatio .NET Core application on Linux](#)" block of articles.

The following on-site deployment options are available for Creatio applications:

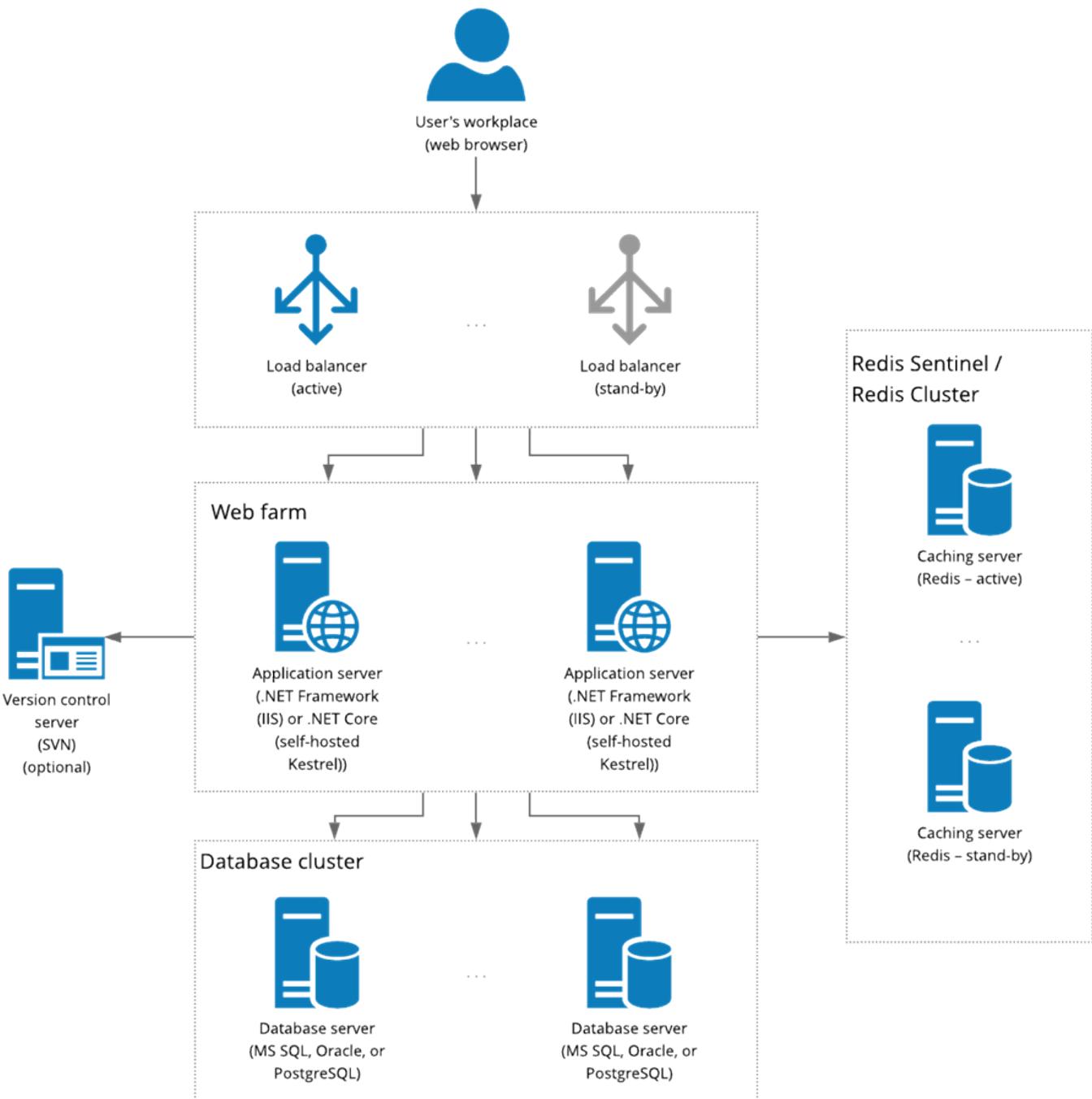
- Without fault-tolerance (Fig. 3).
- With fault-tolerance (Fig. 4).

Fig. 3. Creatio on-site deployment without fault-tolerance



You can use a version control server to set up a fault-tolerant Creatio development environment. However, such complex fault-tolerant systems are normally used in testing and production environments.

Fig. 4. Creatio on-site deployment with fault-tolerance



Cloud deployment

In the cloud mode, the application is deployed on cloud data center servers (Amazon, Azure) managed by Creatio. All application server part is stored in the data centers and administrated by Creatio employees. All issues related to administration, speed, or scalability are handled by Creatio. The customers only use the client part of the application.

Cloud deployment provides the following advantages:

- Timely updates.
- Maximal performance.
- Compliance with industry standards on data availability and security.

Learn more about the limitations of the on-cloud deployment of a Creatio application in the “**Deploying the Creatio cloud application**” article.

To deploy Creatio on-cloud, go to the [trial request page](#). During the 14-days trial period, you can become familiar

with the main features of the application. After the trial period is over, the demo version of the application can be moved to the main Creatio platform.

See Also

- **Deploying the Creatio on-site application**
- **Deploying the Creatio cloud application**
- [On-site deployment](#)
- **Components and services**
- **Developing solutions in Creatio**

Video tutorial

- [Creatio architecture basics](#)
- [Setting up a development environment](#)

Components and services

Contents

- **Mobile application**
- **Portal**
- **Process modeling (Studio Creatio, free edition)**
- **Global Search Service**
- **Bulk duplicate search service**
- **Website event tracking service**
- **Database enrichment service**
- **Machine Learning service**
- **Phone integration service**
- **Sync Engine synchronization service**
- **Exchange Listener synchronization service**
- **Static content bundling service**
- **Bulk email service**

Mobile application

Beginner

Easy

Medium

Advanced

Creatio's mobile applications are remote workplaces with instant access to customer data, calendar, mobile feed, etc. The mobile application is an auxiliary tool for accessing the primary Creatio application on mobile devices.

Introduction

Using the Mobile Creatio provides the following advantages:

- Quick access to data and information exchange between the employee and the management.
- Enhanced interaction of the company's employees and departments.
- The timely arrival of vital information.
- Swift reaction to the arriving information.
- An increase in customer loyalty thanks to the swift reaction.
- An increase in field staff productivity.

Mobile Creatio offers the following opportunities to the users:

- Working with the data of the primary Creatio application on a mobile device.
- The information is accessible even without an established Internet connection (hybrid and offline modes).

Mobile Creatio is implemented using the hybrid approach. A hybrid application is a web application wrapped in a native container. Unlike native applications, hybrid applications have a single codebase for every platform.

To customize Mobile Creatio (for example, change the section list, a set of business fields, the business logic settings, etc), set up the mobile application in the primary Creatio application. Learn more about customizing the Creatio mobile application in the “[Mobile application setup](#)” block of articles. Setting up the list of sections available in the mobile application is covered in the “[Mobile application wizard](#)” block of articles.

One of the steps required to set up the mobile application is to choose the operation mode. A Creatio mobile application has the following operation modes available:

- **Hybrid mode.**

The hybrid mode is designed for accessing the data when a stable connection to the Creatio server cannot be established. It is enabled automatically. This mode enables creating new records and working with schedules. Additionally, the most recent records (10 last records) are available for reading and editing when there is no Internet connection.

- **Online.**

The online mode requires an Internet connection. In this mode, the user works directly with the server (the primary Creatio application). The configuration settings are auto-synced in real-time.

- **Offline.**

The offline mode only requires an Internet connection for the initial imports and subsequent synchronizations. In this mode, the data are stored on the mobile device. To acquire the configuration changes and update the data, run a synchronization with the Creatio application server manually.

Learn more about the operation modes of Mobile Creatio in the “[Mobile application architecture](#)” article. The differences between the modes are covered in the “[Online/offline modes](#)” article.

The mobile application uses the DataService web-service to synchronize with the Creatio server. Learn more about Creatio integration using DataService in the “[DataService](#)” article.

If conflicts occur during the synchronization, the details will be logged to the synchronization log, available in the hybrid and offline modes. Learn more about working with the synchronization log in the “[Online/offline modes](#)” article.

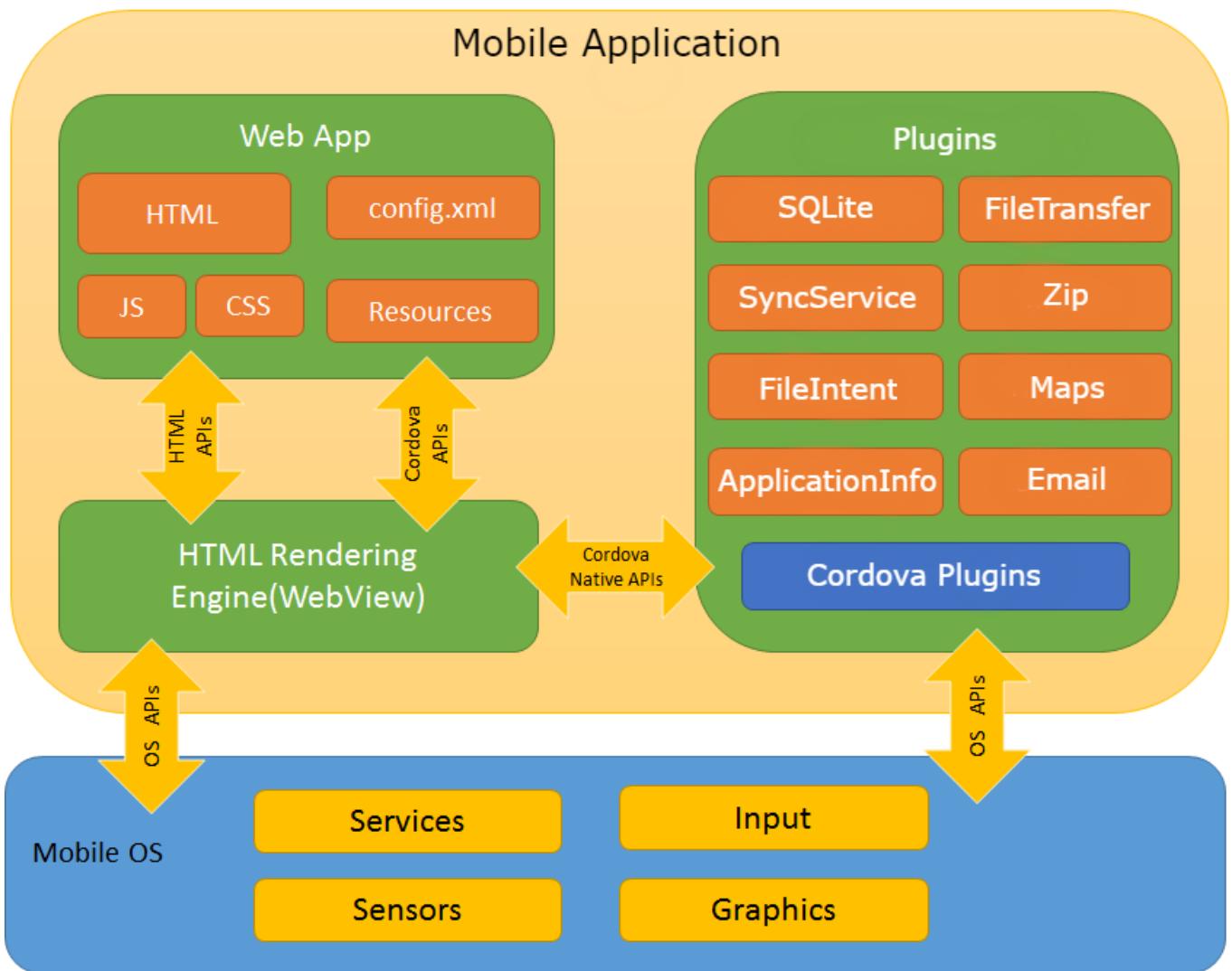
To check if the custom functionality works as intended, use the instructions from the “[Mobile application debugging](#)” article.

Schemas

Mobile application architechture schemas

The architecture of the mobile Creatio application is presented in Figure 1.

Fig. 1. Mobile Creatio architecture



The Creatio mobile application utilizes the [Apache Cordova](#) framework for creating hybrid applications. The Cordova framework has the following advantages:

- Access to native device APIs for interacting with the database or peripheral devices (such as the camera or memory card).
- Native plug-ins for using the APIs of multiple mobile platforms (iOS, Android, Windows Phone, etc.). Additionally, developing custom plug-ins enables adding extra features and extending the API. The list of supported platforms and core plug-ins is available in the [Cordova documentation](#).

Mobile Creatio's core provides a single interface that enables the interaction of the client parts of the mobile application. The JavaScript files that the core utilizes can be divided into basic and configuration scripts.

The basic scripts are part of the application package available in the application store. They include:

- MVC components (page layouts, controllers, models).
- Synchronization modules (data import and export, metadata import, file import, etc.).
- Client classes for web-services.
- Client classes for accessing native plug-ins.

The application downloads the configuration files during the synchronization to the Creatio application server and then saves them to the local file system. The configuration files include the manifest of Mobile Creatio, as well as section schemas and settings.

A manifest is a configuration object with properties that describe the structure (objects and their connections) of the mobile application. The manifest properties of Mobile Creatio consist of the following groups:

- Application interface properties (setting up the application sections, the main menu, and custom images). Learn more about the application interface properties in the [“Manifest. Application interface properties”](#)

article.

- Data and business logic properties (the description of imported data and the custom business logic for processing such data in the mobile application).
Learn more about the data and business logic properties in the "[Manifest. Data and business logic properties](#)" article.
- Application synchronization properties (setting up synchronization parameters for data-syncing with the primary application).
Learn more about application synchronization properties in the "[Manifest. Application synchronization properties](#)" article.

Learn more about the architecture of Mobile Creatio in the "[Mobile application architecture](#)" article.

Starting with Creatio version 7.16.4, the **[Approvals]** section of Mobile Creatio uses [Flutter Framework](#).

Mobile application operation scheme

The mobile Creatio application available in application stores is a set of modules required for synchronizing with Creatio servers. The working principles of Creatio Mobile are presented in Figure 2.

Fig. 2. The operation scheme of Mobile Creatio



Each product and each customer website may contain an independent collection of settings for Creatio Mobile, custom business logic, and custom visual interface. A Mobile Creatio user must first install the application and then synchronize it with the main application.

Mobile application compatibility with Creatio products

Mobile Creatio is part of the Creatio platform. The mobile application is available to the users of the primary Creatio application version 7.15 and up.

After the installation, the user specifies the connection parameter for a specific Creatio server. The application then downloads metadata (application structure, system data, etc.) and regular data. Such an architecture makes the mobile application compatible with all Creatio products.

However, [portal](#) users cannot use the mobile application.

Mobile application installation options

Creatio mobile application is available on:

- [App Store](#) – for iPhone and iPad running [iOS](#) version 8 and higher.
- [Google Play](#) – for mobile devices running [Android](#) version 4.4 and higher.

See Also

- [Mobile application](#)
- [Mobile Creatio development guide](#)

Video tutorial

- [Mobile app wizard](#)

Portal

Beginner

Easy

Medium

Advanced

Creatio portal is a low-code component that provides secure and managed access to Creatio data and functionality for internal and external users, customers, and partners. The interface and tools of the portal are the same as those of the main Creatio application. The portal development process has the same principles as development for the main application. Learn more about setting up the Creatio portal in the “[Getting started with Creatio portal](#)” article. Learn more about portal development is covered in the **Creatio development guide**.

Introduction

Creatio portal provides tools for solving a broad range of business tasks. Creatio portal is suitable for a variety of use cases, the most common being:

- **Customer self-service, such as in technical support.**

Give a self-service option to your customers and focus the time and expertise of your support agents on tasks more important than case registration. Empower your customers to submit support cases and track the resolution progress directly on the portal. Provide your customer access to your knowledge base articles to help them find answers quickly. Service multiple customers at once while avoiding queues and loss in productivity.

- **Communications with internal and external customers, for instance, an HR portal.**

Configure the ability to service external employees and contractors who do not actively use Creatio: create applications, submit them for approval, and track their progress. An HR portal can act as a central hub for all the essential company documents and policies that are in the public domain.

- **Interaction with external users (clients, dealers, and partners) at all sales stages.**

Create partner programs, process leads, and close opportunities along with your partners by using lead management and corporate sales processes. Keep track of the partner tiers, training sessions, and certified experts.

Through the portal, Creatio users can access selected [licensed](#) sections and their associated data. You can use standard [access permissions](#) to choose which of your business data is available on the portal, and make sure that any sensitive and confidential information is safe out of external users’ reach.

Portal users can:

- Create new records and modify the existing ones.
- Add notes.
- Attach files.
- Leave messages for the support service.
- Work using business processes.

In addition to the regular object permissions, the data available for portal users is limited by the **[List of objects available for portal users]** lookup. Only the objects included in the lookup are accessible via the portal UI. Learn more about setting up access permissions in the “[Access permissions on the portal](#)” article.

The most common scenario is the collaborative work of portal users and system users in the same section but with different sets of page fields. Columns that can be configured in the section list and columns for filtering (in quick filters or dynamic groups) are limited for portal users. When you add fields to the edit page using the Section Wizard, the columns are added to the **[List of schema fields for portal access]** lookup by default and become available. If a field is not available on the page, and you want it on the portal list, add the field to the lookup manually. You can choose which information will be displayed in each section list. Learn more about setting up the order of columns in the “[Working with the portal list](#)” article.

Portal users are grouped in the “**All portal users**” organizational role. You can create individual portal users or group them into an organization by connecting them to a specific account. Learn more in the “[Portal users](#)” article.

Learn more about restricting access to web services for portal users in the “**Restricting access to web services for portal users**” article.

Creatio portal is available in the following configurations:

- **Self-service portal.**

You can use the self-service portal both as the primary case registration channel and an auxiliary channel. A portal user can find information in the knowledge base or submit a support service request. This configuration limits the list of sections that portal users can access to the **[Case]** and **[Knowledge base]** sections. You can customize them using the system designer. You cannot add custom sections to the self-

service portal.

Learn more about the self-service portal in the “[Self-service portal](#)” article.

- **Customer portal.**

The customer portal is designed for process automation, e.g., providing services, confirming applications and service requests, etc. Portal users can initiate processes (e.g., create orders, cases, etc.) or participate in the (e.g., approve requests). This portal configuration type allows you to set up and use up to three custom portal sections. Create a custom section in the main application first, then make a portal version of the section. Portal users can also access the **[Case]**, and **[Knowledge base]** sections. Additionally, users of Creatio Bank Customer Journey can access the **[Applications]** and **[Contracts]** sections. You can also add the **[Documents]** section if the main Creatio application supports it. A user-created section built on a licensed base product object is not considered a custom section.

Learn more about the customer portal in the “[Customer portal](#)” article.

- **Partner portal.**

The partner portal is designed for companies that work with customers via partner networks. The partner portal is a joint communication platform for passing leads between partners and partner cross-sales. This configuration enables portal users to access the **[Partner program]**, **[Leads]**, **[Opportunities]**, **[Marketing activities]**, and **[Orders]** sections.

Learn more about the partner portal in the “[Partner portal](#)” article.

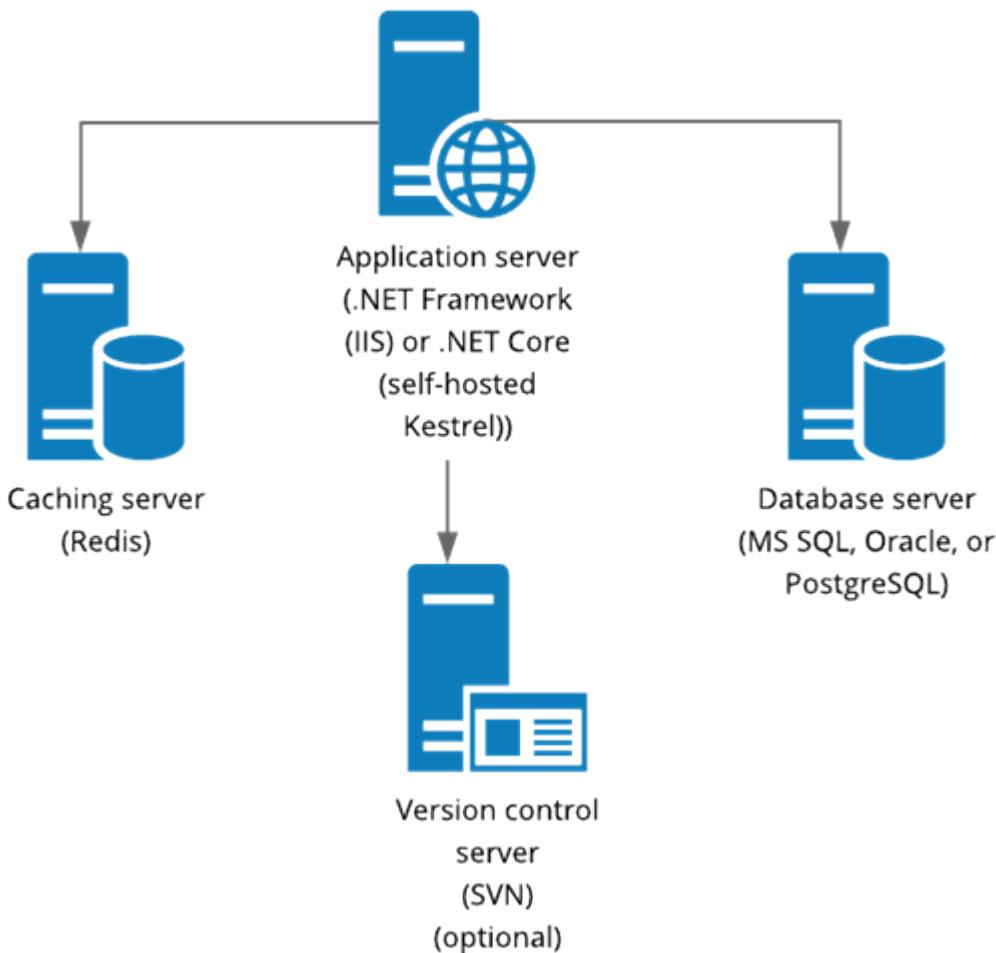
Portal configurations can be used simultaneously. Each configuration enables portal users to access the basic portal functionality (main page, user profile settings, knowledge base, data segmentation). The list of available sections in the Creatio portal depends on the portal configuration.

Learn more about working with the portal in the “[Working with portal](#)” block of articles.

Portal schema

The architecture scheme of the Creatio portal (Fig. 1) is the same as the architecture scheme of the main application. Learn more about the main architecture components in the “[Main application](#)” article.

Fig. 1. The architecture of the Creatio portal

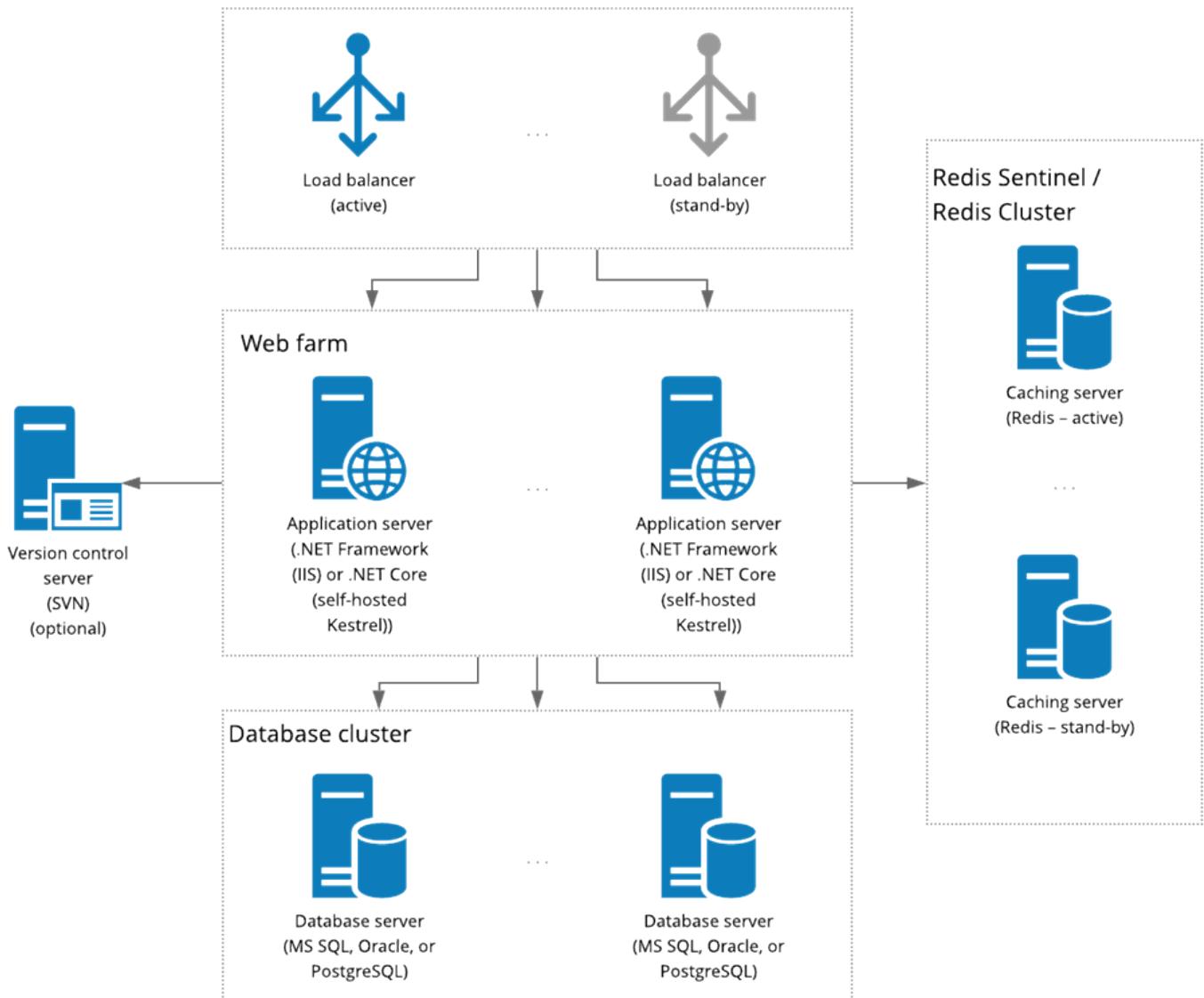


Portal scalability

You can enhance the performance of large-scale Creatio projects through [horizontal scaling](#). When using horizontal scaling, the architecture of the Creatio portal (Fig. 2) includes the following components.

- [Load balancers](#).
The load balancer may be either hardware or software. To work in fault-tolerant mode, use the HTTP/HTTPS traffic balancer that supports the WebSocket protocol. Learn more about setting up the HAProxy load balancer in the “[Set up a web-farm for Creatio application server](#)” block of articles.
- [Web farm](#) (multiple application servers).
- A Redis or Redis Sentinel caching server / a Redis cluster (multiple Redis caching servers). Learn more about the Redis Sentinel mechanism in the “[Set up Redis Sentinel](#)” article.
- A database server or a database cluster (multiple database servers).
- Version control system server (optional). Learn more about setting up a version control system server in the “[Set up version control for a development environment](#)” article.

Fig. 2. The architecture of the Creatio portal when using a web-farm, a Redis cluster, and a database cluster

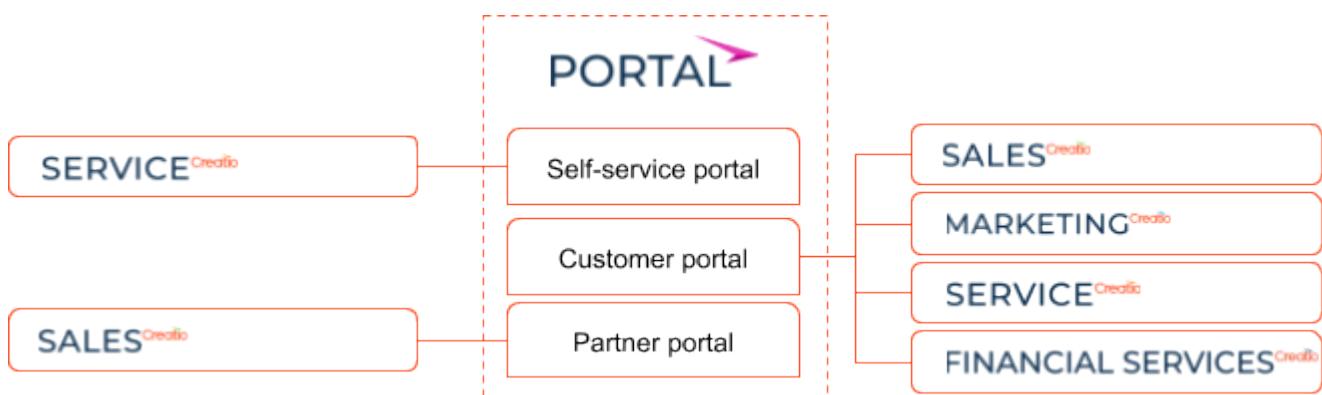


Learn more about secure access to the portal and the scheme for deploying a Creatio application with external network portal access in the [“Set up secure access to the portal”](#) article.

Portal compatibility with Creatio products

The compatibility of portal configurations with the Creatio products is illustrated in Figure 3.

Fig. 3. Compatibility of portal configurations



Portal deployment options

Creatio portal is available for applications deployed both on-site and in the cloud.

To ensure the safety of data, the on-site application must be deployed using a web farm when installing the portal. Learn more about setting up a web farm in the “[Set up a web-farm for Creatio application server](#)” article. Learn more about setting up a Creatio application with external network portal access in the “[Set up secure access to the portal](#)” article.

See Also

- [Portal documentation](#)
- [Self-service Portal](#)

Video tutorial

- [Portals](#)

Process modeling (Studio Creatio, free edition)

Beginner

Easy

Medium

Advanced

Studio Creatio, free edition (Studio Creatio free) is a collaborative business process modeling tool. Business processes in Studio Creatio free correspond to the [BPMN 2.0](#) specification, which is easily understandable by all business users, from analysts to developers.

Introduction

Studio Creatio free can help you with the following tasks:

- **Accelerate business process modeling.**
Build process diagrams using simple visual tools and save them in the process library. A flexible structure, convenient navigation, and a single control environment will help to organize the business processes of the company swiftly and accounting for all requirements.
- **Standardize your business process management.**
The multifunction process designer will ensure compliance with the BPMN 2.0 specification. This makes it possible to build processes that are understandable by customers, business analysts, developers, and end users. Meanwhile, *.bpmn support enables import and export to share business processes with other applications.
- **Arrange collaborative process modeling.**
Involve your coworkers or outside experts, edit business processes along with them in real-time, and share links to the processes for remote viewing and commenting.
- **Facilitate documenting business processes.**
Enter information and add all necessary explanations to the business process as you build the diagram. Creatio enables users to download the description of the process as a *.pdf file to have comprehensive documentation on the process for further use outside Studio Creatio free.

Studio Creatio free provides the following tools:

- **Process designer.**
The process designer with BPMN 2.0 support provides universal tools for designing business processes of any complexity using a visual designer.
Learn more about working with the process designer in the “[Process designer](#)” article.
- **Process library.**
All business processes of the company are stored in the process library, which enables you to add new business processes to your library by [creating](#) or [importing](#) them, organize business processes in a hierarchical structure, and searching folders and processes.
Learn more about working with the process library in the “[Process library](#)” article.

Studio Creatio, free edition compatibility with Creatio products

Studio Creatio free is available as a separate product and can exchange data with all Creatio products utilizing business process export (*.bpmn, *.svg, and *.png) and import (*.bpmn).

Studio Creatio, free edition deployment options

Studio Creatio free is only available in Creatio's cloud, lacks an open API, and cannot be customized by customers.

See Also

- [Documentation for Studio Creatio, free edition](#)

Video tutorial

- [Studio Creatio functionality](#)
- [Business process management in Creatio](#)

Global Search Service

Beginner

Easy

Medium

Advanced

The Global Search Service is created for integration of the [ElasticSearch](#) engine with the Creatio.

Use the global search service to quickly search data in the main Creatio application by entering a search query in the search string. Creatio always searches in all sections (including custom sections).

Introduction

The search service implements recording and transport functions by doing the following:

- Subscribes clients by creating an index in ElasticSearch and saves the connection between the index and the application.
- Disconnects clients by removing their index in ElasticSearch.
- Participates in the indexing process by retrieving data from the application database (DB).

The Global Search Service has the following distinguishing features:

- The records are searched by their text and lookup fields as well as the **[Addresses]**, **[Communication options]**, and **[Banking details]** details.
- Files and links on the **[Attachments and notes]** tab of the record page can be found by their name or description.
- Search requests are processed taking into account common typos and morphology of different word forms in English (other languages are not currently supported). The search query is case-insensitive.
- The search results are ranked by relevance both in the actual results list and with any configured filters. For example, if the search is performed from a section, the records of this section are displayed at the beginning of the results list.
- If a user does not have permissions for a specific object column, such a column is not displayed on the page of global search results.

Use the following system settings to set up global search parameters:

- **[Global search default entity weight]** and **[Global search default primary column weight]** – to set up the rules for displaying search results.
- **[Display search results with partial match]** – to display search results taking morphology, typos, and fuzzy matches into account.
- **[Match threshold for displaying in search results (percent)]** – to manage the number of displayed search results with partial match and increase the chances of finding data for inaccurate search requests.

Learn about system settings in more detail in "[The \[System settings\] section](#)" article.

Global Search Service schema

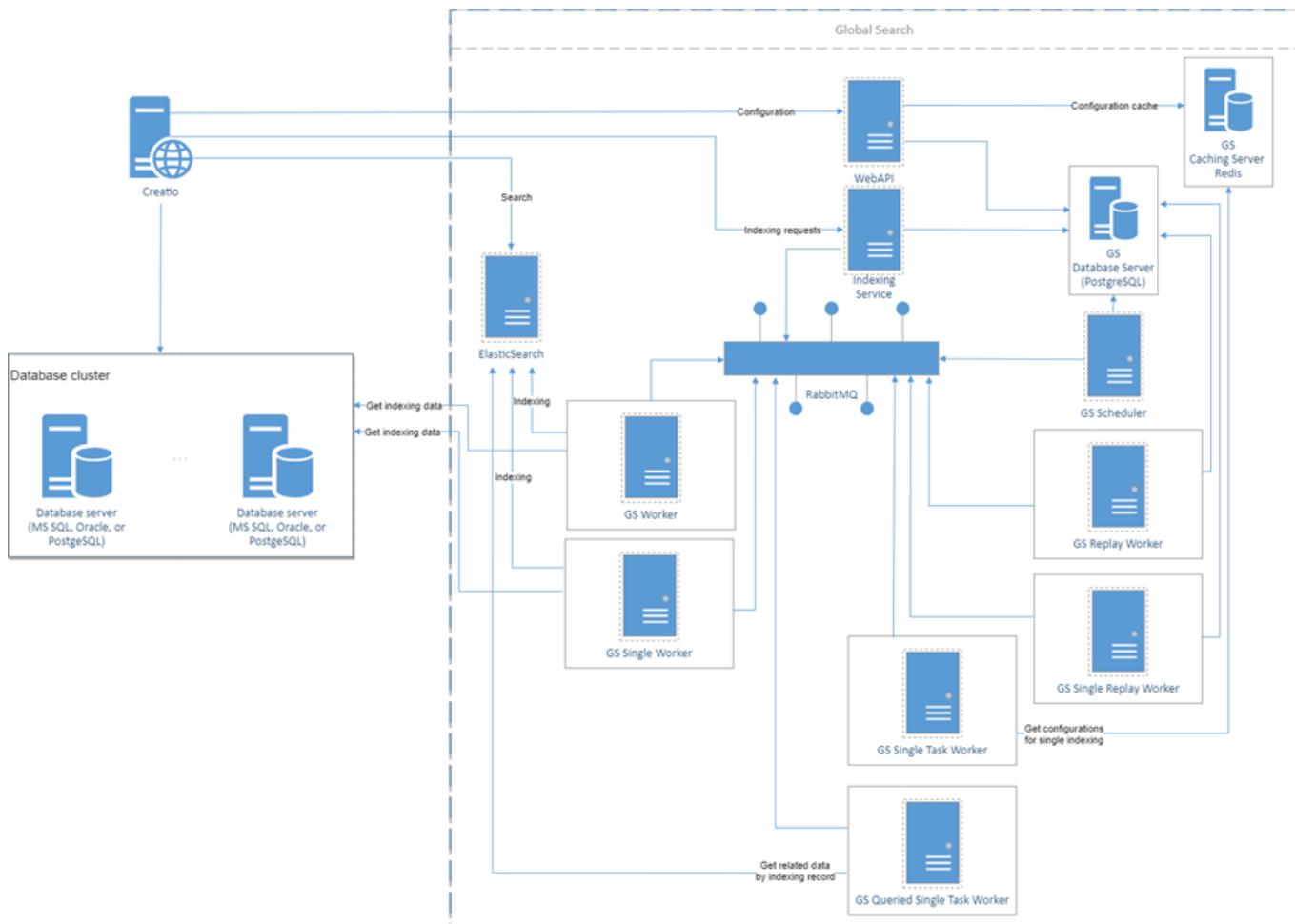
Global search service consists of the following components:

- RabbitMQ – message broker.
- ElasticSearch – a search engine.
- GS Database Server – database for configuring the global search component.
- GS Caching Server Redis – database used for caching and speed.
- WebAPI – web-service for global search component configuration.

- Indexing Service – web service for processing the queries for the targeted indexing of Creatio data.
- GS Scheduler – scheduler for indexing data from Creatio to ElasticSearch.
- GS Worker – component for indexing data from Creatio to ElasticSearch as per the GS Scheduler tasks.
- GS Replay Worker – component for processing indexing results (GS Worker operation results).
- GS Single Worker – component for targeted indexing of business process data in ElasticSearch upon a request from the business process.
- GS Single Replay Worker – component for handling exceptions when processing targeted indexing results (GS Single Worker operation results).
- GS Single Task Worker – component for scheduling tasks for GS Single Worker.
- GS Queried Single Task Worker – component for generating tasks for GS Single Worker.

The working principles of the global search service are presented in Figure 1.

Fig. 1. The operation scheme of the global search service



Global Search Service scalability

Database clustering enables scaling of the global search service in large projects. Learn more about Elasticsearch clustering in the [official documentation](#).

Global Search Service compatibility with Creatio products

The global search service features several versions: 1.4, 1.5, 1.6, [1.7](#), and [2.0](#). Each version is compatible with all Creatio products of version 7.10 and up.

Global Search Service deployment options

You can deploy the global search service on-site and in the cloud.

On-site applications require a preliminary setup of the global search service. To set up the service, you need two

servers (physical or virtual machines) that meet specific system requirements. More information about the system requirements for the servers is available in the “[Server-side system requirements](#)” article. Both servers must run under Linux with [Docker](#) installed. You can find the list of supported Linux distributions in the [Docker documentation](#).

We recommend that you install the most up-to-date version of the global search service.

See Also

- [Set up global search](#)
- [Global Search](#)

Bulk duplicate search service

Beginner

Easy

Medium

Advanced

Bulk duplicate search is a third-party service for bulk deduplication of Creatio section records.

Duplicate records may appear in Creatio whenever users add new records to system sections. Finding and merging duplicates helps maintain the quality of your data in any Creatio section.

Introduction

You need the global search service set up and configured using [ElasticSearch](#) to ensure the operation of the bulk duplicate search service. Learn more about the global search service in the “**Global Search Service**” article.

Creatio implements the following duplicate search modes:

- Bulk duplicate search – check for duplicates is run for the entire database. Launched manually or automatically.
- Duplicates search when saving a record – checks for duplicates for a particular record. It is run automatically when a new record is added and saved in a section.

Additionally, you can manually merge any records in a section, even if they were not flagged as duplicates. This option is available for all system sections. By default, duplicate search is available in the [Accounts], [Contacts] and [Leads] sections. In Creatio, the duplicate search is executed with the help of pre-configured rules. Creatio also provides customization of out-of-the-box duplicate search rules for contacts, accounts, and leads. Create custom rules for any Creatio section, including custom sections.

The bulk duplicate search function is pre-enabled in Creatio applications deployed in the cloud. Creatio applications deployed on-site require the global search service set up and configured before the bulk duplicate search service can be enabled. Learn more about the global search service in the “**Global Search Service**” article.

To connect bulk duplicate search to Creatio, take the following steps:

1. Set up the **[Deduplication service api address]** system setting value. Learn about system settings in more detail in “[The \[System settings\] section](#)” article.
2. Set up the **[Duplicates search]** operation permissions. Read more about managing access permissions in the “[Managing object operation permissions](#)” article.
3. Run the SQL script to enable the bulk duplicate search functionality in Creatio (*BulkESDeduplication*, *ESDeduplication*, *Deduplication*). Learn more about working with additional options in the “**Feature Toggle. Mechanism of enabling and disabling functions**” article.
4. Restart the Creatio application.

More information enabling the bulk duplicate search service is available in the “[Set up bulk duplicate search](#)” article.

Bulk duplicate search service schema

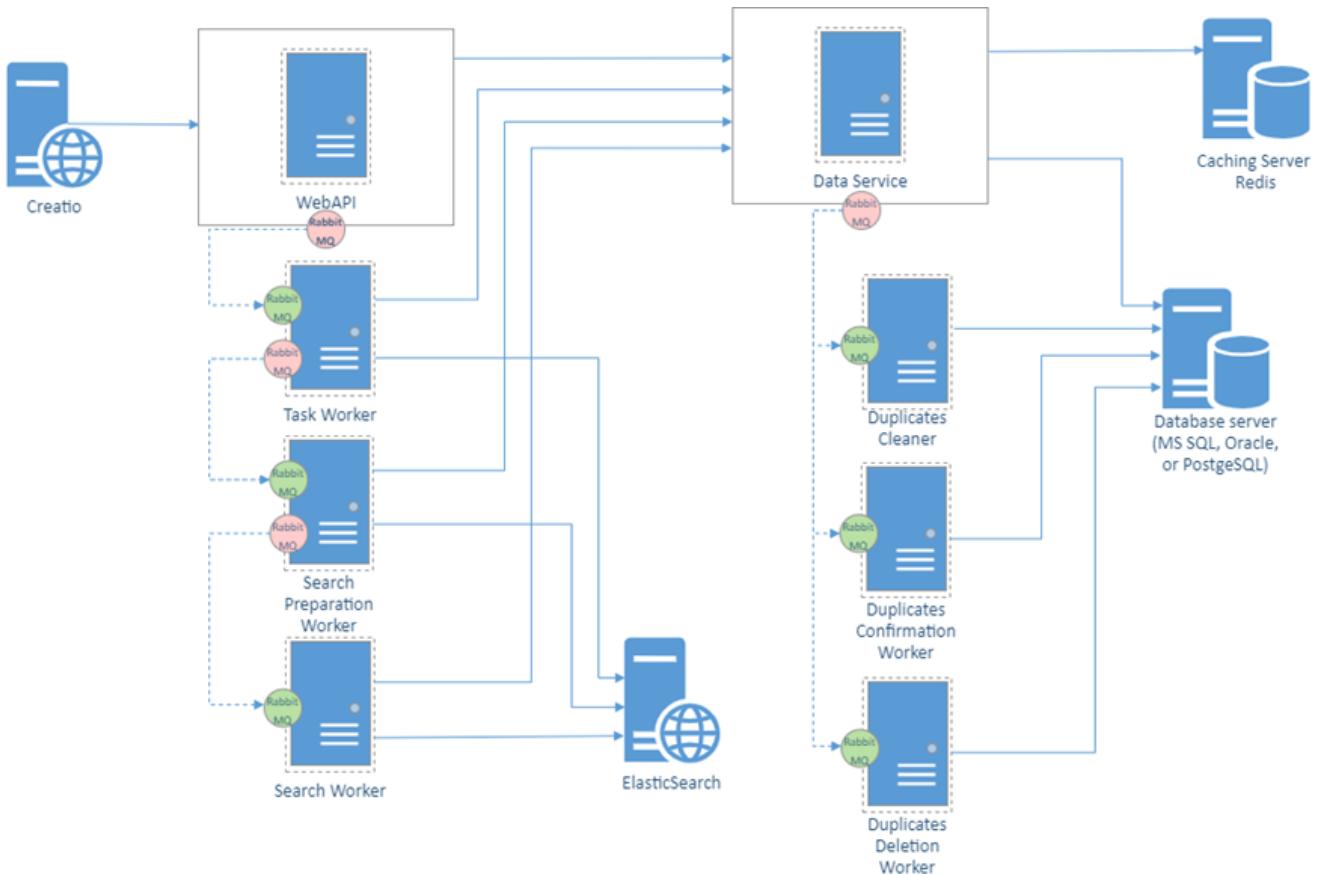
Bulk duplicate search service consists of the following components:

- RabbitMQ – message broker. Bulk duplicate search service component.
- ElasticSearch – a search engine. Bulk duplicate search service component.
- Redis – repository used for caching and speed.
- MongoDB – document-oriented DBMS.
- WebAPI – web service for communicating in the main Creatio application.

- Data Service – internal service for communication with a MongoDB component.
- Duplicates Search Worker – duplicate search component.
- Duplicates Deletion Worker – targeted duplicate deletion component.
- Duplicates Confirmation Worker – component for grouping and filtering the detected duplicates based on their uniqueness.
- Duplicates Cleaner – component for clearing the duplicates.
- Deduplication Task Worker – component for setting the deduplication task.
- Deduplication Preparation Worker – component for preparing the deduplication process. This component generates queries for duplicate search according to the rules.

The working principles of the bulk duplicate search service are presented in Figure 1.

Fig. 1. The operation scheme of the bulk duplicate search service



Bulk duplicate search service scalability

Database clustering enables scaling of the bulk duplicate search service in large projects. Learn more about Elasticsearch clustering in the [official documentation](#).

Bulk duplicate search service compatibility with Creatio products

The bulk duplicate search service features several versions: 1.0-1.5, 2.0. Each version is compatible with all Creatio products of version 7.14 and up.

Bulk duplicate search service deployment options

You can deploy the bulk duplicate search service on-site and in the cloud.

On-site applications require a preliminary setup of the global search service. Learn more in the [“Set up global search”](#) article. To set up the bulk duplicate search service, you need a server (a physical or virtual machine) that meets specific system requirements. More information about the system requirements for the server is available in the [“Set up bulk duplicate search”](#) article. Both servers must run under Linux with [Docker](#) installed. You can find the list of supported Linux distributions in the [Docker documentation](#).

We recommend that you install the most up-to-date version of the bulk duplicate search service.

See Also

- [Setting up bulk duplicate search](#)
- [Adding a duplicate search rule](#)
- [Adding a rule for duplicates search when saving a record](#)
- [Deduplication](#)

Website event tracking service

Beginner

Easy

Medium

Advanced

The website tracking service enables to track events from on the customer's site and pass them to the tracking cloud for further processing and display in Creatio.

Introduction

The website tracking service enables building analytics and business processes based on custom scenarios. The tracking service is integrated into the Creatio platform. In the future, we plan to introduce interaction with all Creatio modules to enrich the data and interact with business processes.

The website event tracking service offers the following opportunities to the users:

- Collect data about page views, visitor numbers, traffic source, device properties, etc.
- Collect data about user activity, goal achievements, conversion rates, routes, and use cases for websites and mobile applications.
- Deanonymize page views.

To set up website activity tracking, make the preliminary settings in Creatio, then embed the tracking JavaScript code into the HTML source code of each page on the website. As a result, Creatio will receive information about all redirections of potential customers to the website and their activity. The tracking code is triggered each time a customer performs a tracked activity on the website.

The tracking code generates the *[BpmTrackingId]* cookie files, which stores unique customer session IDs. This enables Creatio to gather information about customer's website events, both before and after actual registration.

Learn more about setting up the website event tracking service of version 1.0 in the "[How to set up website event tracking](#)" block of articles.

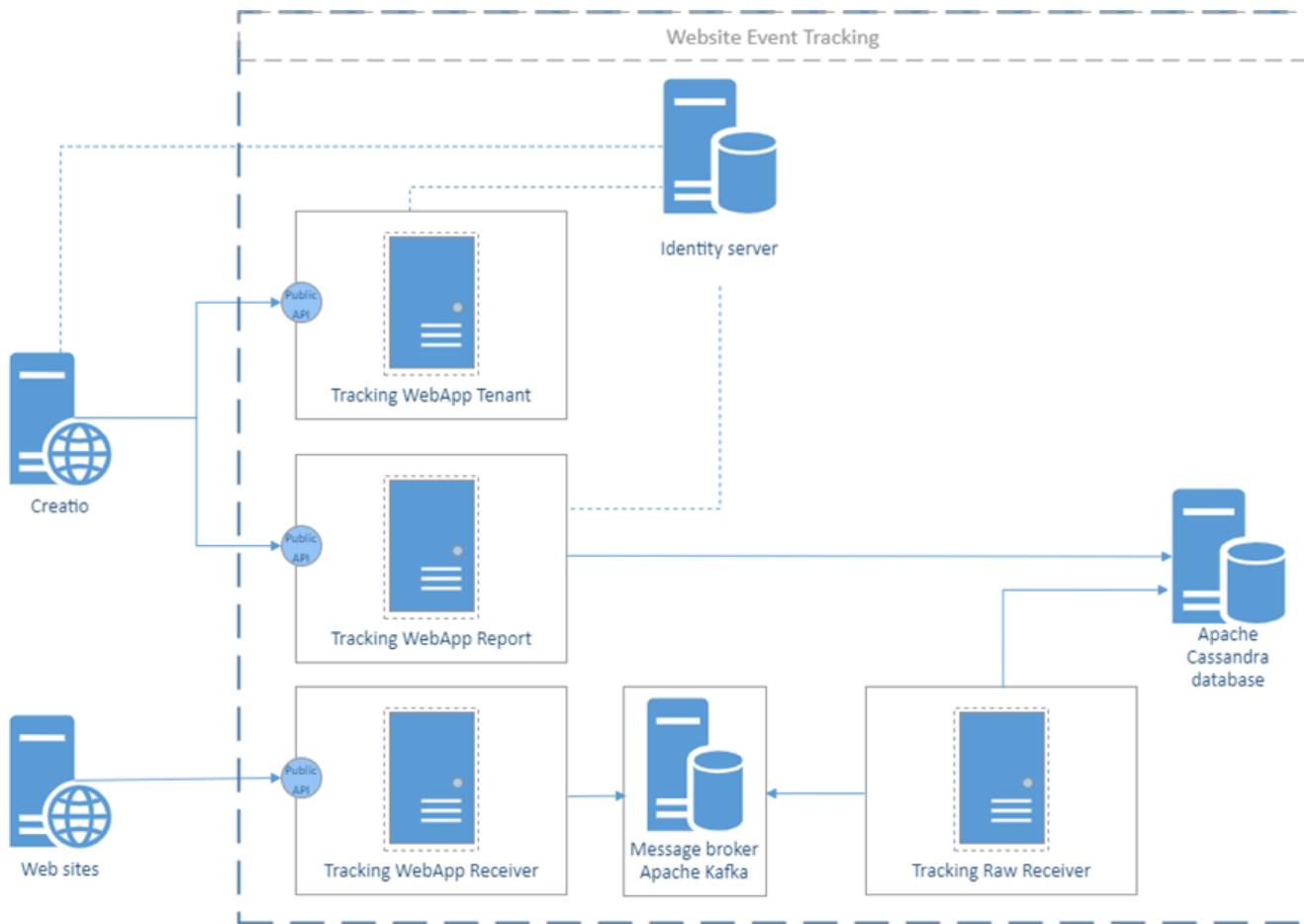
Website event tracking service schema

The website event tracking service of version 2.0 is built on the microservice principles and comprises of the following components:

- Identity server – used to identify the Creatio application in the website event tracking service.
- Tracking WebApp Tenant – responsible for storing the metadata and settings of the connected primary Creatio application.
- Tracking WebApp Report – used to select reporting data on request.
- Tracking WebApp Receiver – used to receive incoming events from web-sites.
- Message broker Apache Kafka – used for data exchange between components.
- Tracking Raw Receiver – used to receive raw tracking data.
- Apache Cassandra database – a distributed NoSQL database management system designed for building highly scalable and reliable repositories capable of storing huge amounts of hashed data. Stores the tracking data. To enable storing huge amounts of data, you can scale the website event tracking service and create new clusters (Nodes).

The working principles of the website event tracking service are presented in Figure 1.

Fig. 1. The workflow of the website event tracking service of version 2.0



Learn more about setting up and working with the website event tracking service of version 1.0 in the "[Website event tracking](#)" block of articles.

Website event tracking service scalability

The microservice architecture of the website tracking service of version 2.0 enables scalability via the standard means available in every component, such as clustering, horizontal scaling, and vertical scaling.

Website event tracking service compatibility with Creatio products

The website event tracking service of version 1.0 is compatible with Creatio Marketing version 7.14 and up.

The website event tracking service of version 2.0 is compatible with Creatio Marketing version 7.16.4 and up.

Website event tracking service deployment options

The website event tracking service of version 2.0 can be deployed in the cloud. At the moment, on-site deployment of the service is not implemented.

See Also

- [Website event tracking](#)

Database enrichment service

Beginner

Easy

Medium

Advanced

The database enrichment service has the following use cases:

- **Account data enrichment service.**

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

- **Email-based contact enrichment service.**

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

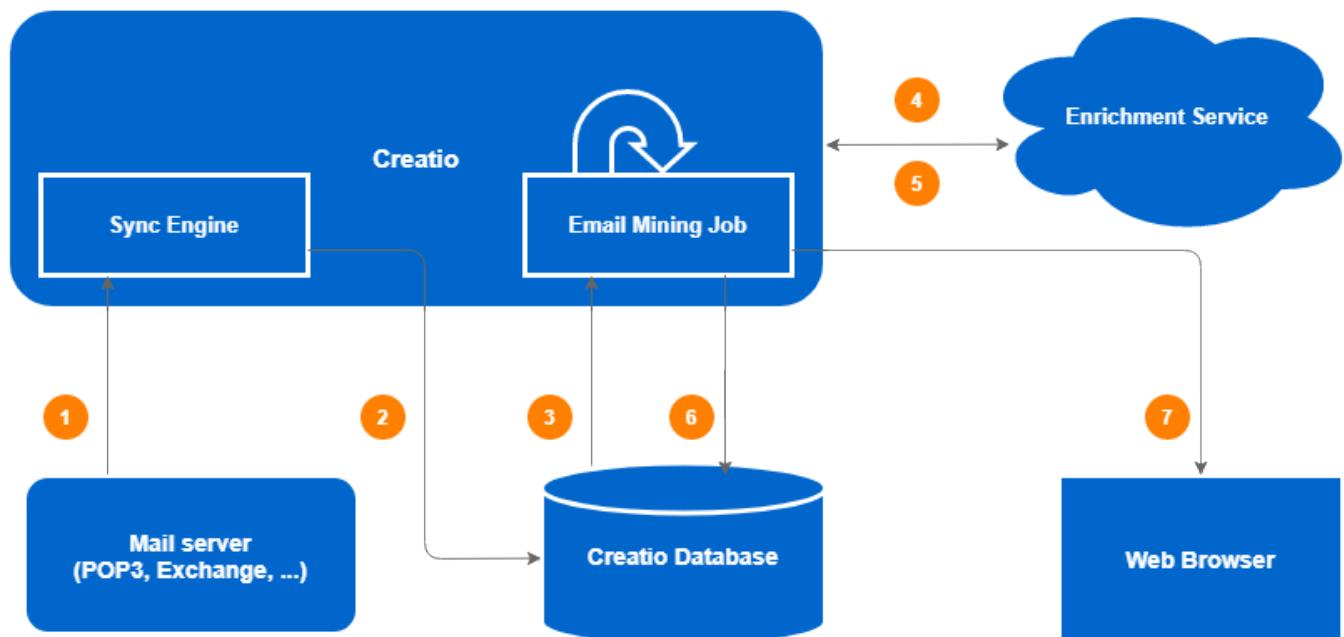
Database enrichment service workflow

The working principles of the email-based data enrichment service:

1. The current **Sync Engine mechanism** synchronizes with the email server. The email server sends new messages to Sync Engine.
2. Sync Engine saves the messages in the database as *Email* activities.
3. The Creatio task scheduler occasionally launches the *Email Mining Job* process. The mining process extracts a batch of the newest unprocessed *Email* activities from the database. Then, the process extracts the message body and the corresponding format (plaintext or HTML) from each activity.
4. The *Email Mining Job* process sends an HTTP query to the cloud *Enrichment Service* for each selected email.
5. The *Enrichment Service* performs the following operations:
 - Extracts a chain of individual messages (replies) from the email message.
 - Extracts the signature of each message.
 - Performs entity extraction on the signature data: contact (name), phones, email and web addresses, social profiles, other communication options, physical address, the name of the organization.
6. The *Email Mining Job* process parses the structure returned by the enrichment service and stores it raw in the Creatio database.
7. The *Email Mining Job* process sends a notification about the email data mining completion. The notifications about processed emails are sent to the communication panels of the users via WebSocket.

The working principles of the email-based data enrichment service are presented in Figure 1.

Fig. 1. The working principles of the email-based data enrichment service



Database enrichment service compatibility with Creatio products

The data enrichment service is compatible with all Creatio products of version 7.10 and up.

Database enrichment service deployment options

You can use the data enrichment service both on-site and in the cloud.

A personal cloud key and the URL to Creatio cloud services are required to use data enrichment.

Use the following system settings to specify these values:

- **[Account enrichment service URL]** – by default, this setting is populated for all applications.
- **[Text parsing cloud service]** – set up contact data enrichment service URL.
- **[Creatio cloud services API key]** – this setting is populated by default for all cloud applications. To set up data enrichment on-site, request a personal key from the Creatio support, and use the key to set up the service. Learn more about setting up the data enrichment service on-site in the “[Set up data enrichment](#)” article.

See Also

- **Contact data enrichment from emails**
- [Set up data enrichment](#)
- [Data enrichment](#)

Machine Learning service

Beginner

Easy

Medium

Advanced

The machine learning (lookup value prediction) service uses statistical analysis methods for machine learning based on historical data. For example, a history of customer communications with customer support is considered historical data in Creatio. The message text, the date, and the account category are used. The result is the **[Responsible Group]** field.

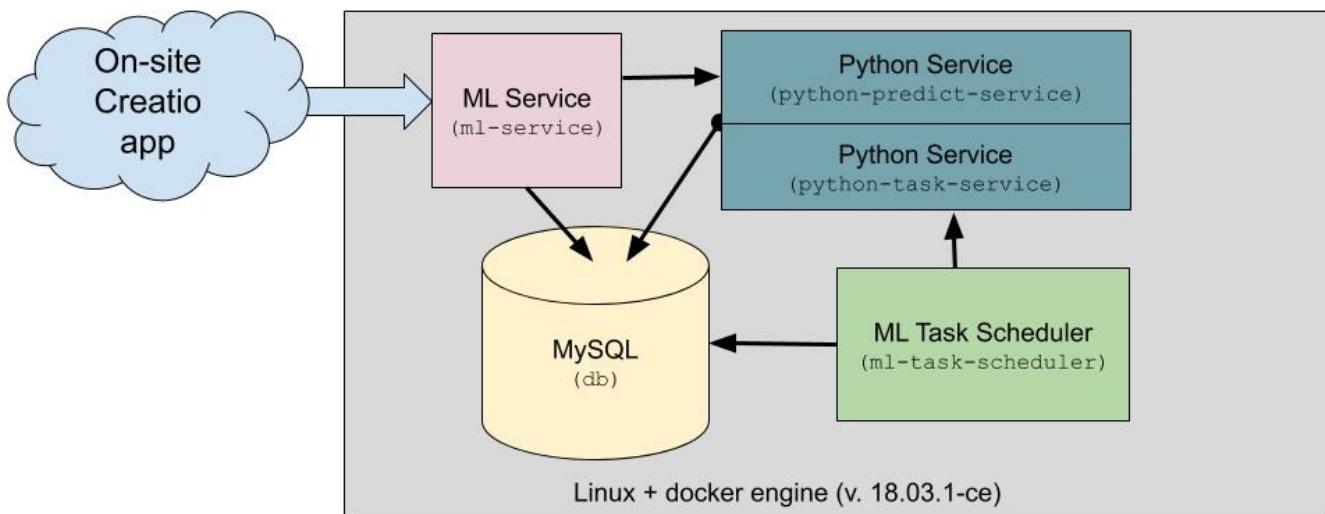
Machine Learning service workflow

The machine learning service consists of the following components:

- ML Service – machine learning web service. The only component enabling external access.
- Python Engine – a service wrapper for open-source machine learning libraries.
- ML Task Scheduler – task scheduler.
- MySQL – MySQL database. You can access it via the standard 3306 port.

The working principles of the machine learning service are presented in Figure 1.

Fig. 1 – Machine learning service workflow



There are two stages of model processing in Creatio: training and prediction.

The prediction model is an algorithm that builds predictions and enables the system to automatically make decisions based on historical data.

Training

The ML model is “trained” at this stage.

Main training steps:

1. Establish a session for data transfer and training.
2. Sequentially select a portion of data for the model and upload it to the service.
3. Request to include a model into a training queue.
4. *ml-task-scheduler* processes the queue.
5. *python-task-service* performs model training and writes the parameters to the database.
6. Creatio occasionally queries the service to get the model status.
7. Once the model status is set to Done, the model is ready for prediction.

Prediction

The prediction task is performed through a call to the cloud service, indicating the Id of the model instance and the data for the prediction. The result of the service operation is a set of values with prediction probabilities, which is stored in Creatio in the *MLPrediction* table.

If there is a prediction in the *MLPrediction* table for a particular entity record, the predicted values for the field are automatically displayed on the edit page.

Machine Learning service scalability

The employment of [Docker](#) and [Kubernetes](#) makes the machine learning service scalable.

Machine Learning service compatibility with Creatio products

The on-site machine learning service is compatible with all Creatio products of version 7.10 and up. The cloud machine learning service is compatible with all Creatio products of version 7.13.3 and up. To set up the service in an older Creatio version, use the docker image of the corresponding version available on [Docker Hub](#).

Machine Learning service deployment options

Using predictive analysis in Creatio on-site requires additional preliminary setup.

To set up the service, use a server (physical or virtual machine) with Linux or Windows OS installed. [Docker](#) software is used for installing the service components. Download the archive containing the configuration files and installation scripts.

We recommend using a Linux-based server for the production environment. You can only use a Windows-based server for the development environment. Contact the support service to receive Docker containers that are compatible with Windows.

See the “[Machine learning service setup](#)” article for more details.

See Also

- [Machine learning service](#)
- [Machine learning service setup](#)

Video tutorial

- [Artificial intelligence and machine learning in Creatio](#)

Phone integration service

Beginner

Easy

Medium

Advanced

Creatio users can integrate their application with multiple [telephone exchanges](#) (telephone switches, [private branch exchanges](#), PBX) to manage phone calls directly from the Creatio interface. Telephony integration functionality is available via the CTI-panel ([Computer Telephony Integration](#)) as well as in the **[Calls]** section.

Introduction

The CTI-panel provides the following features:

- Displaying information about incoming calls and searching contact/account by phone number.
- One-click calls using the Creatio interface.

- Call management in Creatio (answer, place on hold, resume, end, transfer).
- Displaying the call history to enable convenient management of call connections management, redials, and call-backs.

All calls completed or accepted using phone integration are saved in the **[Calls]** section. Use it to view the timing detail of the calls (start time, end time, duration), and the connected Creatio objects. Use the section tools to view detailed information about each call, as well as to build charts and generate analytical reports.

Phone integration service workflow

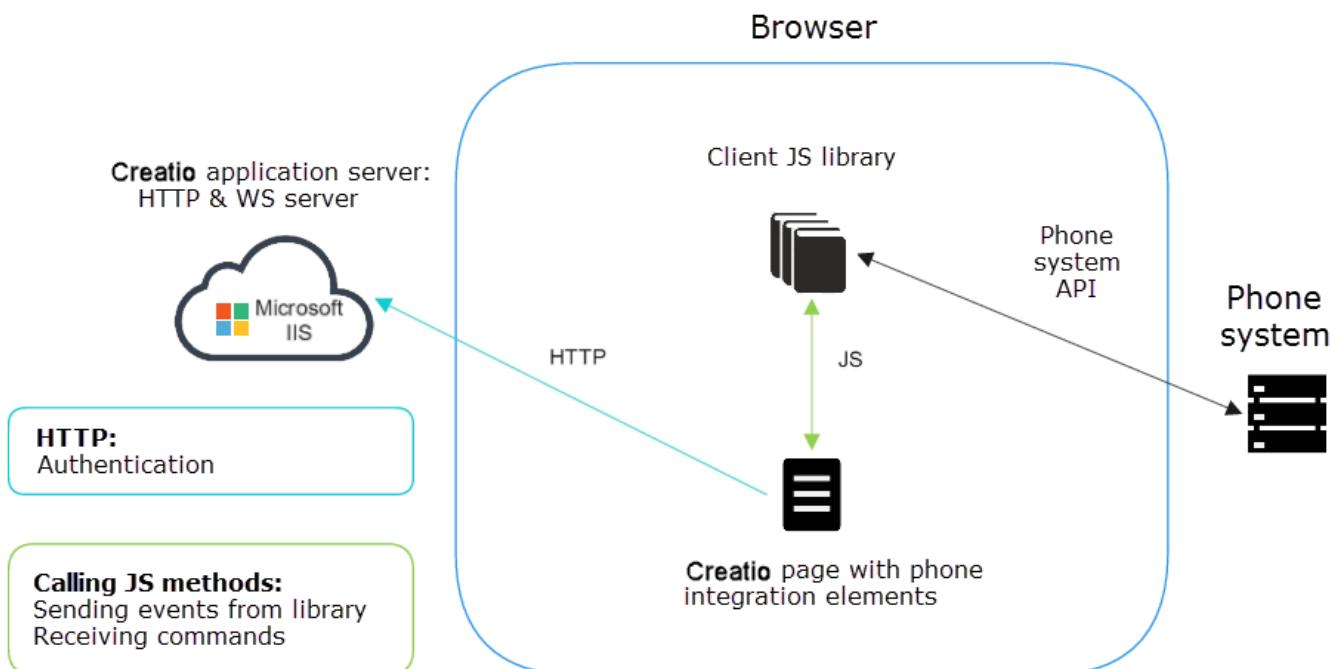
Depending on the specific PBX and the specifications of the available API ([Application Program Interface](#)), different architectural mechanisms (described below) are used. Additionally, the chosen API can affect the set of available features. For instance, not every PBX offers the playback function for recorded calls, and Webitel is currently the only phone integration service that enables the Webphone feature. However, regardless of the chosen integration mechanism, the interface of the CTI-panel is the same for all users.

Integration methods can be divided into two groups: *first-party* and *third-party* integrations.

In the case of *first-party* integration, each user has a dedicated link to the PBX server. The link processes all PBX events of the user. This integration method applies to *first-party* telephony API, for example, the Webitel, Oktell, and Finesse connectors. Webitel and Oktell use [WebSocket](#) as the connection protocol. Finesse uses [long-polling](#) HTTP queries instead.

The advantage of the *first-party* integration is that it does not require additional nodes, such as Messaging Service. The CTI-panel in the user browser connects directly to the API of the telephony server using the integration library (Fig. 1).

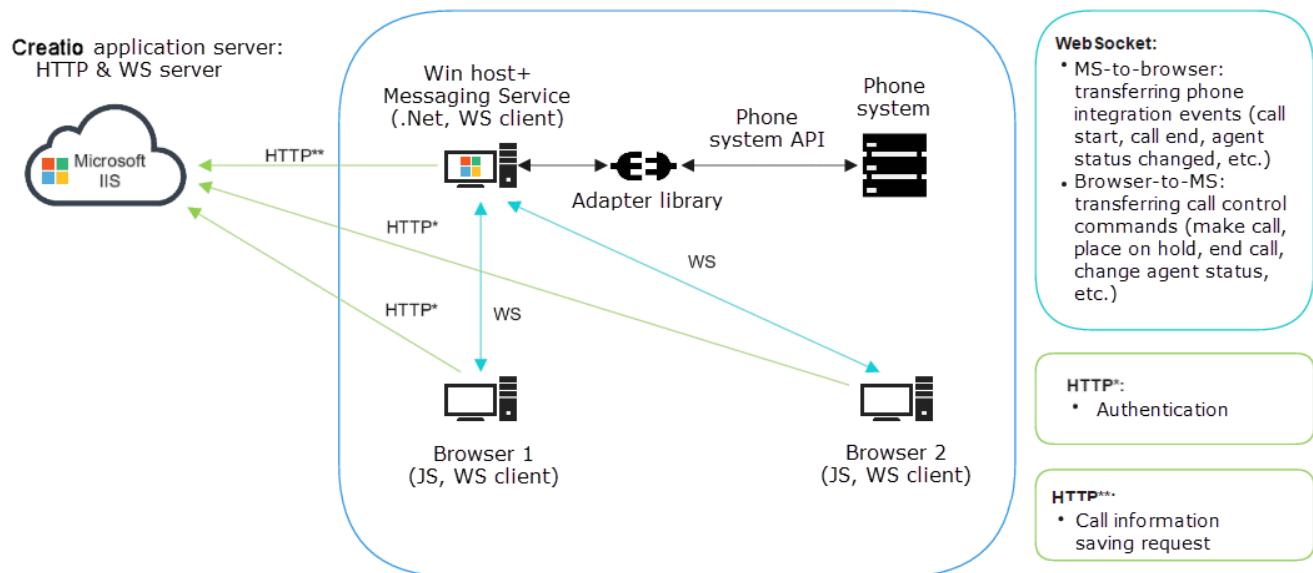
Fig. 1. The workflow of integrating a first-party telephony API using a client-side JavaScript adapter



Third-party integrations feature a single connection to the PBX server. The server processes the PBX event for all integration users. In the case of *third-party* integration, the *Messaging Service* distributes data streams of multiple users.

All telephony events pass through the service when integrating with Terrasoft Messaging Service (TMS) on the server-side (Fig. 2). The service interacts with the PBX using the library by the manufacturer of the PBX. The library interacts with the PBX through the API. The TMS also interacts with the Creatio application server via the HTTP(S) protocol when requesting to store information about the call in the database. The client-side application interacts (sends events and accepts executable commands) with the server via WebSocket. As with the integration using a client-side JavaScript adapter, Creatio pages interact with the application server to authenticate using the HTTP(S) protocol.

Fig. 2. The workflow of a third-party telephony API integration with TMS on the server-side



This integration method can be applied to *third-party* telephony APIs (TAPI, TSAPI, New Infinity protocol, WebSocket Oktell). The integration method requires *Messaging Service* – a Windows proxy service that interfaces with the exchange adapter library. *Messaging Service* is a universal host for PBX integration libraries, such as Asterisk, Avaya, Callway, Ctios, Infinity, Infra, and Tapi. Upon receiving a client connection, this service will automatically connect the library that Creatio uses and initiate a connection to the PBX. The *Messaging Service* is a functional wrapper that enables the telephony functionality in the browser (event generation and processing, data transfer) for phone connectors lacking client integration support. The user computer communicates using two methods:

- Via the HTTP protocol by authenticating to the Creatio application server and accessing the host server running *Messaging Services*.
- Via WebSocket to work with the telephony functionality directly (Fig. 2)

Phone integration service compatibility with Creatio products

The phone integration service is compatible with all Creatio products.

Phone integration service deployment options

Creatio integrates with the Webitel phone integration service. Creatio users can make internal calls just using the headset without any additional software installation. To use other phone integration services in Creatio, additional settings are needed.

Phone integration works in browsers that support WebRTC. The telephony integration service works in the following browsers:

- Google Chrome, the latest official version on the Creatio release date.
- Mozilla Firefox, the latest official version on the Creatio release date.
- Microsoft Internet Explorer 11 and up.
- Microsoft Edge.
- Apple Safari, the last official version on the Creatio release date

The users of Creatio applications deployed in the cloud can call each other by default without using third-party programs.

See Also

- [Phone integration](#)
- [Phone integration](#)
- [Integrations](#)

Sync Engine synchronization service

Beginner

Easy

Medium

Advanced

Creatio implements a mechanism for synchronization with remote repositories (Sync Engine). Sync Engine enables users to create, modify, and delete *Entity* objects in Creatio based on the data imported from remote repositories, as well as export data to remote repositories.

The synchronization process requires the *SyncAgent* class implemented in the *Terrasoft.Sync* namespace on the application core level.

Sync Engine synchronization service workflow

Classes used by the synchronization mechanism

- Synchronization agent (*SyncAgent*) – the class's only public method—*Synchronize*—initiates synchronization between the data repositories passed as arguments.
- Synchronization context (*SyncContext*) – the class is an aggregation of repositories and metadata required by *SyncAgent*.
- Provider – a specific data repository that hosts synced data.
Local storage (*LocalProvider*) – enables working with *LocalItem* in Creatio.
Remote storage (*RemoteProvider*) – the remote service or application whose data sync with Creatio.
- Synchronization item (*SyncItem*) – a set of remote and local objects mapped one-to-one.
Remote synchronization item (*RemoteItem*) – an atomically synced set of data from a remote repository.
The set can comprise one or several entities (records) from a remote repository.
Synchronization entity (*SyncEntity*) – a wrapper for specific *Entity* objects. *SyncEntity* is required to work with *Entity*: as the synchronized object, as the state of the object, or as the action to apply to the *Entity* (add, delete, modify).
Local item (*LocalItem*) – one or several Creatio objects synchronized with a remote repository as a single item. In turn, a remote synchronization item converted into entities of the *LocalItem* type contains a set of *SyncEntity* instances.
- The *SysSyncMetadata* metadata table contains service information on the synchronized items.
Essentially, it is a *RemoteItem*-*LocalItem* bridge table. Learn about metadata in more detail in the “[Synchronizing metadata in Creatio](#)” article.

The general synchronization algorithm

Before initiating synchronization, create a *SyncAgent* instance and a *SyncContext* object, then use Creatio data to update the records in the metadata table. To do this, call the *CollectChangesInSyncedEntities* method of a class that implements the *IReplicaMetadata* interface.

To update metadata records:

1. If a previously synced Creatio entity has been changed since the last synchronization, the modification date of the corresponding metadata record changes. The *LocalState* property is set to “Modified,” and the *LocalStore* identifier is set as the modification source.
2. If a synced Creatio entity has been deleted since the last synchronization, the *LocalState* property of the corresponding metadata record is set to “Deleted.”
3. A Creatio entity that lacks a synchronization metadata record is ignored.

After that, the synchronization process of repositories commences:

1. All changes are requested one by one from remote repositories.
2. The synchronization process requires to retrieve the metadata for each remote item.

The following scenarios are possible:

1. If metadata retrieval fails, the item is considered new. Such an item is converted into a Creatio item and saved. To populate the synchronization item, the application calls the *FillLocalItem* method of a specific *RemoteItem* instance. A metadata record is also saved. The metadata includes the *Id* of the remote storage, the *Id* of the remote item, the creation and modification date (set to current), and the source of creation and modification (set to the remote storage identified).
2. Successful retrieval of the metadata implies that the item has already been synced with Creatio. Move on to version conflict resolution. The latest local or remote changes (implemented in the *RemoteProvider*) have a priority by default.

3. The metadata for the current pair of synced items is updated.

After iterating through all the modified remote items, the metadata (between the previous and the current synchronizations) is bound to contain items changed locally but not remotely.

1. Retrieve elements changed in Creatio between the previous synchronization and the beginning of the current synchronization.
2. The changes are applied in the remote storage.
3. Update the modification date of elements in the metadata (Creatio change source).

After that, add the new Creatio records that have not yet been synced to the remote storage. Add the metadata for the new items as well.

Sync Engine synchronization service compatibility with Creatio products

The Sync Engine synchronization service is compatible with all Creatio products.

Sync Engine synchronization service deployment options

The Sync Engine synchronization service is hardcoded into the core of the application and does not require additional settings.

See Also

- [Sync Engine synchronization mechanism](#)

Exchange Listener synchronization service

Beginner

Easy

Medium

Advanced

The Exchange Listener synchronization service synchronizes Creatio with MS Exchange and IMAP/SMTP mail services using a subscription mechanism.

Exchange Listener synchronization service workflow

The service consists of two required components:

- The Exchange Listener primary module.
- NoSQL Redis DBMS.

Exchange Listener module

The Exchange listener module uses the mailbox credentials and creates a subscription to “new message” events. The open subscription remains in the component memory to ensure fast response time when new emails arrive. When a corresponding event is received, the email instance loads. Using an in-memory repository will be enough for deploying the service.

NoSQL Redis DBMS

Redis DBMS enables creating a scalable and fault-tolerant system of processing nodes. The Redis repository contains information about the mailboxes that are served. This lets any container process Creatio queries for adding a new subscription or check the status of a specific subscription regardless of the subscription node.

Requirements to Redis:

- Anonymous access allowed.
- Separate database available for the Exchange Listener service operation.

Exchange Listener synchronization service scalability

By default, separate nodes of the StatefulSet type process the requests based on 1 processor replica per 50 active mailboxes. The number of replicas depends on the *replicaCount* parameter. You can increase the number of processors by specifying the needed value. You can configure automatic scaling depending on the number of active subscriptions. To learn more, contact the Creatio technical support service.

Exchange Listener synchronization service compatibility with Creatio products

The Exchange Listener synchronization service version 1.0 (MS Exchange support) is compatible with all Creatio products of version 7.15.2 and up.

The Exchange Listener synchronization service version 2.0 (IMAP/SMTP support) is compatible with all Creatio products of version 7.16 and up.

Exchange Listener synchronization service deployment options

We recommend using the Kubernetes orchestrator and Helm package manager to deploy the service and ensure the operation of the application in the production environment. Learn more about deploying the synchronization service via Kubernetes in the "[Deploying the synchronization service via Kubernetes](#)" article.

You can also use Docker to speed up the deployment in the development environment. Learn more about deploying a containerized version of the synchronization service in the "[Deploying the synchronization service in Docker](#)" article.

See Also

- [Exchange Listener synchronization service](#)
- [Server-side system requirements](#)

Static content bundling service

Beginner

Easy

Medium

Advanced

All custom content (e.g., the source code of custom schemas, CSS-styles) is generated in a special Creatio directory to improve performance. The benefits of this approach are described in a separate article - "**Client static content in the file system**." However, having to process a considerable amount of files, the browser sends a large number of requests while loading the application, which significantly increases the loading time. To ensure stability, similar files are collected into bundles. For this particular purpose, the static content bundling service is implemented in Creatio by default.

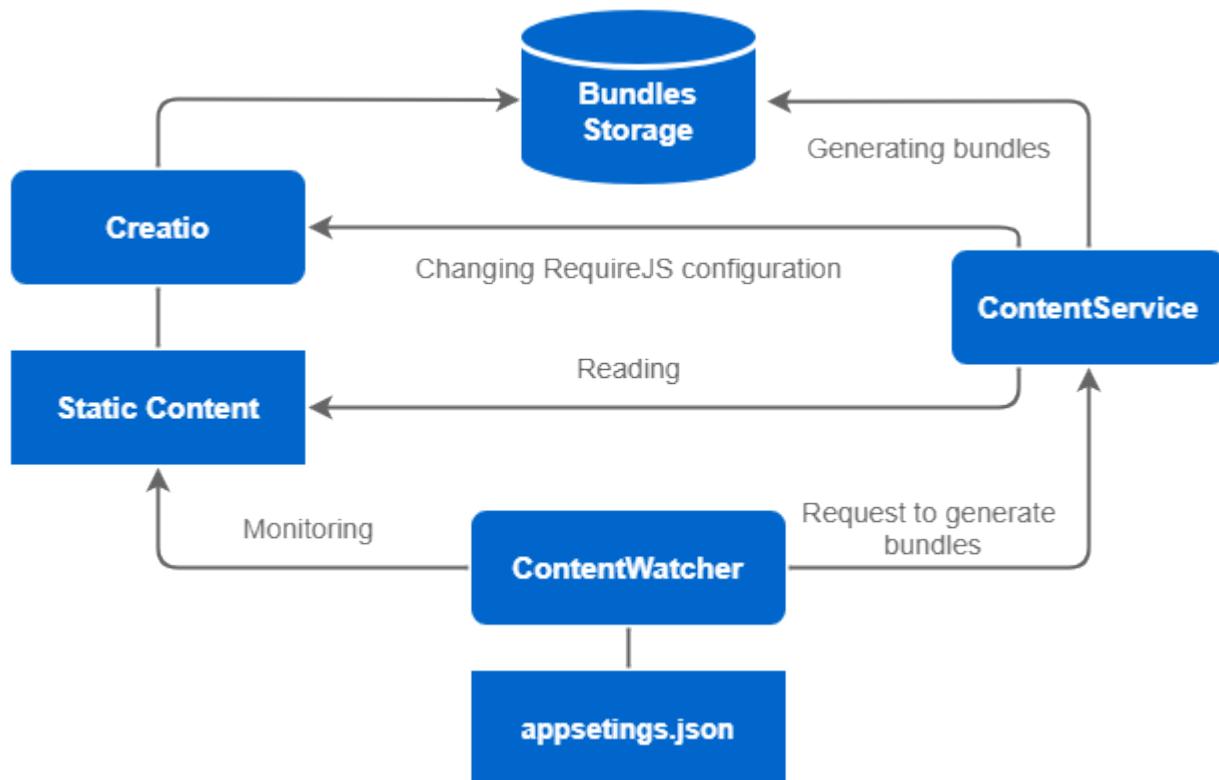
Static content bundling service workflow

The "watcher" app (*ContentWatcher*) monitors the files in a static content directory and notifies the web-service about any changes.

Upon request (either manual or from *ContentWatcher*), the web-service (*ContentService*) re-generates bundle-files and modifies a specific Creatio configuration file so that it uses bundles instead of static content.

Fig. 1 illustrates the basic work principles of the static content bundling service.

Fig. 1. Basic work principles of the static content bundling service



A web-service can be installed without the “watcher” app (*ContentWatcher*). In this case, all requests to the web-service (*ContentService*) for bundling or [minification](#) can only be done manually.

The service components can be installed on the same machine as the Creatio application service or on a separate machine. If the application server and the components of the bundling service are not installed on the same machine, the service components must be able to access Creatio static content files.

ContentService

ContentService is a .NET Core 2.1 web-service which performs the following operations (access points):

- `/` – tests the service efficiency (the GET method).
- `/process-content` – generates minified bundle-files (the POST method).
- `/clear-bundles` – clears bundle-files (the POST method).
- `/minify-content` – minifies content (the POST method).

ContentWatcher

ContentWatcher is .NET Core 2.1 application. ContentWatcher is run as a service (it can also be run using .NET Core CLI Tools). The primary task of ContentWatcher is to monitor any changes in a file specified in the `fileFilter` parameter. The path to the file itself is specified in the `directory` parameter (e.g., `--directory='c:\temp' --fileFilter='readme.txt'`). By default, the `fileFilter` parameter value is `"_MetaInfo.json."` If this file is changed, ContentWatcher considers this to be an update of all static content. When changes are detected, ContentWatcher notifies ContentService to re-generate bundle-files.

Service configuration structure:

- `ets/content-watcher/appsettings.json` - ContentWatcher configuration file.
- `Docker-compose.yml` - docker-compose utility configuration file.
- `.env` – the file containing environment variables for running the components.

Static content bundling service compatibility with Creatio products

The static content bundling service is compatible with all Creatio products of version 7.11 and up.

Static content bundling service deployment options

You can use the static content bundling service both on-site and in the cloud. Use a Docker container to deploy the service on-site. Contact the Creatio technical support to deploy the bundling service in the cloud.

System requirements:

- A Linux OS server (stable versions of Ubuntu or Debian are recommended) with a stable version of Docker installed and configured. The requests to the image repository ([Docker Hub](#)) must be allowed from this server.
- Both Docker and Docker Compose must be installed on the server (see the [Docker documentation](#)).

See Also

- **Static content bundling service**

Bulk email service

Beginner

Easy

Medium

Advanced

The bulk email service is designed for integrating Creatio with bulk email providers. Bulk emails are one of the most effective marketing tools for promoting products and services.

Introduction

In Creatio, emails are managed in the **[Email]** section, which you can use to:

- Set up email templates.
- Segment email recipients.
- Access delivery analytics.
- Access individual bulk-email feedback.

The following marketing email functions are available in Creatio:

- **Bulk emails.**
Sent once to a set number of recipients. Bulk emails enable you to actively engage your customers. Learn more about bulk emails in the [“Bulk emails”](#) block of articles.
- **Trigger emails.**
Trigger emails are sent automatically to each recipient who triggers them (e.g., submits a web form, clicks a link, etc.). Learn more about trigger emails in the [“Trigger emails”](#) block of articles.

Access to the marketing email functionality is licensed separately. Learn more in the [“Creatio licensing”](#) article.

Set up your email service integration with Creatio for using the bulk email service. All cloud email service settings for bulk emails are consolidated on the bulk email setup page in the **[Email]** section. Learn more about working with the **[Email]** section in the [“The \[Email\] section”](#) block of articles.

Email domain verification is required before using the email functionality. Learn more about setting up an email domain in the [“Email domain verification”](#) article. Two mail services are available for sending bulk emails from Creatio: UniOne and Elastic Email. By default, Creatio is integrated with UniOne. The UniOne email provider is covered in the [“Domain verification for the UniOne provider”](#) article. Contact Creatio support to send emails via Elastic Email. The Elastic Email provider is covered in the [“Domain verification for the Elastic Email provider”](#) article.

Set up the bulk email contents that the recipients will see (the email template) before sending bulk emails. There are two types of marketing email templates in Creatio: templates that display the same content for all recipients (static content), and templates, whose content differs for different target audiences (dynamic content). Email templates are created via a no-code visual drag&drop editor called “Content Designer.” Learn more about working with the Content Designer in the [“Marketing Content Designer”](#) article. Learn more about setting up an email template in the [“Marketing email templates”](#) block of articles.

You can use email analysis to see the email results and evaluate their effectiveness. The **[Email]** section analytics provides detailed statistics both for individual marketing emails and for aggregated metrics. Learn more about analyzing bulk email results in the [“Email analysis”](#) block of articles.

Bulk email service compatibility with Creatio products

The bulk email service function is available in Marketing Creatio.

Bulk email service deployment options

You can deploy the bulk email service on-site and in the cloud. Learn more about setting up bulk emails in the “[Set up bulk emails](#)” article

See Also

- [Set up bulk emails](#)
- [The \[Email\] section](#)

Video tutorial

- [Email marketing](#)

Developing solutions in Creatio

Contents

- **Introduction**
- **Low-code and no-code development**
- **Front-end (JS)**
- **Back-end (C#)**
- **Integrations**

Introduction

Beginner

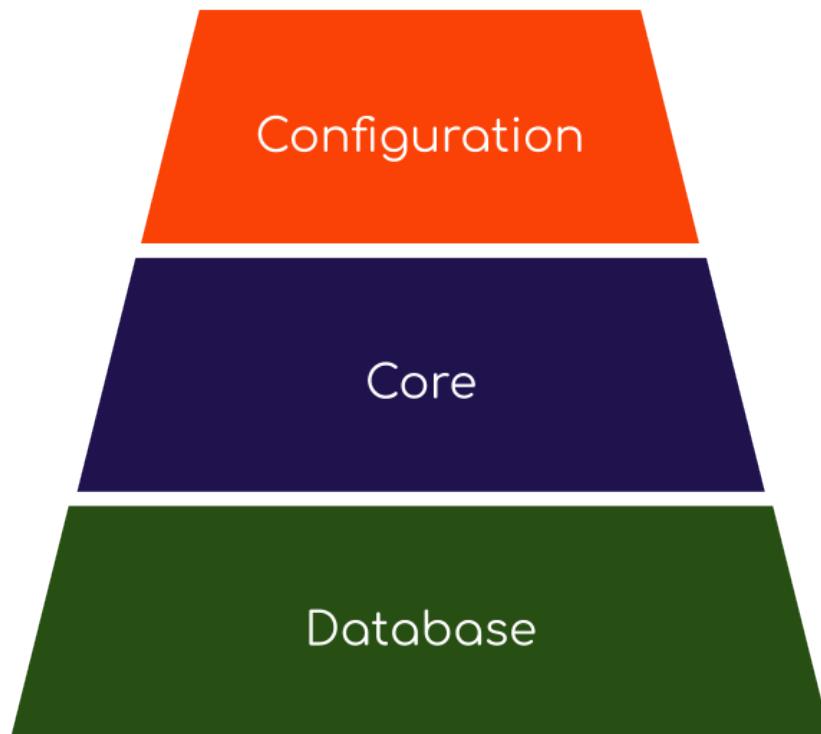
Easy

Medium

Advanced

As far as the components are considered, Creatio can be represented as follows (Fig. 1):

Fig. 1. Components



The development of solutions in Creatio involves different customization levels depending on the type or complexity of a business task. Note that the application core level is an immutable system component. Development in Creatio is done on the configuration and database levels.

Database

This is the level of physical data storage. The database stores not only the client data but also the application settings, as well as the settings of access permissions.

As a rule, development on the Creatio platform does not require working with database objects directly. Creatio tools enable users to work with data using the native interface.

That said, using views and stored procedures on the database level allow users to implement custom business logic and call it from custom configuration elements.

For certain tasks, development on the database level is the most logical solution and the fastest method.

Core

The core is an immutable part of Creatio. It is a set of libraries implementing the base application functionality.

Back-end libraries are implemented in C# with using the .NET Framework platform classes. Developers can create back-end class instances and use the functionality of back-end libraries, but they cannot make any changes in these classes and libraries.

The main back-end components:

- ORM - data model and methods of working with it. In most cases, we recommend using the object model, though the direct access to the database is also implemented in the back-end core components.
- **Packages and replacement mechanism.**
- Libraries of the server control elements. These control elements include pages server-generated using the ASP.NET technology, for example, pages of the **[Configuration]** section.
- System web services.
- The functionality of the main designers and Creatio sections.
- Libraries for integration with third-party services.
- Business process engine (ProcessEngine). This is an important Creatio element that can implement algorithms set up as process diagrams.

The primary task of the front-end components of the core is providing the operation of client modules. The front-end classes of the core are implemented in JavaScript using different frameworks. They are designed for creating user interfaces and implementing other business tasks on the side of the browser.

The main front-end components:

- External libraries of client frameworks.
- **Sandbox** – a special client core component, designed for providing the interaction between different client modules by exchanging the messages.
- **Base modules** – the files written in JavaScript, where the functionality of the primary Creatio objects is implemented.

Configuration

Configuration – is a set of functionality available for Creatio users of a specific workspace, namely:

- Server logic.
- Auto-generated classes that are the product of system setting operation.
- Client logic – pages, buttons, actions, reports, business processes, and other configuration elements.

The configuration is an easy to modify part of the system. A specific configuration is formed by the following element types:

- Objects – entities designed for data storage that combine a table in the DB and a class on the server-side.
- Business processes – the configured elements that represent a visual algorithm of user actions.
- Client modules.

In Creatio, all the configuration elements are grouped in **packages**.

A package is a final set of functionality that can be added to the configuration and can be deleted.

The final Creatio functionality is formed by the set of packages in the configuration.

The package mechanism in Creatio is built on the open-closed principle (one of the SOLID OOP practices).

According to this approach, all entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that new logic and features must be implemented by introducing new entities rather than modifying the old ones.

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

The Creatio packages can be divided into two types:

- Pre-installed packages. These include packages with basic functionality (e.g., *Base*, *NUT*), packages that extend the functionality of the system (for example, 1C integration packages, telephony, etc.), and packages developed by third-party developers. Such packages are either delivered along with the primary application or installed from zip-archives **as Marketplace applications**.
- **Custom packages** – created by the system users. These packages can be attached to the SVN repository.

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

The existing functionality is modified using the replacement mechanism. To change the behavior of a pre-installed element, create a new custom package element that extends it. For the created object, set a property implying that the object will replace the parent object in the object hierarchy. All modifications that have to be applied to the pre-installed object are to be implemented in the replacement object. When calling a preinstalled object, Creatio will then run the logic of the corresponding replacement object.

A single base object can be replaced in multiple custom packages. The resulting replacing object implementation in the compiled configuration is based on the hierarchy of packages that contain the replacing objects.

To use the functionality of a package in another package, specify a dependency on the source package.

The dependent package extends or changes the functionality of the source package. Thus, Creatio builds a hierarchy of dependencies and packages. Low-level packages can extend or modify the logic and features of any package higher in the dependency tree. More information about the package hierarchy and dependencies is available in the **“Package dependencies” article**.

A complete list of all packages installed in the workspace is available on the [Packages] tab of the [Configuration] section.

Package structure:

1. Schemas – configuration items that define new or existing configuration elements.
2. External assemblies – third-party libraries required for development and integration with third-party systems They can be used in source code schemas after the installation.
3. SQL scripts – arbitrary SQL scripts run upon package installation. You can use SQL scripts to transfer a package to other configurations if it is associated with database changes.
4. Data – section records, lookup values, and system settings that have been developed in the package. You may require package data to transfer the package to other development environments if the package is associated with specific records and database values.

More information about working with packages is available in the **“Working with packages” article**.

Configurable functionality:

Use Creatio development tools to customize the application for solving specific business problems of the user:

1. Add new or modify existing system objects, as well as sections, pages, and application details.
2. Create or edit business-processes.
3. Implement the required business logic for the application components.
4. Implement solutions integrating external applications.

You can use the following tools to develop solutions in Creatio:

1. Low-code and no-code tools (section wizard, content designer, process designer).
2. Front-end development using the built-in IDE or a third-party IDEs using the functionality of core front-end

components (changing the business logic of interface components, creating custom components).

3. Back-end development using the built-in IDE or a third-party IDEs using the functionality of core back-end components (data processing, implementing integration solutions, user's business process logic).

See Also

- **Low-code and no-code development**
- **Front-end (JS)**
- **Back-end (C#)**
- **Integrations**

Low-code and no-code development

Beginner

Easy

Medium

Advanced

Low-code and no-code technologies use visual interfaces to enable users to develop their own IT solutions without in-depth knowledge of any programming language.

No-code technologies cater to [citizen developers](#) since they do not require knowledge of the programming language syntaxes. Organizations choose platforms that combine both technologies for more flexibility and control over the software development life cycle.

Advantages of low-code platforms:

- Fast application development.
- Fast deployment.
- Application execution and management.
- Declarative high-level development.
- Ability to model data, develop interfaces, and business logic.
- User (no-code) application configuration.

Low-code tools

The Creatio platform has a wide range of low-code/no-code customizations: from customizing the existing applications to creating the user's business solutions.

[Drag&drop](#) visual designers enable the users of low-code platforms to set up applications for solving a multitude of business tasks: from automating customer processes and enhancing teamwork to data management and third-party integration.

Process designer

The process designer is a visual designer for building “executable” business processes with BPMN 2.0 notation. Executable business processes enable the implementation of custom business logic, from automating routine tasks to creating complex iterations. Employ ready-to-use elements to design business processes that enable users to plan their activities, work with interface pages, process data, call web services, and more. The user-friendly setup interface and built-in validation tools will help not just to design or update a business process scheme, but also to debug the scheme accounting for all execution details and nuances. Use the process designer to automate and describe business processes of varying complexity for better performance.

The principles of working with business processes in Creatio are provided in the [business process guide](#).

Case designer

Case management enables users to automate unstructured processes with a dynamic flow in line with the established business logic. A business case consists of stages. Each stage may include a sequence of consecutive or simultaneous “steps,” i.e. automatic or manual actions. You can use the no-code case designer to change the sequence of steps in a process stage, move steps to other stages, or rearrange the sequence of stages using the [drag&drop](#) designer.

Learn more about working with business cases in the [Case designer guide](#) block of articles.

Section wizard

Create and set up sections, pages, and mini pages to add or edit section records fast. Use the section wizard to add new sections and edit existing ones. Use the visual designer to change the positions of fields, add or hide fields, tabs, and details. Employ the user interface to change the business logic of the system. For example, set up a conditional view or change which section fields are required or available.

Learn more about the section wizard in the “[Section wizard](#)” block of articles.

Machine Learning technology

The machine learning technology enables decision-making automation by analyzing historical data and recognizing correlations between large amounts of data. For example, you can use Creatio smart technologies to set up customer profile categorization, route help desk calls, predict the probability of closing a sale, or recommend products to your customers. Use the machine learning algorithms and other AI technologies to accelerate data processing, reduce the number of manual operations, and improve the quality of decision-making.

Learn more about working with the machine learning service in the “[Machine learning service setup](#)” block of articles.

Built-in integration tools

The advanced integration potential (based on .Net, REST, SOAP, OData, open API, and other tools), as well as a powerful administration and access control system, accelerate the safe integration of Creatio into the digital ecosystem of any enterprise. The Creatio platform also enables unlimited third-party integration flexibility.

Integration options using low-code technologies are covered in the “[Integrations](#)” article.

Front-end (JS)

Beginner

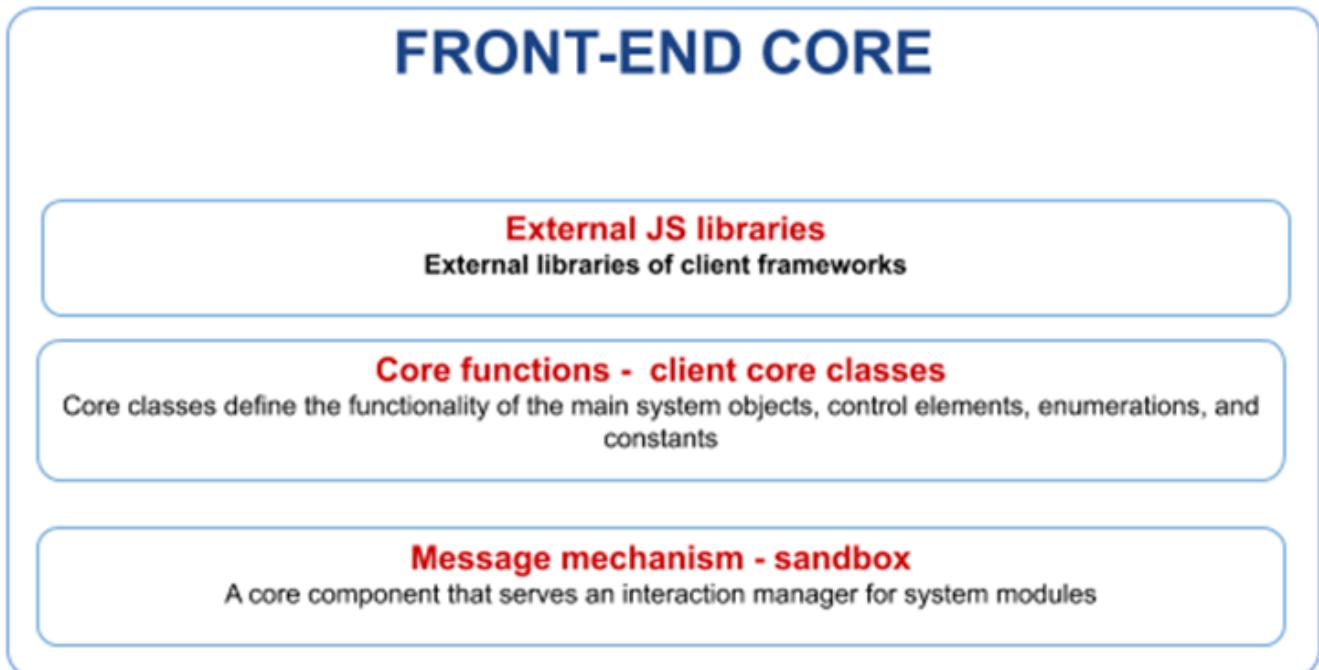
Easy

Medium

Advanced

Base JS classes and external libraries

Fig. 1. The scheme of the front-end core



External libraries

ExtJS – is a JavaScript framework for developing web apps and UIs. The primary purpose of this framework is the creation of complex and feature-rich interfaces. Creatio uses ExtJS as a mechanism for generating the class structure

of the client part of the core. The framework makes it possible to implement the object-oriented approach, which is not fully supported in pure JavaScript. ExtJS enables developers to create classes, implement an inheritance hierarchy, and group classes in namespaces.

RequireJS – is a library designed for the implementation of the [Asynchronous Module Definition \(AMD\)](#) approach. The AMD approach essentially declares the mechanism that defines modules and dependencies and loads them asynchronously.

Angular – is designed for the development of one-page apps. The primary purpose of Angular is the extension of browser apps using the MVC pattern, as well as easier development and testing. You can embed custom Angular components in Creatio using a single Angular core.

Base JS classes

Base JS classes are an immutable part of Creatio. The primary purpose of the base classes is to ensure the operation of client configuration modules. Base classes are stored as executable files in the application folders. Base classes define the functionality of the main system objects, control elements, enumerations, and constants.

Message mechanism

Sandbox – is a core component that serves an interaction manager for system modules. Sandbox provides the module message exchange mechanism (the `sandbox.publish()` and `sandbox.subscribe()` methods) and the mechanism for loading modules into the application's UI on-demand (the `sandbox.load()` method).

Asynchronous module definition

The client part of the application has a modular structure.

A module – is a set of functionality encapsulated in a separate block, which can use other modules as dependencies.

Module creation in JavaScript is codified in the “[Module](#)” design pattern. A classic way to implement this pattern is the use of a lambda function that returns a specific value (object, function, etc.) that is associated with the module. The value of the module is exported to the global object.

The main drawback of this approach is the complexity in declaring and using dependencies in such modules:

1. All module dependencies must be loaded when the lambda function is executed.
2. Module dependencies are loaded in the `<script></script>` HTML container in the page header. They are then accessed using the global variable names. A developer must have a clear idea of the loading order of all dependency modules and be able to implement it.
3. Due to the position of the HTML container, the modules are loaded before the page is rendered. This prevents the modules from accessing page controls for implementing any custom logic.

Effectively, this drawback implies the inability to load modules dynamically, apply any custom logic when loading modules, etc. Large projects, such as Creatio, have one more problem arising from the sheer complexity. A large number of modules makes managing them rather difficult due to the many dependencies that may overlap.

To address the shortcomings mentioned above, Creatio loads modules and their dependencies using the [Asynchronous Module Definition](#) (AMD) approach.

The AMD approach declares the mechanism of defining and asynchronous loading of modules and their dependencies. When working with the system, this method enables loading only the data currently required for operation. The AMD concept is supported by different JavaScript-frameworks. In Creatio, we use the [RequireJS](#) loader for working with modules.

The main operating principles of the RequireJS loader:

1. A module is declared in the `define()` function, which registers a function factory for instantiating the module but does not load it immediately upon a call.
2. The module dependencies are passed as an array of string values, they are not passed through the global object properties.
3. The loader performs loading of all dependency modules passed as arguments to `define()`. The module loader works asynchronously and indiscriminately.
4. After the module loader loads all module dependencies, the function factory is called to return a module instance. The loaded dependency modules will be passed to the library-function as arguments.

To ensure loader interaction with the asynchronous module, the module should be declared in the source code

through the `define()` function as follows:

```
define(
  ModuleName,
  [dependencies],
  function (dependencies) {
  }
);
```

Operation parameters of the `define()` function are listed in table 1.

Table 1. The parameters of the `define()` function

Argument	Value
ModuleName	A string with the module name Optional parameter. If not specified, the loader will assign the module a name depending on the module's position in the scripts tree of the app. However, to make the module callable from other parts of the application (including for asynchronous loading as another module's dependency), the name of the module must be specified explicitly.
dependencies	An array with the names of dependency modules. Optional parameter. RequireJS will load all dependencies passed in the array. Note that the order of dependency enumeration in the <code>dependencies</code> array must match the order of parameter enumeration passed to the function factory. The function factory will only be called after all of the items in the <code>dependencies</code> array are loaded. Dependency modules are loaded asynchronously.
function(dependencies)	A lambda function factory that is instantiated by the module. Required parameter. The function accepts as arguments the objects that the loader associates with the dependency modules enumerated in the <code>dependencies</code> array. These arguments are required to access the properties and methods of the dependency modules in the created module. The order of module enumeration in the <code>dependencies</code> array must match the order of arguments passed to the function factory. The function factory will only be called after all of the dependency modules of the current module (enumerated in the <code>dependencies</code> parameter) are loaded. The function factory must return a value that the loader will associate as the exported value of the created module. The returned value can be: <ul style="list-style-type: none"> • <i>A system module instance.</i> After the initial download to the client device, the module is stored in the browser cache. Clear the cache and reload the module to display any updates if the module declaration has been changed after the download (for example, due to the implementation of the configuration logic). An example of declaring and instantiating a module is available below. • <i>Module function factory.</i> Pass an instance of the context (where the module will be created) to the constructor as an argument. Loading such a module will generate an instance of the module in the client application. A repeated download of the module using the <code>require()</code> function will generate another module instance. The system will treat the two instances of the same module as two separate independent modules. An example of an instantiable module declaration is the <code>CardModule</code> module of the <code>NUI</code> package.

Module development in Creatio

System functions are implemented via client modules.

Fig. 2. Hierarchy of modules in Creatio

Module

custom code item
(for example, **ConfigurationConstantsV2**)

Non-visual module

system functionality that is not associated with data binding or data display in the UI
(for example, **BusinessRuleModule**)

Visual module

view model module (**ViewModel MVVM** pattern):
responsible for data binding, data display in the UI, and the methods for working with that data

View model schema

used for the creation and configuration of the main UI elements - sections, pages, and details
(for example, **ContactPageV2**)

The majority of Creatio customization tasks are implemented on this level

All client modules in Creatio share description structures that correspond with the AMD module description format.

```
define(  
    "ModuleName",  
    ["dependencies"],  
    function(dependencies) {  
        // someMethods...  
        return { moduleObject };  
    } );
```

In this code:

- ModuleName – module name
- Dependencies – dependency modules whose functionality can be used in the current module
- moduleObject – a configuration object of the created module.

The following client module types are available in Creatio:

- non-visual module
- visual module

Non-visual module

Non-visual modules represent system functionality that is not associated with data binding or data display in the UI.

The description structure of a non-visual module:

```
define(  
    "ModuleName",  
    ["dependencies"],  
    function(dependencies) {  
        // Methods that implement the required business logic  
        return;  
    } );
```

Examples of non-visual modules in Creatio are utility modules that implement service functions.

Visual module

A visual module (view model) is used to create UI elements.

Visual modules are used to implement ViewModel presentation models in Creatio according to the MVVM pattern. Visual modules encapsulate both the data used in the GUI controls and methods for working with that data.

A visual module must contain the following methods:

- *init()* – the module initialization method.
Responsible for initializing the properties of a class instance and for subscribing to messages.
- *render(renderTo)* – the method for rendering a module view in the DOM.
Must return a module view.
Only accepts the *renderTo* argument – the element to insert the module view.
- *destroy()* – this method is responsible for the deletion of the module view, for the deletion of the module view model, for the unsubscription from the message to which a subscription has been made, and for the destruction of the module class instance.

You can use the base classes of the core to create a visual module. For example, create a module class inheriting from *Terrasoft.configuration.BaseModule* or *Terrasoft.configuration.BaseSchemaModule* in the module. These classes of the client core already implement the required visual module methods – *init()*, *render(renderTo)*, and *destroy()*.

The source code structure of the visual module description (inherited from the *Terrasoft.BaseModule* base class):

```
define("ModuleName", ["dependencies"], function(dependencies) {
    // Class module definition
    Ext.define("Terrasoft..configuration.className") {
        alternateClassName: "Terrasoft.className"
        extend: "Terrasoft.BaseModule",
        ...
        // properties and methods
        ...
    };
    // Creating module object
    return Ext.create(Terrasoft.className)
});
```

Examples of visual modules are modules that implement the control element functionality on the application main page.

View model schema

The most frequent application customization tasks include the creation and modification of the main UI elements – sections, pages, and details.

The modules of these elements have a patterned structure and are known as view model schemas.

A view model schema is a configuration object for generating a view and view model by the *ViewGenerator* and *ViewModelGenerator* Creatio generators.

The majority of Creatio customization tasks require the use of the replacement mechanism for substituting base schemas. All schemas from the pre-installed configuration packages are considered base schemas.

The primary base schemas of view models:

- *BasePageV2*
- *BaseSectionV2*
- *BaseDetailV2*

All the schemas have a common source code structure:

```
define("SchemaName", ["dependencies"],
    function(dependencies) {
        return {
            entitySchemaName: "ExampleEntity",
            mixins: {},
            attributes: {}
        }
    }
});
```

```
        messages: {},
        methods: {},
        rules: {},
        businessRules: {},
        modules: {},
        diff: []
    );
}) ;
```

Examples of view schemas include the schemas of pages, sections, and details.

Video tutorial

- [Development on Creatio platform](#)

Back-end (C#)

Beginner

Easy

Medium

Advanced

The development of solutions in Creatio involves different customization levels depending on the type or complexity of a business task. Note that the application core level is an immutable system component. Development in Creatio is done on the configuration level.

Developers can create back-end class instances and use the functionality of back-end libraries implemented on the application core level, but they cannot make any changes in these classes and libraries.

Back-end development areas

Back-end application customization involves:

- the ORM data model and methods of working with it. In most cases, we recommend using the object model, though the direct access to the database is also implemented in the back-end core components
- implementing direct access to the database
- creating and using of the system web service
- setting up integration with external services (for example, the SyncEngine service)
- working with the system components and additional services (for example, the machine learning service)
- extended setup of business processes and built-in processes of the application objects
- developing object business logic.

ORM data model and direct access to the database

The ORM model is implemented in the *Terrasoft.Core.Entities* namespace classes. The *Terrasoft.Core.Entities.EntitySchemaQuery* class is used to build queries for selecting records in Creatio database tables. The *Terrasoft.Core.Entities.Entity* class is designed to provide access to an object that represents a record in the database table

Direct database access is provided by a group of server core classes from the *Terrasoft.Core.DB* namespace. Use this class group to perform all basic CRUD operations, account for the access permissions of the current user, and put the retrieved data to the cache storage. Learn about classes in more detail in the “**Working with database**” article.

If fetching the needed data requires a complex database query (for example, several nested filters, various combinations of JOIN commands, etc.), you can query the database directly using the *Terrasoft.Core.DB.DBExecutor* class.

System web services

Creatio service model implements the base set of web services, which you can use for integration with third-party applications and systems. Examples of system services:

- **odata** – data exchange with Creatio via the [OData4](#) protocol.
- **ProcessEngineService.svc** – running Creatio business processes using third-party applications.

Creatio also provides configuration web services designed for calls from the client part of the application.

The developers can create a custom web service designed for solving a project-specific task.

Integration with external services

The core of the application implements classes that provide third-party integration features. For example, the *Terrasoft.Social* namespace enables integration with social networks.

System components and services

The *Terrasoft.Sync* namespace provides the classes of the built-in mechanism for synchronization with remote repositories (Sync Engine). Sync Engine enables users to create, modify, and delete [Entity](#) objects in Creatio based on the data imported from remote repositories, as well as export data to remote repositories. The synchronization process relies on the *SyncAgent* class. Learn more about the synchronization mechanism in the “[Creatio synchronization with external storages](#)” article.

Set up business processes and built-in processes of the application objects

You can use back-end development to create complex business processes (the **[Script-task]** process element) or recurrent user's business process operations (the **[User Task]** configuration schema).

You can configure the built-in processes of an object using the no-code tools as well as back-end development. Using the program code allows configuring a more flexible object behavior accounting for specific events.

Develop the business logic of objects.

Creatio supports the development of the business logic of an object without using event sub-processes. To do this, make a class that inherits the *Terrasoft.Core.Entities.Events BaseEntityEventListener* base core class, decorate it using the *EntityEventListener* attribute, and specify the name of the entity whose event subscription must be executed. Then, you can reload the handler methods of the required events. < Learn more about this in the “[Entity event layer](#)” article.

Tools and utility capabilities for back-end development

Source code schema

The main back-end development method is the creation of **[Source code] schemas** in the custom package.

All changes to the schema made in the object designer are kept in memory. To save the changes on the schema metadata level, save the schema. To do this, click **[Save]** in the object designer. To apply the changes to the database, publish the schema.

You can develop **[Source code]** schemas in the file system development mode using an IDE of your choice. Learn more about the file system development mode in the “[Working with the server side source code](#)” article.

Configure business processes

You can enable a specific back-end logic in a business process by introducing the source code in a **[Script task]** element.

It is often necessary to perform similar operations repeatedly while working with business processes in Creatio. The **[User task]** process element is best suited for these operations. Specify the most suitable schema title in the **[Which user task to perform?]** property of the process element. Learn more about the **[User task]** element in the “[\[User task\] process element](#)” article.

By default, some user tasks are already available in the system. You can add new user tasks if needed.

The “User task” configuration schema type is used to create new user tasks. The process task partially replicates the logic of the **[Script task]** process element. However, a user task can be reused in different processes. Any changes to the task will be immediately applied to all processes that contain the mentioned task.

External libraries

The structure of a custom package may include external libraries created by the user. This approach supports implementing complex business logic, inheritance mechanism, and encapsulation for developing a project-specific solution.

Project package

The project package model is one of Creatio's standards for accelerating the development of the server schemas for the application. A project package is a package that enables developing new functionality like a regular C# project. The new functionality is included in the **file content package** as a compiled dynamic library and a collection of CS files. When starting or restarting, the Creatio application will collect information about pre-set libraries and promptly include them. Learn more about creating a project package in the “[Developing the source code in the file content \(project package\)](#)” article.

Video tutorial

- [Development on Creatio platform](#)

Integrations

Beginner

Easy

Medium

Advanced

Software system and product integration is data exchange between the systems, with or without subsequent data processing. The point of integration is to transfer the user's data input from one system to another automatically.

Creatio's open API supports creating integration solutions of any complexity.

Integration of external applications with Creatio

Creatio has a wide range of integrations with custom third-party applications. The choice of the integration method depends on the needs of the customer, the type and architecture of the external application, and the expertise level of the developer.

You need to **authenticate** your requests to Creatio.

Integration with Creatio entails achieving the following objectives:

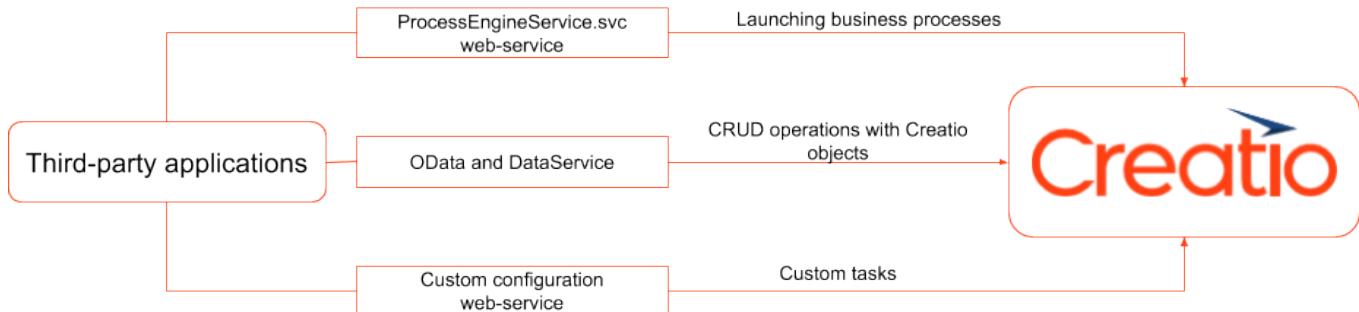
- CRUD operations with Creatio objects.
- Launching business processes.
- User tasks can be solved within the open Creatio API.

Creatio provides the following integration options for CRUD operations:

- The OData protocol.
- The DataService web service, developed by Creatio.

The Creatio integration scheme for external applications is presented in Figure 1.

Fig. 1. Integration of external applications with Creatio



The OData service

OData (Open Data Protocol) is an ISO/IEC-approved OASIS standard. It defines a set of best practices for building and consuming REST APIs. It enables creating REST-based services that allow web clients to publish and edit resources using simple HTTP requests. Such resources should be identified with a URL and defined in the data model.

Learn about the protocol in more detail in the [OData documentation](#).

Creatio supports OData 3 and OData 4 protocols. OData 4 replaces OData 3 and enhances the capacity of the predecessor protocol considerably. The protocols are not compatible concerning the format of data returned by the servers.

To integrate with Creatio, use OData 4.

Learn more about Creatio integration using OData version 3 and 4 in the “**Integration via OData protocol**” block of articles.

The DataService web service

The DataService web service is the main link between the Creatio client and server parts. It helps to transfer the data entered in the UI by the user to the server-side of the application for further processing and saving to the database.

Learn more about Creatio integration via DataService and the major operations supported by the service in the “**DataService**” article.

Business process launcher service (ProcessEngineService)

Running the business processes is one of the purposes of integrating an external application with Creatio. The Creatio service model implements the *ProcessEngineService.svc* web service for launching business processes from an external application.

Learn more about Creatio integration using the web service in the “**The ProcessEngineService.svc web service**” article.

Custom configuration web service

Creatio enables to create custom web services in the configuration that can implement specific integration tasks. The configuration web service is a RESTful service based on WCF technology.

Learn more about creating a custom configuration web service in the “**Creating a user configuration service**” article.

Creatio integration with third-party applications

Use the low-code/no-code integration tools to combine different corporate applications into a single digital ecosystem. Learn about the low-code and no-code tools in more detail in the “**Low-code and no-code development**” article.

Develop integration solutions

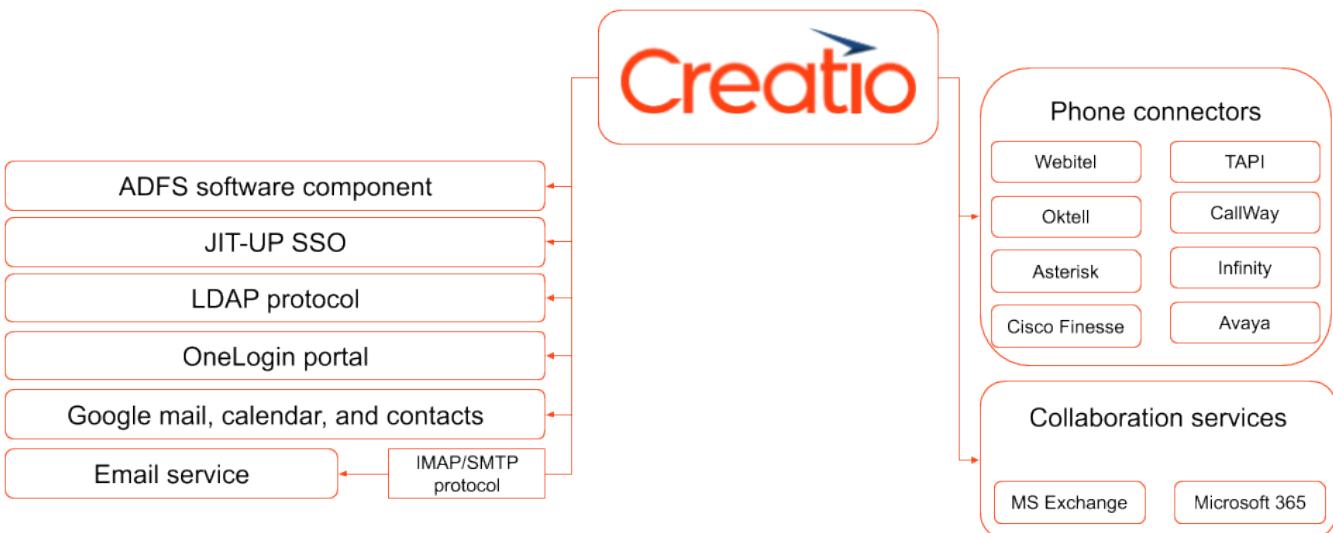
Creatio supports integration with a custom RESTful API via no-code tools. After setting up a web-service integration, you can call it from a business process. The REST API tools support interaction with third-party web services without coding.

Learn more about web service integration in the “[Calling web services from business processes](#)” article.

Out-of-the-box integration solutions

The scheme of out-of-the-box Creatio integration solutions is presented in Figure 2.

Fig. 2. Out-of-the-box Creatio integration solutions



You can integrate Creatio with the following external applications:

- the [OneLogin](#) SSO portal, used as a single sign-on point for all your services, including Creatio. Learn more about OneLogin integration in the “[Setting up Single Sign-On via OneLogin](#)” article.
- the Active Directory Federation Services (ADFS) software component used for managing single sign-on for all system users. Learn more about Active Directory Federation Services integration in the “[Setting up Single Sign-On via ADFS](#)” article.
- the Just-In-Time User Provisioning (JIT UP) Single Sign-On, which alleviates the need to create accounts for each separate service and keep the user database up-to-date. Learn more about Just-In-Time User Provisioning integration in the “[Setting up Just-In-Time User Provisioning](#)” article.
- the Lightweight Directory Access Protocol (LDAP), which enables access to the dedicated database used to store the credentials of users, machines, etc. Learn more about Lightweight Directory Access Protocol integration in the “[Setting up LDAP integration](#)” article.
- email providers by the IMAP/SMTP protocol. Learn more about email integration via the IMAP/SMTP protocol in the “[Integration with email services by the IMAP/SMTP protocol](#)” article.
- Google mail, calendar, and contacts. Learn more about integration with Google in the “[Integration with Google services](#)” article.
- phone connectors, such as [Webitel](#), [Oktell](#), [Asterisk](#), [Cisco Finesse](#), [TAPI](#), [CallWay](#), [Infinity](#), and [Avaya](#).
- the [MS Exchange](#) and [Microsoft 365](#) collaboration services. Learn more about MS Exchange and Microsoft 365 integration in the “[Integration with the MS Exchange and Microsoft 365 services](#)” article.

See also

- **External requests authentication to Creatio services (on-line documentation)**
- **The ProcessEngineService.svc web service**
- **DataService**
- **Integration via OData protocol**
- [Integrations](#)

Video tutorial

- [Integrations. Calling external web-services from the program code](#)
- [Data integration. Working with OData/DataService](#)

Developer tools

Contents

- **Introduction**
- **For solving simple customization tasks**
- **For solving complex customization tasks and server-side development**

Developer tools

Beginner

Easy

Medium

Advanced

Introduction

In Creatio, “customization” refers to the implementation of any functionality that is not part of a standard product package. Customization examples include creating new sections, setting up default values, adding validations, applying default filtering, adding pop-up hints, configuring the editable columns, etc. Creatio comes with a variety of customization tools for implementing customization tasks of varying complexity.

Basic customization tasks can be implemented with built-in Creatio developer tools and do not require installing any additional software. Basic tasks include, for instance, adding standard interface elements, such as fields and buttons.

Developers perform basic customizations in the **[Configuration] section**. Here they can manage configurable Creatio components also known as “configuration elements”: **objects**, **source code**, **modules**, **business processes**, etc. Configuration elements implement a specific set of functions and are grouped in **packages**. The **version control system (SVN)** is used for managing changes in custom packages, as well as for **transferring changes to other environments**. If necessary, developers can **debug the source code of the configuration schemas** with built-in browser tools.

Unlike basic tasks, solving complex customization tasks implies working with the server code and developing projects that provide a wide range of functions, such as creating a configuration service, implementing multi-language features, etc.

Such tasks usually require collaboration between several developers, as well as the use of additional third-party tools for implementing different development stages: team development using version control systems, working with the databases, logging, debugging and transferring the solutions. For this, Creatio provides **file system development mode**.

Detailed instructions for working with additional tools on different developing stages are available in the **list of articles**. Here you can find information on **setup** and **using the integrated Microsoft Visual Studio development environment** when **working with the server code**, **using version control systems**, **debugging the server code**, DBMS-related specifics (MS SQL, Oracle, PostgreSQL).

Testing of separate program components with a .NET application Unit-testing framework, NUnit, is covered in the **“Testing tools. NUnit” article**. Logging with third-party-libraries (NLog) is covered in the **“Logging in Creatio. NLog” article**.

For solving simple customization tasks

Contents

- **Introduction**
- **Development tools. Built-in IDE**
- **Development tools. Packages**
- **Version control system. Built-in IDE**
- **Debugging tools**
- **Delivery tools. Built-in IDE**

Simple customization tasks

Beginner

Easy

Medium

Advanced

Introduction

You can implement basic customization tasks with built-in Creatio developer tools without installing any additional software.

Customization is primarily done in the **[Configuration] section**, which has the following functions:

- managing configuration packages and their contents,

- organizing the integration with version control systems,
- transferring changes between the development environments.

The functionality is implemented in packages that include configuration components. To simplify the development process for configuration items Creatio provides the following designer tools:

- **object designer**,
- **module designer**,
- **source code designer**,
- **process designer for embedded business processes**,
- **process task designer**,
- **image lists designer**.

For each configuration package, a developer sets **dependencies** and **data binding**. To test the implemented functionality, developers can **debug the code** in the application with the built-in browser tools. Using a **version control system** provides access to package change history and facilitates **transferring changes** to other development environments.

Development tools. Built-in IDE

Contents

- **Introduction**
- **The [Configuration] section. The [Data] tab**
- **Source code and metadata viewport**
- **Designers of configuration items**
- **Creating the entity schema**
- **Creating a custom client module schema**
- **Creating the [Source code] schema**

Built-in IDE. The [Configuration] section

Beginner

Easy

Medium

Advanced

Introduction

The [Configuration] section is designed for managing configuration elements that implement Creatio configuration functions.

The [Configuration] section tools enable:

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

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

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

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

The section interface is available on Fig. 1.

Fig. 1. The [Configuration] section

The screenshot shows the Creatio Configuration interface. On the left, there's a sidebar titled 'Actions' with sections like General, Configuration, Source Code, Metadata, Profile, Database Structure, SQL script, and Data. Below this is a 'Packages' list containing items such as ActionsDashboard 7.8.0, Base 7.8.0, Base_ENU 7.8.0, etc. To the right is a large grid titled 'Schemas' with columns for Name, Package, Title, and Database Update Required. The grid lists numerous schema names, their packages, titles, and update requirements.

Actions in the [Configuration] section

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

General

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

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

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

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

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

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

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

Configuration

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

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

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

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

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

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

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

Source Code

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

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

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

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

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

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

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

Metadata

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

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

Profile

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

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

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

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

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

Database Structure

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

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

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

SQL script

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

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

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

Data

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

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

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

Configuration elements

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

Schemas

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

Different types of schemas are identified with different icons:

 – objects

 – business processes and their actions

 – reports (legacy configuration element)

 – source code and client modules

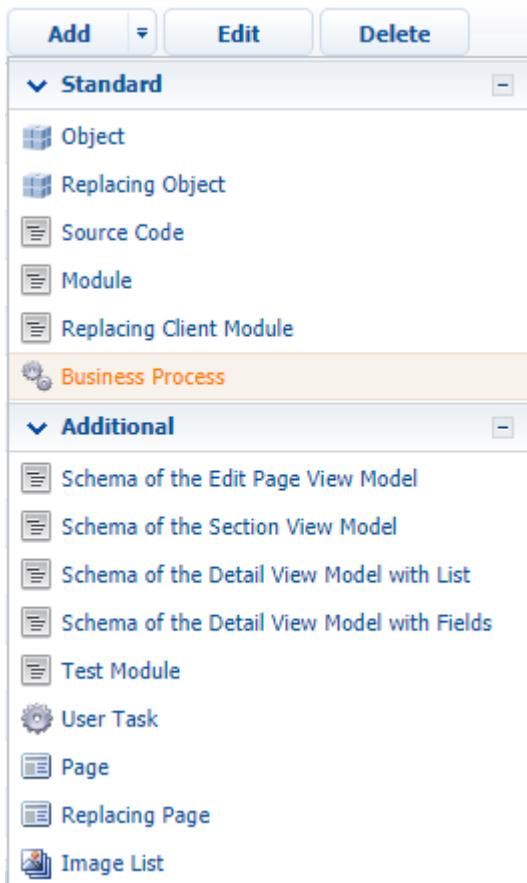
 – image lists (legacy configuration element)

 – pages (legacy configuration element).

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

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

Fig. 2. Selecting the schema type



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

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

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

External Assemblies

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

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

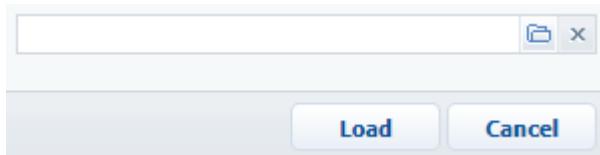
The screenshot shows a table on the 'External Assemblies' tab. At the top, there is a search bar labeled '<Enter search text>' and a 'Search' button. Below the search bar are several filter dropdowns: 'Schemas: All', 'External Assemblies: All', 'SQL Scripts: All', 'Data: All', and 'Package Dependencies'. There are also 'Add' and 'Delete' buttons. The main table has two columns: 'Name' and 'Package'. The data in the table is as follows:

Name	Package
Google.Apis.dll	Base
Google.GData.Client.dll	Base
Google.GData.Contacts.dll	Base
Google.GData.Extensions.dll	Base

At the bottom of the table are various navigation and search icons.

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

Fig. 4. Adding an external library to a package



SQL Scripts

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

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

Schemas: All	External Assemblies: All	SQL Scripts: All	Data: All	Package Dependencies
Add	Edit	Delete		
Name	Package	Needs to be installed in database		
ActualizeAdminUnitInRoleMSSql	Base	<input type="checkbox"/>		
ActualizeCustomPackageUIDSettingsVal...	Base	<input type="checkbox"/>		
ActualizeCustomPackageUIDSettingsVal...	Base	<input type="checkbox"/>		
AddIndexOnQRTZ_TRIGGEROracle	Base	<input type="checkbox"/>		
ClearSysSchemaFolder	Base	<input type="checkbox"/>		
CreateClientIPColumnInTrackChangesT...	Base	<input type="checkbox"/>		

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

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

Add	Edit	Delete
Add		
Add file		
itInRoleMSSql	Package	
ActualizeCustomPackageUIDSettingsVal...	Base	

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

The toolbar also contains buttons for editing:

[Edit] – edit a previously added SQL script.

[Delete] – delete a previously added SQL script.

Data

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

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

Schemas: All	External Assemblies: All	SQL Scripts: All	Data: All	Package Dependencies
Add	Edit	Delete		
Name	Package	Schema		
AcademyURL_SysLookup	Base	AcademyURL		▲
Account	Base	Account		≡
AccountAnnualRevenue	Base	AccountAnnualRevenue		≡
AccountCategory	Base	AccountCategory		≡
AccountEmployeesNumber	Base	AccountEmployeesNumber		≡
AccountIndustry	Base	AccountIndustry		▼

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

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

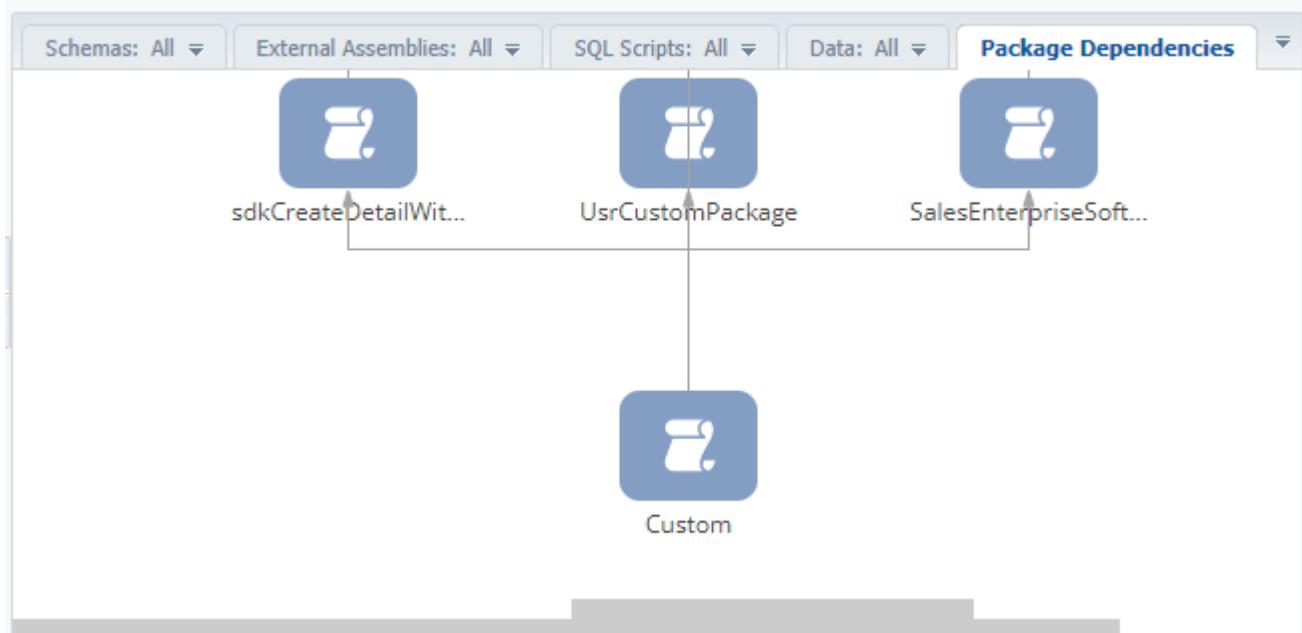
[Edit] – edit previously added data.

[Delete] – delete previously added data.

Package Dependencies

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

Fig. 8. The [Package Dependencies] tab



The [Configuration] section. The [Data] tab

Beginner Easy **Medium** Advanced

Introduction

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

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

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

The screenshot shows a software interface for managing data packages. At the top, there are dropdown menus for 'Schemas: All', 'External Assemblies: All', 'SQL Scripts: All', and 'Data: All'. Below these are buttons for 'Add', 'Edit', and 'Delete'. The main area is a table with three columns: 'Name', 'Package', and 'Schema'. The 'Name' column lists various entities like 'AcademyURL_SysLookup', 'Account', etc. The 'Package' column shows all entries as 'Base'. The 'Schema' column shows the corresponding schema names. At the bottom of the table are standard grid navigation buttons.

Name	Package	Schema
AcademyURL_SysLookup	Base	AcademyURL
Account	Base	Account
AccountAnnualRevenue	Base	AccountAnnualRevenue
AccountCategory	Base	AccountCategory
AccountEmployeesNumber	Base	AccountEmployeesNumber
AccountIndustry	Base	AccountIndustry

The [Data] tab actions

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

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

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

[Delete] – delete previously bound data.

The page of binding data to packages

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

The [Properties] tab

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

Fig. 2. The [Properties] tab

The screenshot shows the 'Properties' tab of a data binding configuration page. On the left, there's a panel for defining columns with fields for 'Name', 'Forced update', and 'Key'. It lists columns like 'Id', 'Unique identifier of object', and 'Type column'. Below this is a condition builder with a 'OR' clause: 'Id that is equal to ee69280c-47a7-4120-a1f3-46fafbe7efcf'. On the right, the 'Display data' tab is active, showing a table with one record. The table has columns 'Type column' and 'Unique identifier of object', with the value '0a22fbe-144c-46cc-b2ec-ff5d1e2767a7' listed. Navigation buttons are at the bottom of both panes.

The tab contains the following fields and groups:

1. [Name] – the name of the data bound to a package.
2. [Object] – the object that package bound data are connected to. Use the caption and not the object name when you select it.
3. [Installation type] – specificities of adding the bound data to the application during package installation. The following types are available:

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

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

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

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

The tab actions:

1. [Display data] – displays data bound to a package.
2. [Save] – saves the data bound to a package and closes the page of binding data.
3. [Cancel] – closes the page of binding data without binding the data.

The [Bound data] tab

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

Fig. 3. The [Bound data] tab

Type column	Unique identifier of object
	0a22fbae-144c-46cc-b2ec-ff5d1e2767a7

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

If you need to apply any changes of the connected data in the development environment, additionally save the

changed data connection element in the package. Otherwise, the data connection element will contain the old (unchanged) data.

Recommendations

1. The data installation type depends on how a customer will further work with data after the update. The [Installation] type should be used in most cases.
2. We recommend to delete the [Created by], [Modified on], [Modified by], [Active processes] standard columns from the list of columns to bind. Leave only the [Created on] service column.
3. We recommend you to be very careful with using the [Forced update] property of the bound columns. Avoid selecting it for system setting values, external system keys, web-service URLs, i.e. for all columns that affect Creatio operation capacity.
4. Filter the bound data by identifier or object name (code). We do not recommend using object captions, modification dates, etc.

Typical mistakes in data binding

1. Additional data, such as custom lookup or system setting values are not bound for new functions created in custom packages. New functions do not work after installation of a package into a new application, since the necessary data are missing.
2. Filters are not set during binding data of a new custom section. As a result, all data including those used for testing will be bound to the package. Such data volume can be quite big, which leads to longer package installation.
3. The bound data were not applied during package installation via SVN, for example, when automatic data applying is disabled (see "[Installing packages from repository](#)").

Source code and metadata viewport

Beginner

Easy

Medium

Advanced

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

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

Fig. 1. – Source code viewport

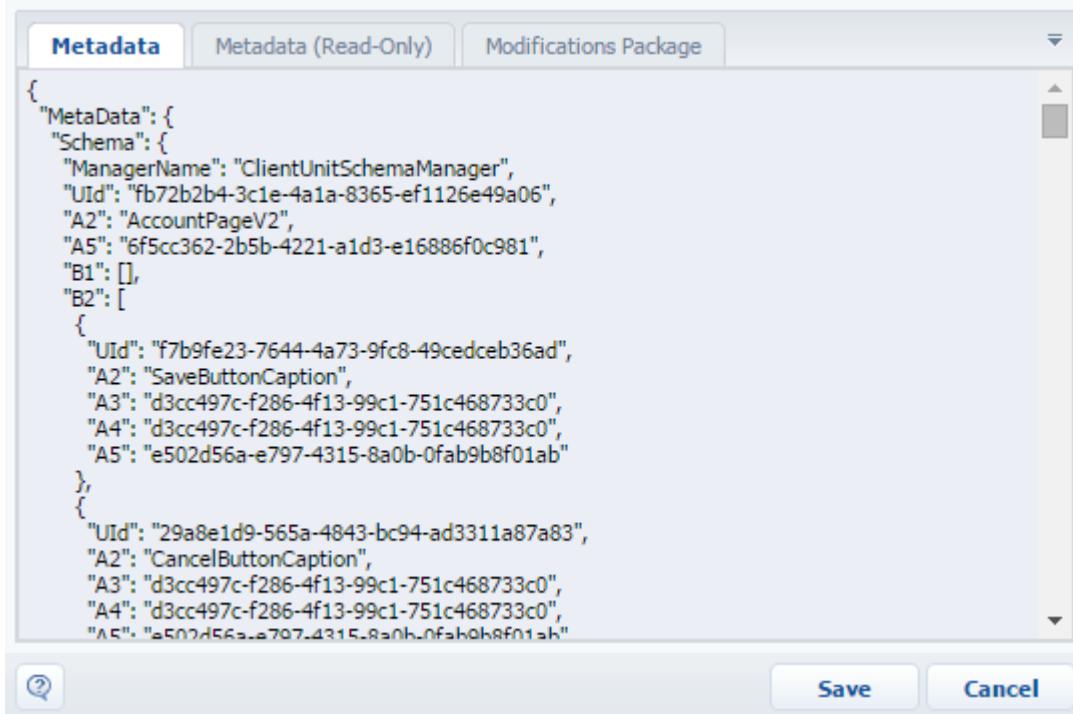
The screenshot shows a code editor window titled "AccountInfoByTypeConverte". The code is written in C# and defines a class with two methods: Evaluate and getAccountInfo. The Evaluate method takes an object value and a string arguments, and returns a string. It uses a try-catch block to handle exceptions, returning the error message if arguments is "DEBUG" and "mistake" otherwise. The getAccountInfo method takes a Guid account ID and returns a string, using EntitySchemaQuery to perform the query.

```
25  private _UserConnection _userConnection;
26
27  public string Evaluate(object value, string arguments = "") {
28      try {
29          _userConnection = (UserConnection)HttpContext.Current.Session["U";
30          Guid accountId = new Guid(value.ToString());
31          return getAccountInfo(accountId);
32      } catch (Exception err) {
33          if (arguments == "DEBUG") {
34              return err.Message;
35          } else {
36              return "mistake";
37          }
38      }
39  }
40
41  private string getAccountInfo(Guid accountId) {
42      try {
43          EntitySchemaQuery esq = new EntitySchemaQuery(_userConnection.EntitySchemas["Account"]);
44          esq.AddFilter("id", accountId);
45      }
46  }
47
48  public void Dispose() {
49      _userConnection.Dispose();
50  }
51 }
```

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

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

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

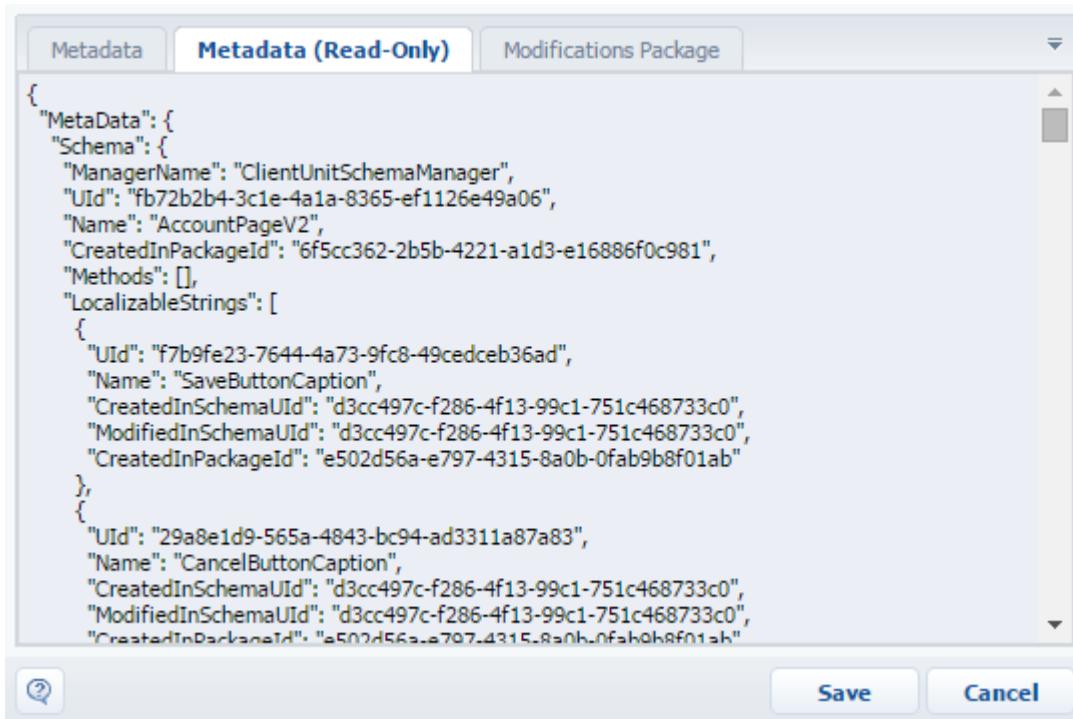


The [Metadata] tab shows metadata in their initial view. Use this tab in order to edit metadata manually.

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

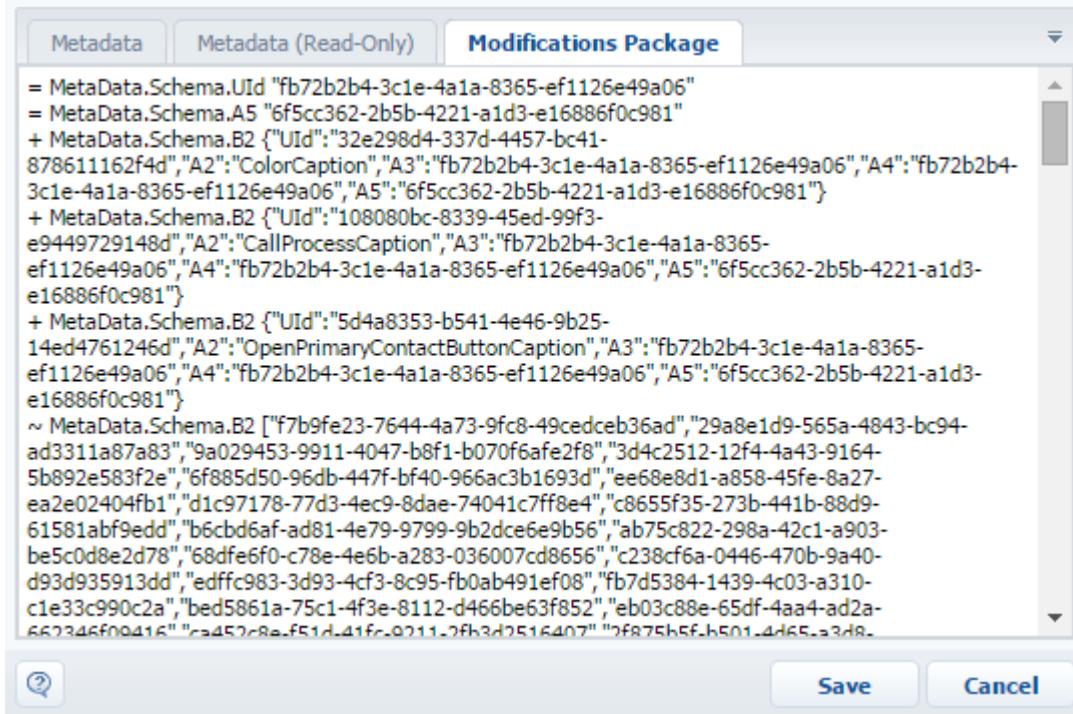
The [Metadata (Read)] (Figure 3) displays data that is similar to that displayed by the [Metadata] tab, but in a form suitable for reading. Internal identifiers (for example, "A2") are replaced with actual values of items, specified in the [Name] property field (for example, "AccountName"). This tab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

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



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

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



Designers of configuration items

Contents

- **Workspace of the Object Designer**
- **Source code designer**
- **Module designer**

- The process designer for embedded business processes
- User task designer workspace
- Workspace of image list designer

Workspace of the Object Designer

Beginner

Easy

Medium

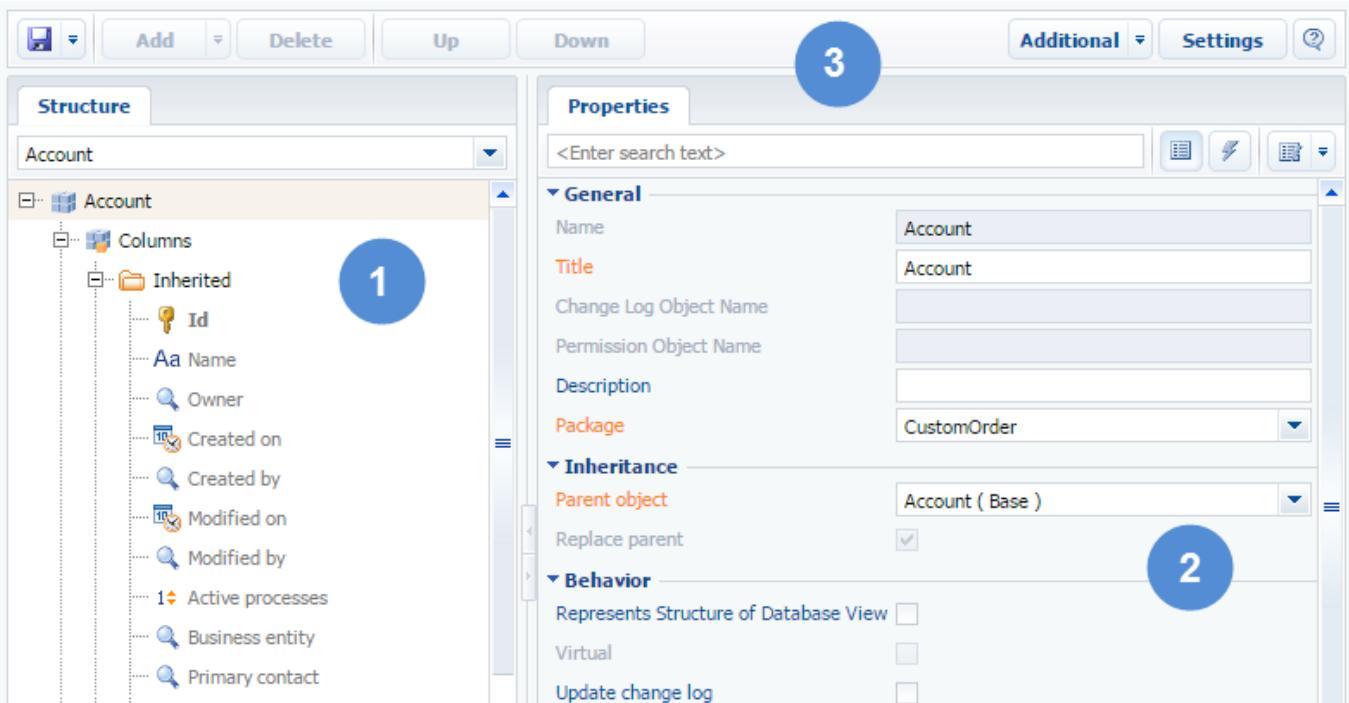
Advanced

Introduction

Use the Object Designer to configure entity schemas (*EntitySchema*). You can add new columns, indexes, in-built processes to the schema, as well as define the properties and events of the schema elements.

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

Fig. 1. — The object designer workspace



Object structure area (1)

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

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

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

Properties and events area (2)

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

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

Toolbar (3)

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

Add	Add an item to the object structure. The menu contains the list of all available types of columns and indexes.
Delete	Delete of columns from the object. Deleting columns from an object is similar to deletion of columns from the corresponding table of the system database.
Up	Move the item up in the object structure.
Down	Move the item down in the object structure.

Settings window

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

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

Module designer

Beginner

Easy

Medium

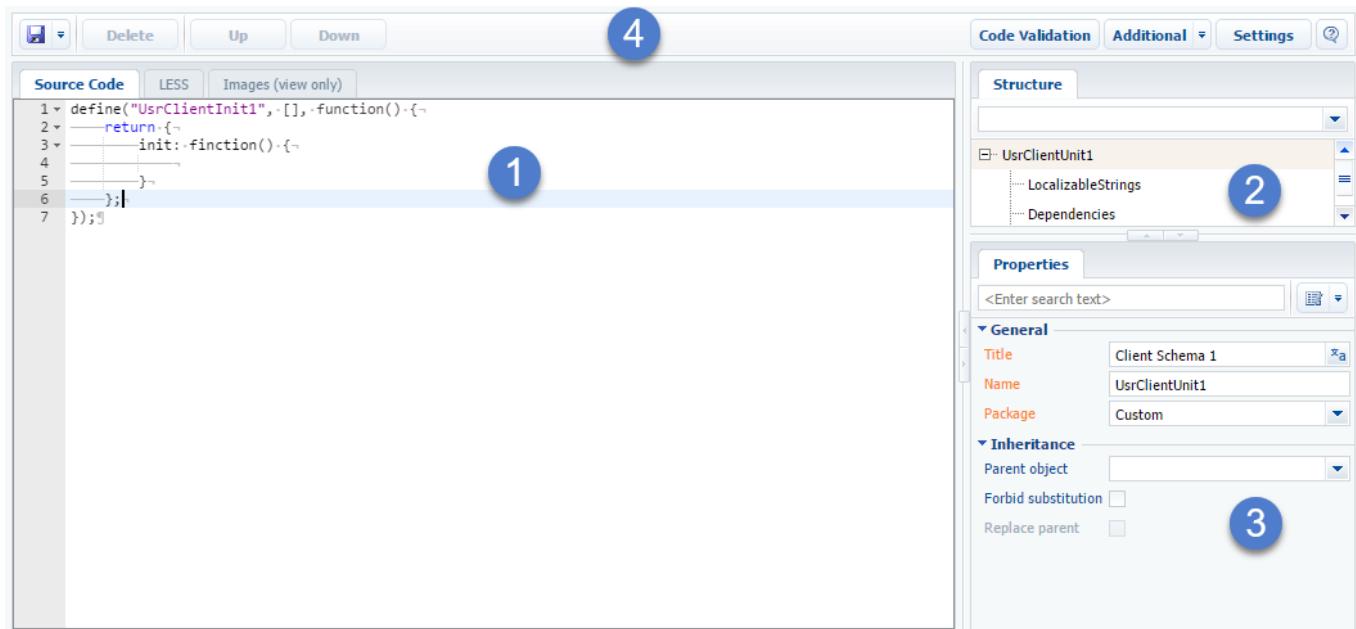
Advanced

Introduction

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

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

Fig. 1 User interface of the module designer



Source code editor

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

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

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

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

Schema structure window

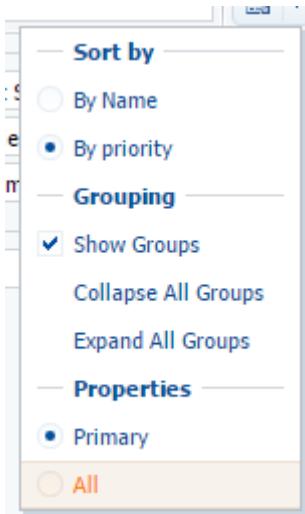
The schema structure area (2) is used to display the schema structure; the root element, resources, dependencies, messages and parameters.

Property window

The properties of the element selected in the structure area (2) are displayed in the properties window (3). If the root element of the structure is selected, then the main properties of the schema source code are displayed. If the localized string is selected, then the properties of the localized string are displayed.

Configuration of the properties view is performed with menu commands (Fig. 2). Select the [All] command to display all properties of the selected element of the structure.

Fig. 2 View properties configuration commands

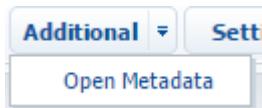


Toolbar

The module designer toolbar (Fig. 1,4) has the following menu and buttons:

Name	Purpose
Save	Menu with save and publish commands.
Delete	Deletes the selected element of the module structure.
Up	Moving an element to a position higher from its current position in the module structure.
Down	Moving an element to a position lower from its current position in the module structure.
Code validation	Performs source code check for formatting errors. Result is displayed on the [JS errors] tab.
Additional	Contains commands for opening metadata view windows (Fig. 3).
Settings	Opens the settings window

Fig. 3 Additional module designer commands

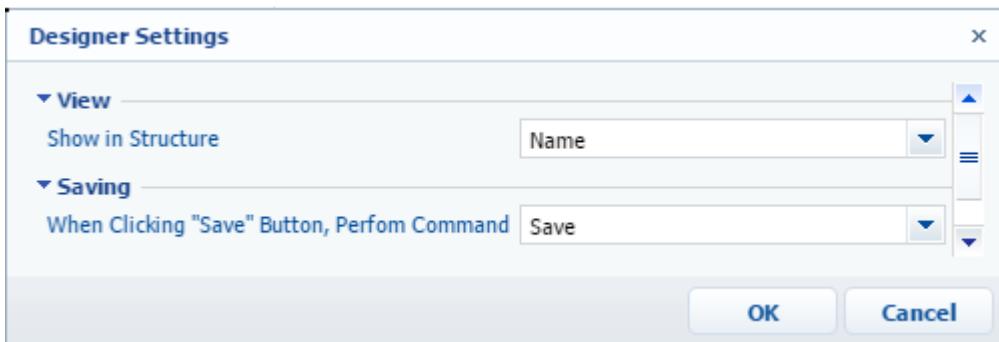


Settings window

The window of the module designer settings (Fig. 4) has the following items:

Name	Purpose
Show in Structure	Select to show a name or a title of the schema element in the structure.
When Clicking "Save" Button, Perform Command	Select the command that will be performed after clicking the [Save] button. Possible options include saving metadata and publishing a schema.

Fig. 4 Module designer settings



Source code designer

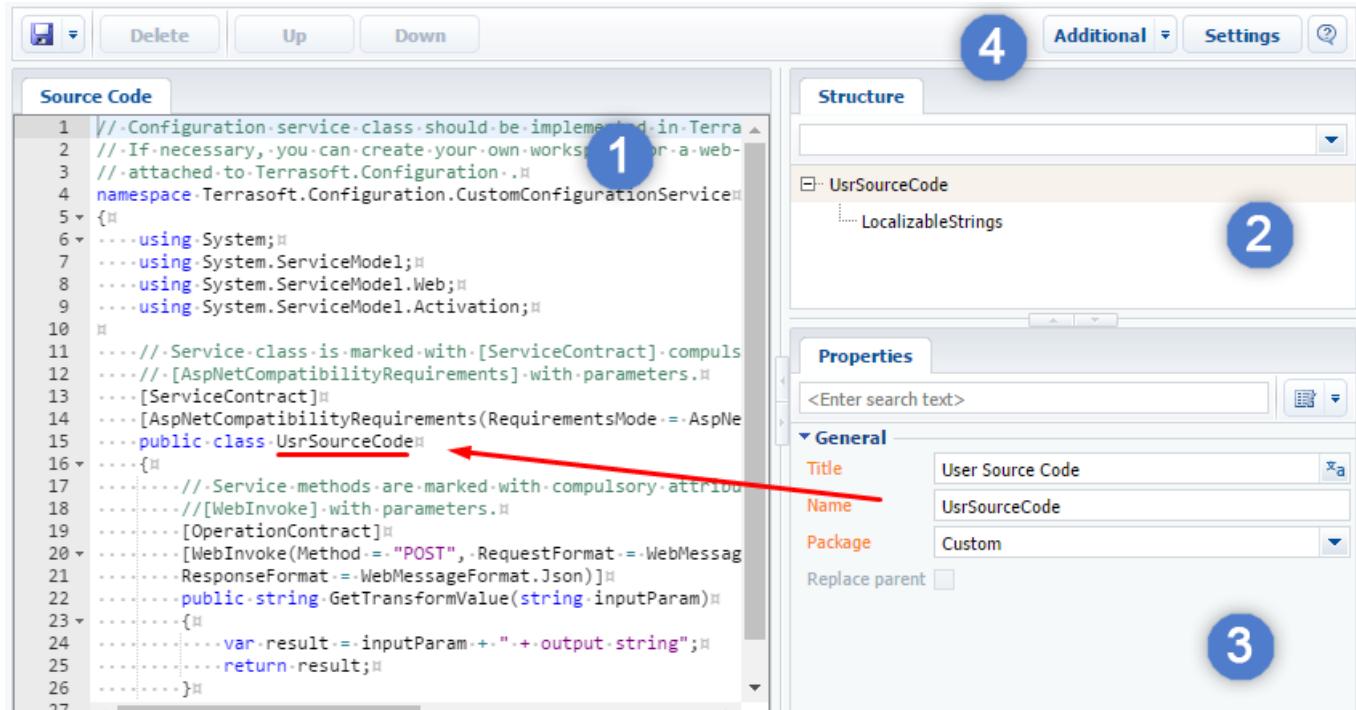
Beginner Easy Medium Advanced

Introduction

Source code designer is used to configure schemas of the *SourceCodeSchema* type. Use it to add the C# source code of classes to the schema and localizable strings used to localize UI text implemented by these classes.

The working area of the source code designer (Fig. 1) consists of several functional areas and contains the controls and tools necessary for creating source code schemas.

Fig. 1. Source code designer



Source code editor

Use the source code editor area (1) to edit the C# source code of custom classes. Add, delete and format the source code of custom functions. Note: Source code debugging is not available in the editor.

Schema structure window

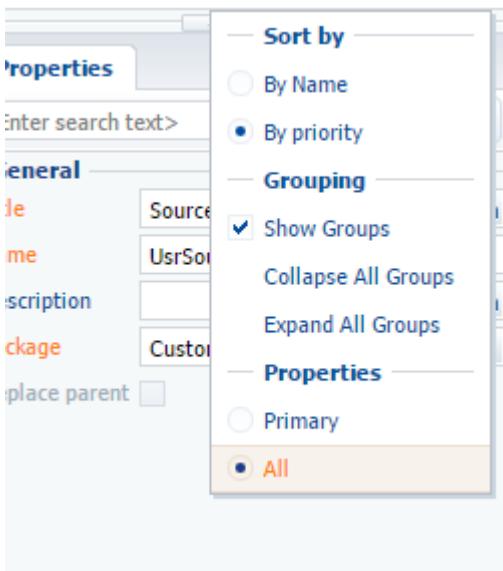
The schema structure (the root element and localizable strings of the source code) is located in the structure area (2).

Properties window

The properties window (3) displays the properties of the element selected in the schema structure (2). If you select the root element of the structure, you will see the general properties of the source code schema on the right-hand side. If you select a localizable string, you will see its properties.

Use the context menu (Fig. 2) to display all available properties of the selected element in the schema structure. Click [All] to switch to advanced mode.

Fig. 2. Context menu

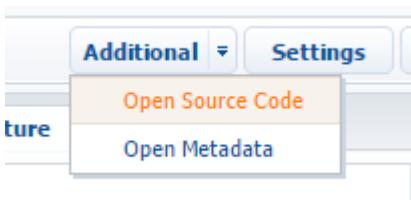


Toolbar

The source code designer toolbar features the following menus and buttons:

Name	Purpose
Save	Saving and publishing the schema.
Delete	Deleting the selected object.
Up	Moving the selected element up.
Down	Moving the selected element down.
Additional	Opening the source code or the metadata window (Fig. 3).
Settings	Opening the settings window.

Fig. 3. Source code designer settings



Settings window

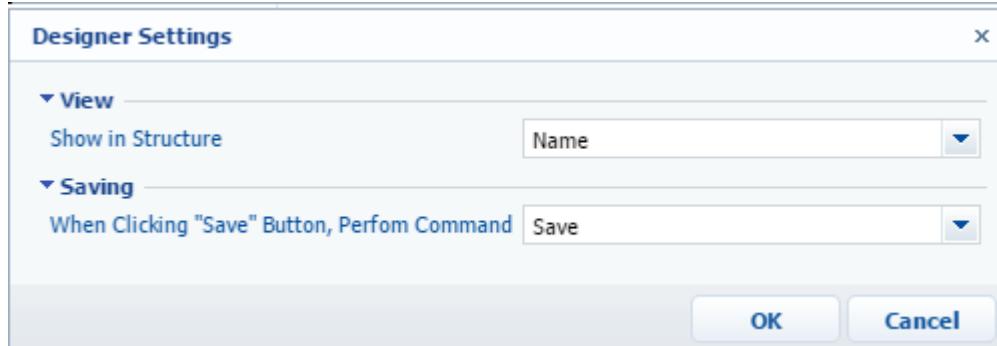
The source code designer settings window (Fig. 4) includes:

Name	Purpose
Show in Structure	Choosing to display either the "Name" or the "Title" of the schema elements in the structure.

When Clicking "Save" Button, Perform Command

Choosing what happens when you click the [Save] button. Possible options – “Save” or “Publish” the schema.

Fig. 4. Source code designer settings



The process designer for embedded business processes

Beginner

Easy

Medium

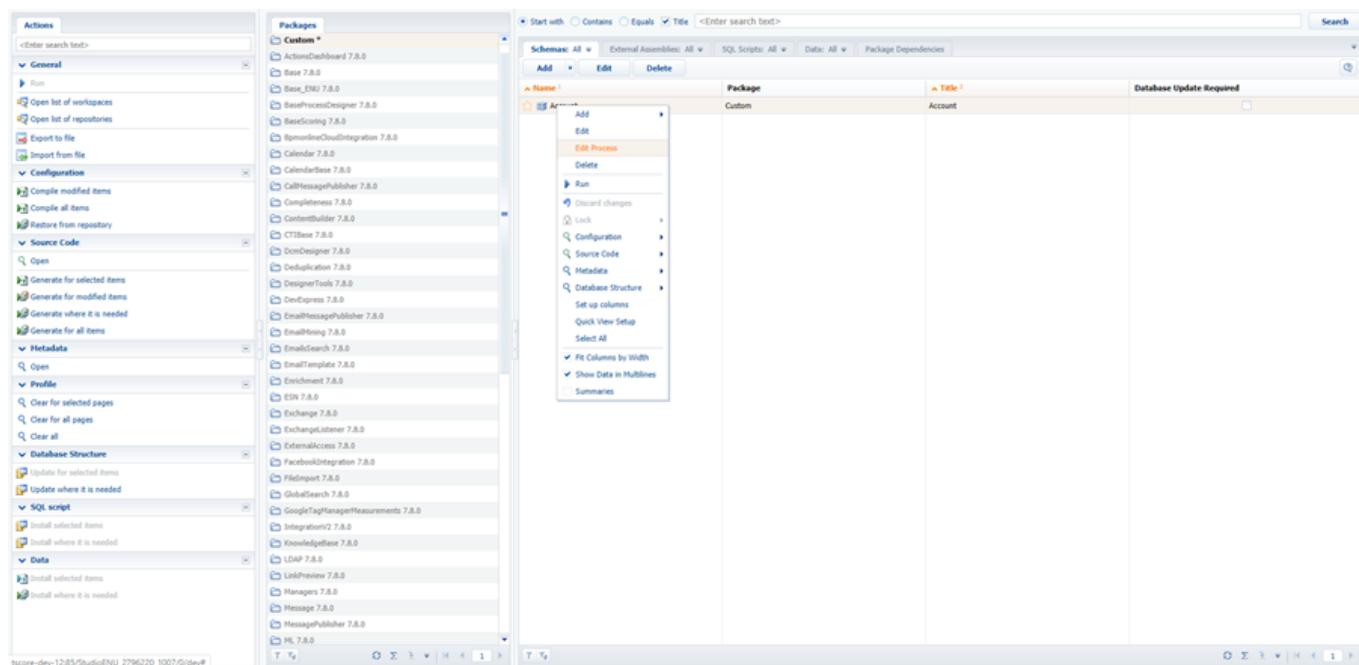
Advanced

Introduction

The embedded business processes designer is intended for creating and configuring business processes embedded into objects (*Entity Schema*). There are two ways to run the process designer:

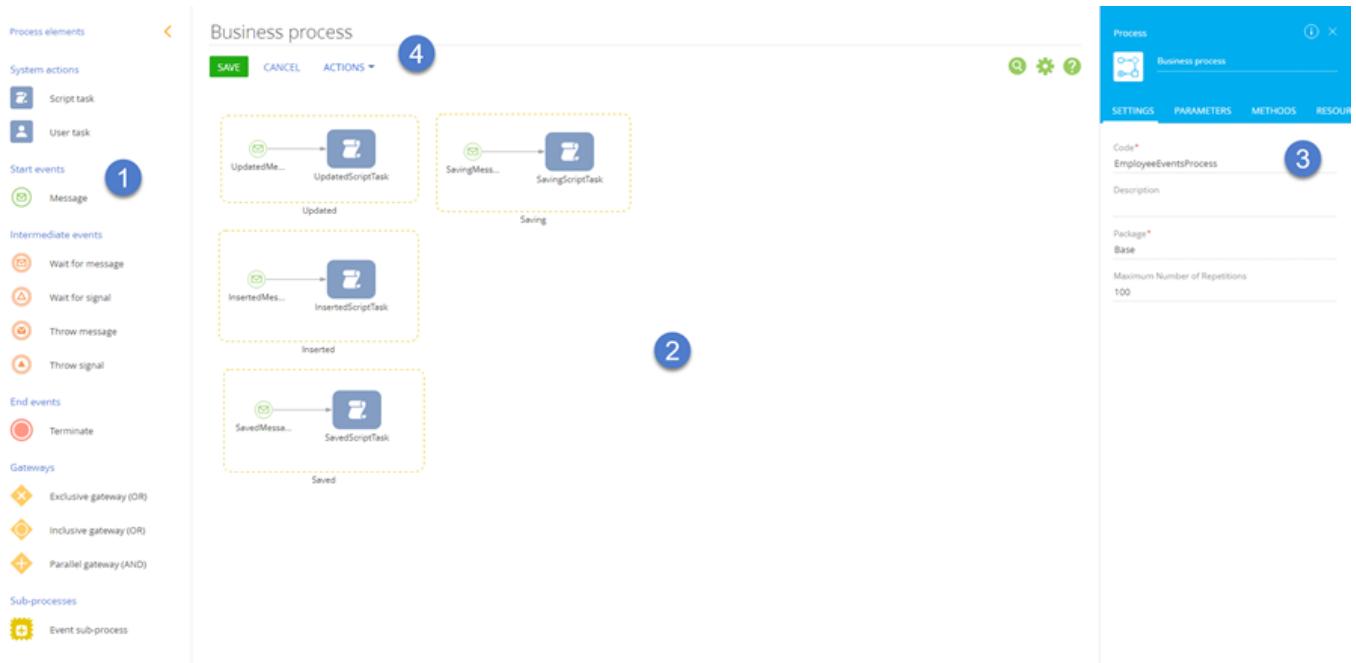
- by running the [Open Process] command in the [Additional] menu from the **object designer**;
- by right-clicking the object and running the [Edit Process] command (fig. 1).

Fig. 1. – Running the process designer



Process designer (fig. 2) consists of several functional areas and has the controls and tools necessary for process creation.

Fig. 2. – The process designer



The process elements area

The [Process elements] area (1) contains a list of elements used to make up a process.

Depending on the intended purpose, the elements are split into the following groups:

- [System actions]
- [Start events]
- [Intermediate events]
- [End events]
- [Gateways]
- [Sub-processes]

Process designer workspace

The process designer workspace (2) provides a schematic representation of the process's algorithm. It is the primary workspace for developing an embedded business process. You can introduce new elements and connection objects into the scheme during development, as well as remove unneeded components. You can also edit the captions and other properties of the process elements that are placed on the workspace.

Process area

The [Process] area (3) includes the following element groups:

- The [Settings] group is intended for configuring of the main properties of an embedded business process, such as the header, the code, the need for logging the execution, etc.
- The [Parameters] group is intended for adding process parameters for information exchange between different processes or between the elements within the same process.
- The [Methods] group is intended for adding program methods for use by the process.
- The [Resources] group is intended for adding localized strings.

In the process designer, you can specify the value of a localized string as a string. Setting the value by using a formula is also supported.

You can use a formula in a conditional flow to define the conditions for moving between process elements.

Toolbar

The Toolbar has the following buttons:

- The [Save] button is used for saving the business process. If no changes were introduced into the process

that would require you to publish it (i.e. an interpreted process), then the users will be working with the updated version of the process after saving.

- The [Cancel] button is used for closing the Process designer without saving the new changes.

The [Actions] menu of the Process designed contains the following commands:

- The [Open Parent Process] command opens the interface of the parent process for the current process in a new tab.
- The [Source Code (Ctrl+K)] command opens a source code window for the current process.
- The [Copy Element (Ctrl+C)] command copies a diagram process element.
- The [Paste Element (Ctrl+V)] command inserts a copied element into the process designer workspace.

The  [Search elements (Ctrl+F)] button is intended for searching process elements.

The  [Process Parameters] button opens the settings page for the elements.

The  [Online Help (F1)] button opens tutorials for working with business processes.

Search in the process designer

To start searching (fig. 3), click the [Search elements (Ctrl+F)] button  or use the Ctrl+F shortcut.

Fig. 3. – The search field in the process designer



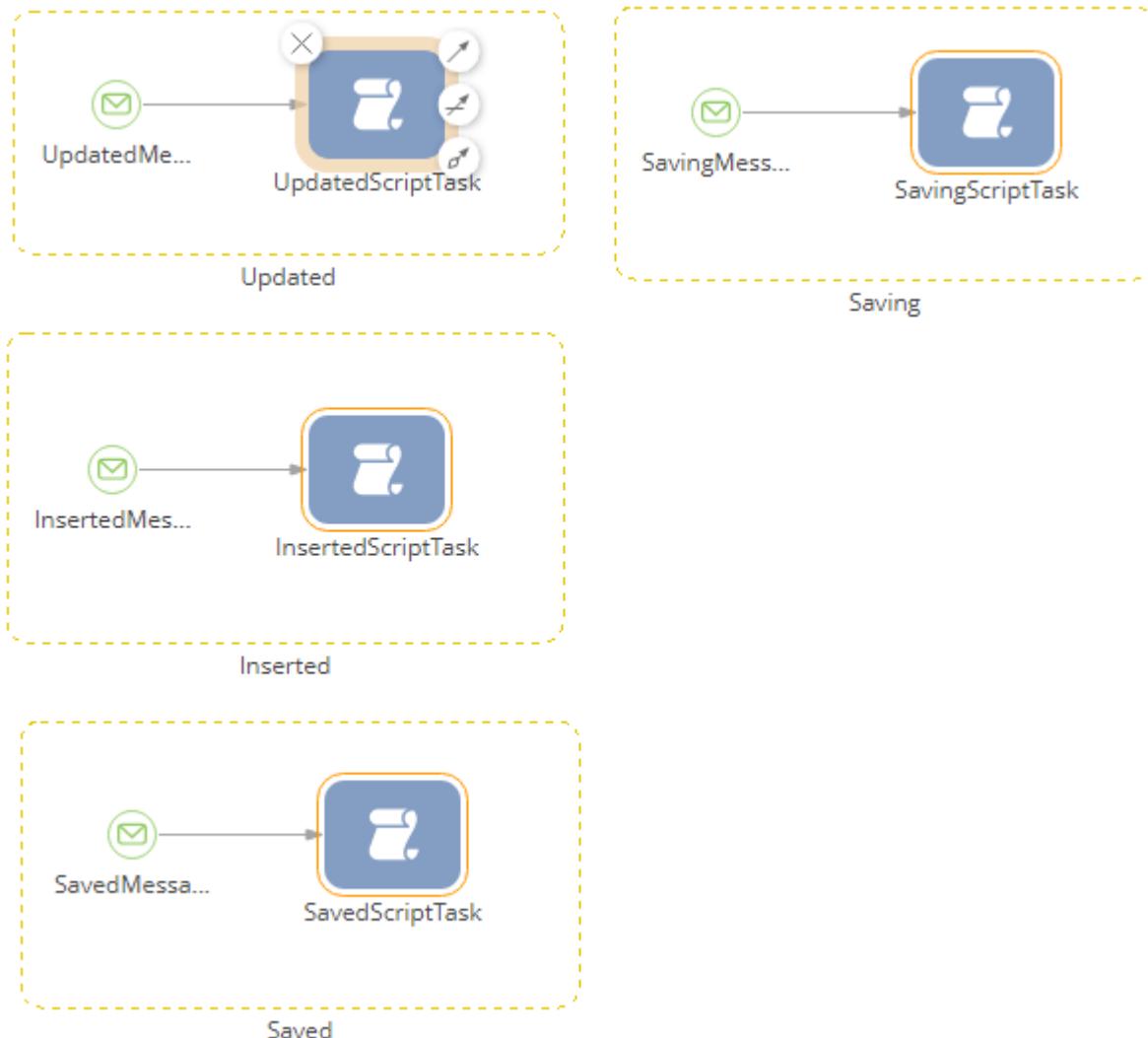
Search is based on the name of the process element or its code. To run the search, click the [Search elements (Enter)] button  or press Enter. On the right side of the workspace, you can find the number of matching process elements (fig. 4). To go back to the previous match, click the [Go to previous (Shift+F3)] button  or use the Shift+F3 shortcut. To go to the next match, click the [Go to next (F3)] button  or press either F3 or Enter.

Fig. 4. – Searching in the process designer



The element with the matching text string will be highlighted orange (fig. 5).

Fig. 5. – Search results in the process designer



To hide the search field, use the [Hide search (Esc)] button or press Esc.

User task designer workspace

Beginner

Easy

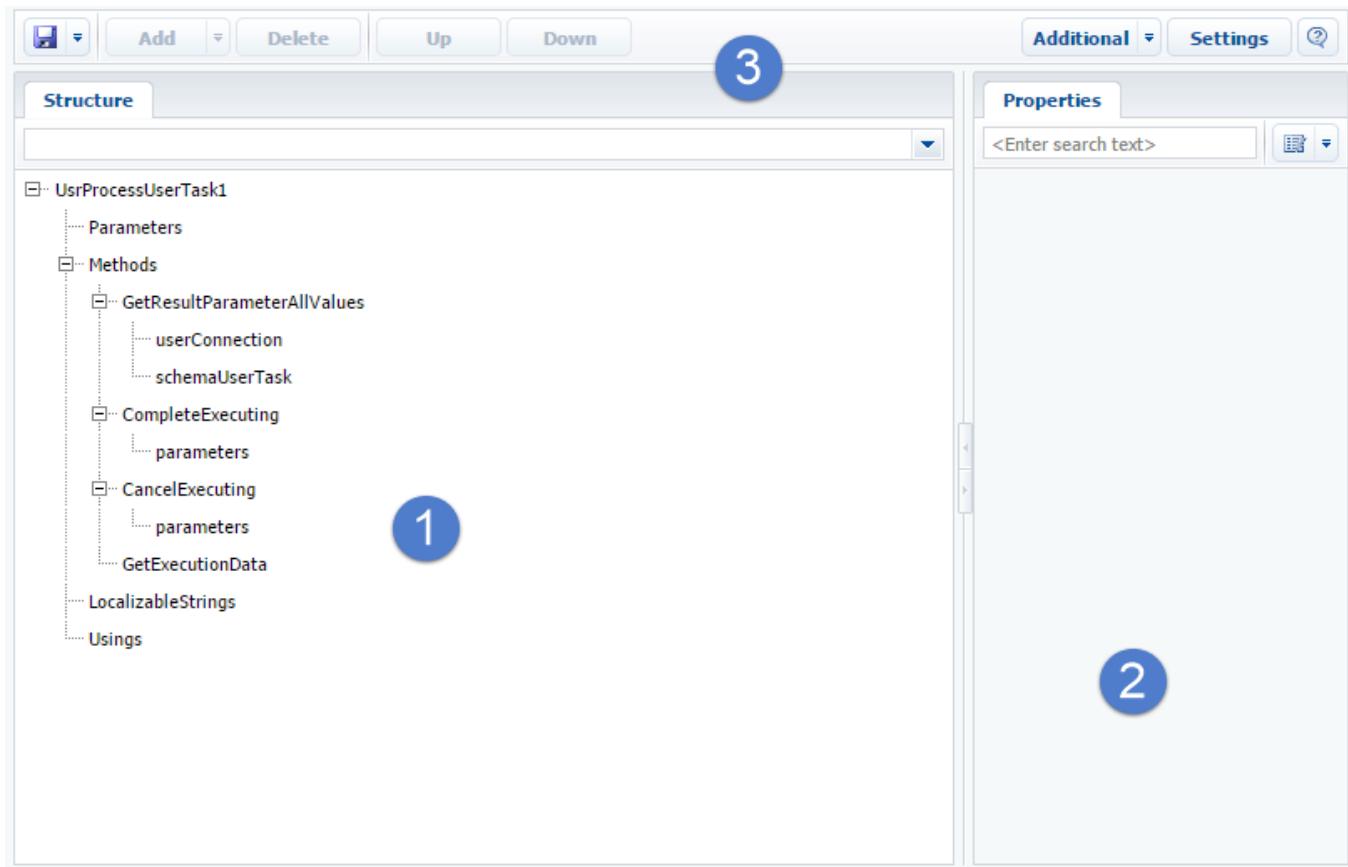
Medium

Advanced

Overview

The User Task Designer (Fig. 1) consists of a number of functional areas and contains tools for creating custom activities for use in business processes.

Fig. 1. User Task Designer work area



Structure area (1)

The [Structure] area contains a tree-like structure of the business process elements.

The Properties Area (2)

Use the [Properties] area to modify the number of separate characteristics of a user task and any of its items.

The Toolbar (3)

In addition to the standard buttons, the toolbar of the user task designer also contains the following buttons:

[Add] – adds an item to the user task structure. The item currently selected in the structure will determine the type of an item that will be added by clicking the button. For example, if the [Parameters] group or a parameter is selected in the structure, clicking the [Add] button will add a new parameter. The [Add] button menu also contains the following commands:

- [Add Parameter] – adds a parameter to the user task structure.
- [Add Method] – adds a method to the user task structure.

You can also add an item by using the [Add] command of the right-click menu in the [Structure] area.

[Delete] – deletes the selected item from the user task structure.

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

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

Workspace of image list designer

[Beginner](#)

[Easy](#)

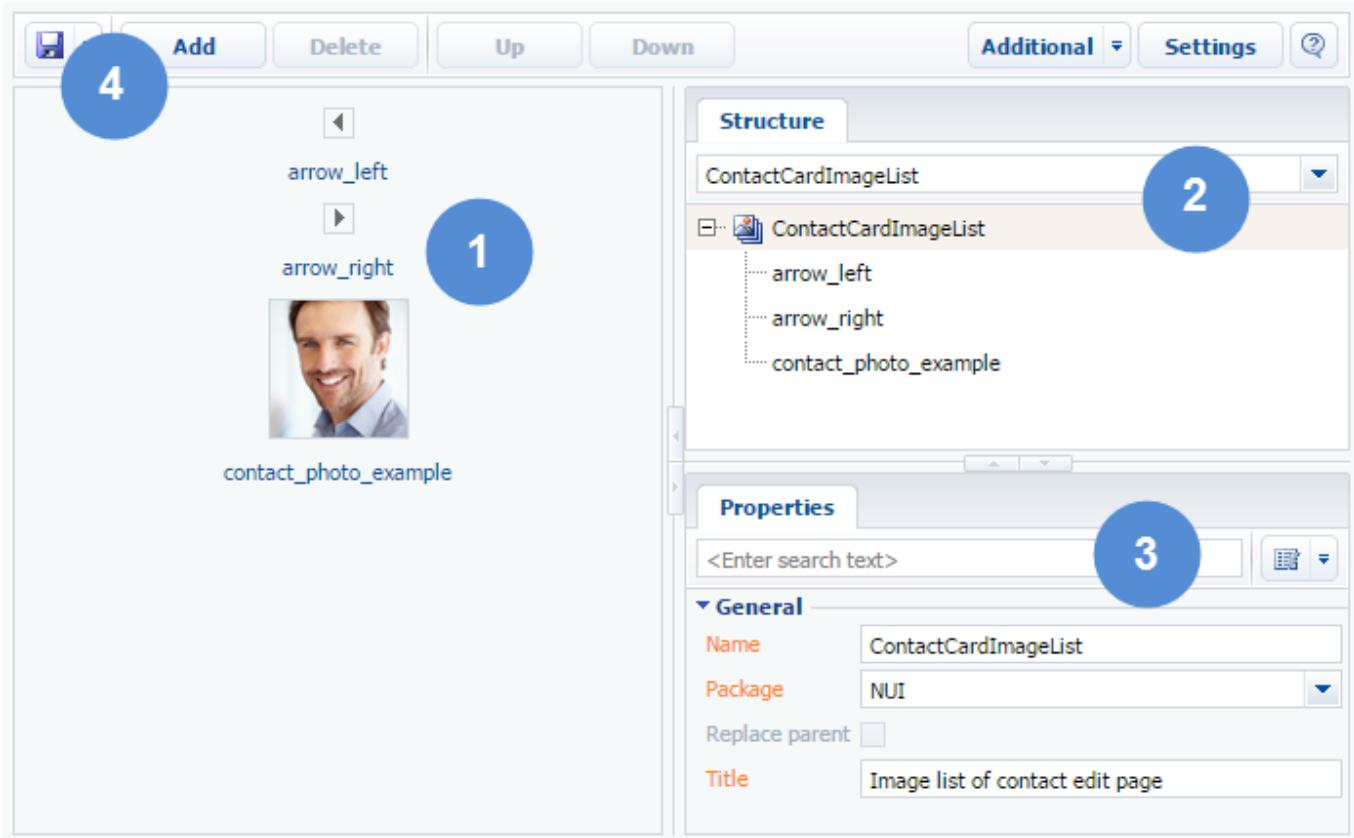
[Medium](#)

[Advanced](#)

Overview

The workspace of the image list designer (figure 1) consists of four main functional areas and contains necessary tools for creating image lists.

Fig. 1. – Working area of image list designer



Specifics of the image list designer are described in chapter, "Specifics of handling of image list designer".

Designer image area (1)

List items in the form of image sketches are located in image areas.

Image list structure area (2)

A tree-type structure of image list items is displayed in the [Structure] area.

Property area (3)

You can change a set of individual characteristics of an image list and also each item in the [Property] area. Image files are downloaded into the list through the same area.

Toolbar (4)

In addition to the standard buttons, the toolbar of image list designer also includes the following buttons.

Add	Add a new item into the list. The item doesn't include images upon adding.
Delete	Delete selected elements from image list.
Up	Movement of the item above its current position in the object structure.
Down	Movement of the item below its current position in object structure.

Creating the entity schema

Beginner

Easy

Medium

Advanced

Introduction

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

Creatio configuration is based around schemas. Every type of the configuration item is represented by a schema of the appropriate type. From the implementation view point, any type of schema is a kernel class that is inherited from the base *Schema* class. More information about schemas and their properties can be found in the "[Configuration architectural elements](#)" article.

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

There are base and custom entity schemas.

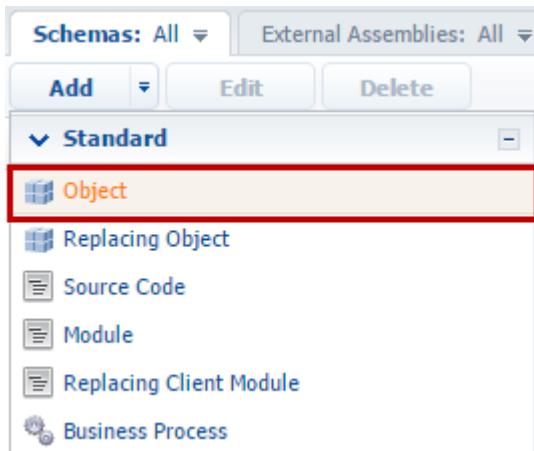
Base entity schemas are not available for editing and are located in the pre-installed packages.

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

Creating a custom object schema

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

Fig. 1. Creating a new object schema



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

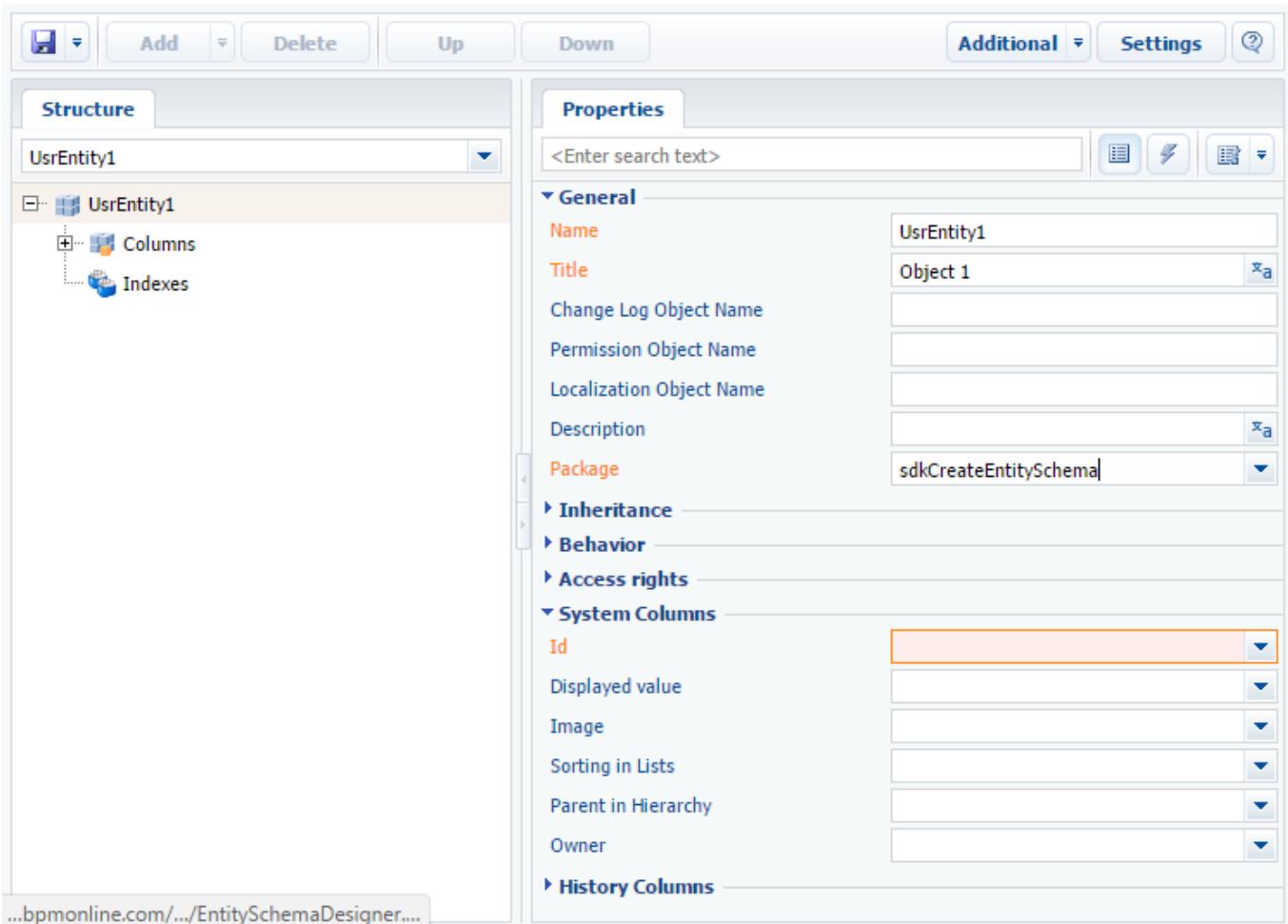
- [Title] – localized schema title. The default value is set by object designer and can be modified.
- [Name] – schema name. The default value is set by object designer and can be modified. Contains the prefix specified in the [Prefix for object name] system setting (*SchemaNamePrefix*). By default the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

Prior to 7.9.1 version, the maximum allowed length of the entity name was 30 characters. Starting with version 7.9.1, the maximum length of the entity name is 128 characters.

Entities with names longer than 30 characters can not be used on Oracle databases that are earlier than version 12.2.

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

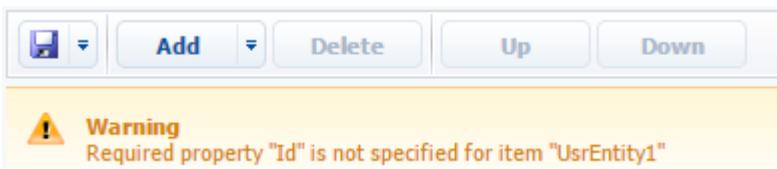
Fig. 2. Required entity schema properties in the object designer



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

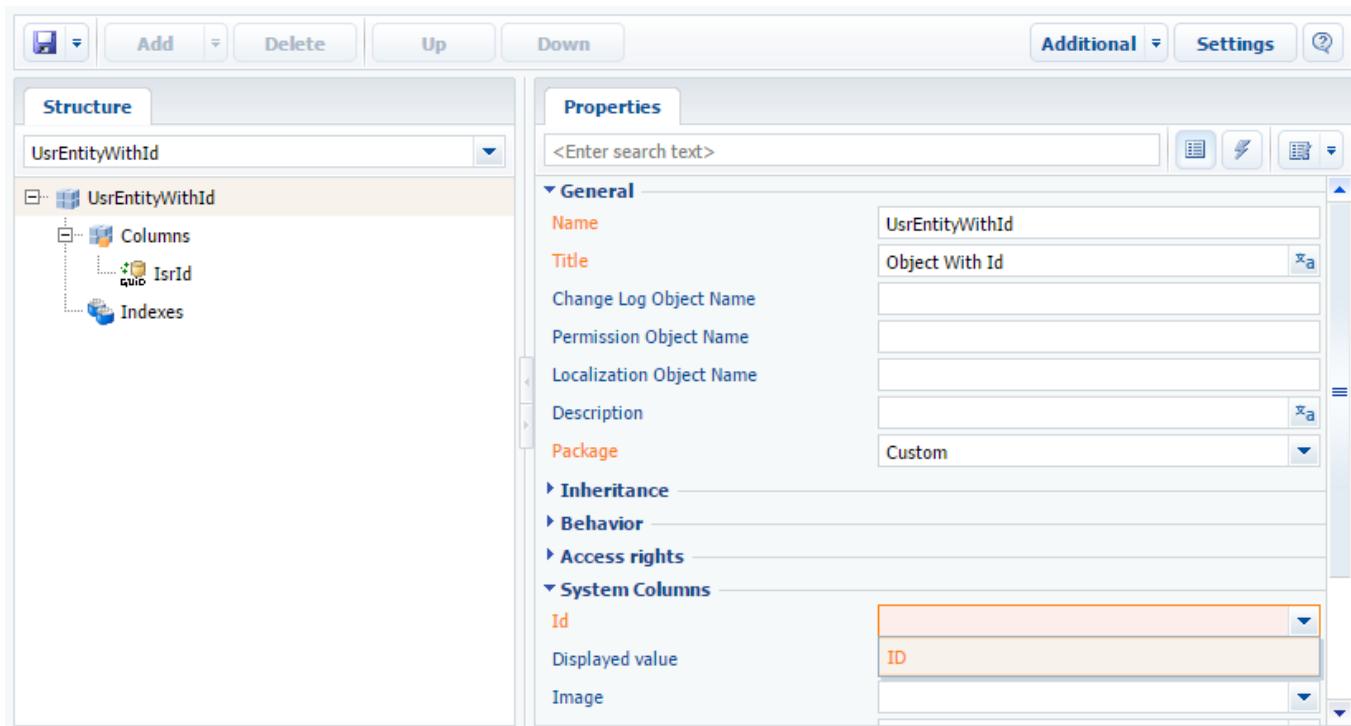
The entity is the representation of the table record in the database and it must have the id column used as a table primary key. If you try to save an object schema without an Id, a warning will pop up (Fig. 3).

Fig. 3. Empty [Id] property warning



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

Fig. 4. Setting the Id column



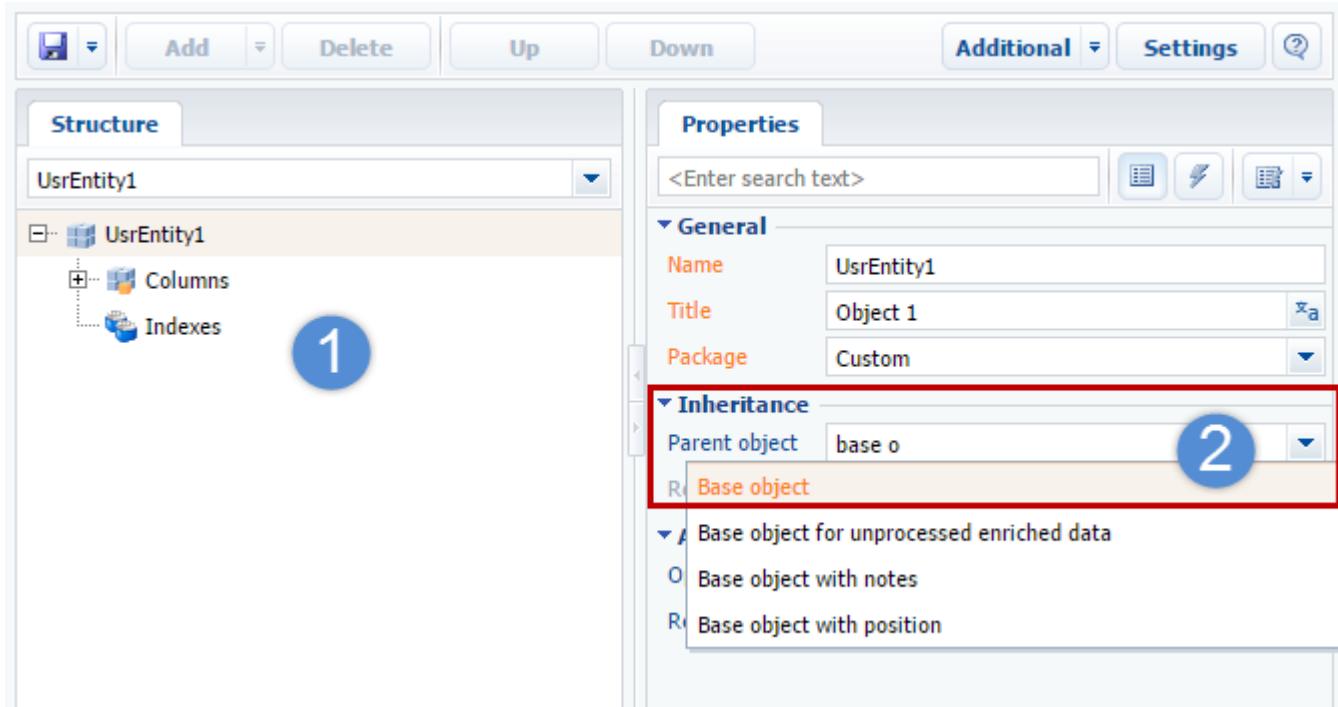
The procedure for adding the column is covered in the “Adding the custom column to the entity” section.

Specifying the parent object

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

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

Fig. 5. Selecting the parent schema



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

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

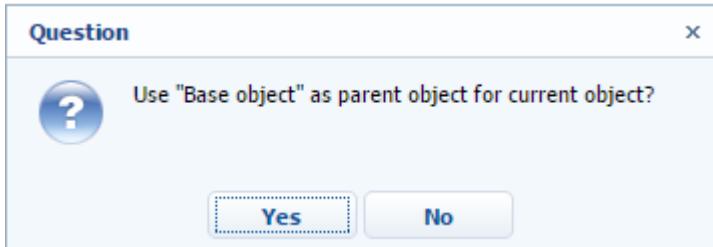


Fig. 7. Columns inherited in the entity structure

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

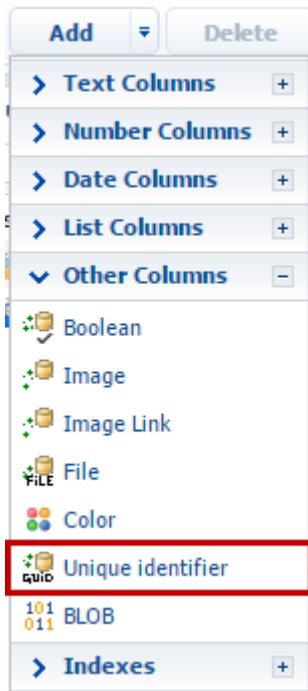
Save the object schema to preserve the metadata changes. Publish the schema to create the corresponding table in the database and make the changes available to Creatio users.

Adding a custom column to an entity

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

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

Fig. 8. Adding an Id column



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

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

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

Fig. 9. Adding the Id column

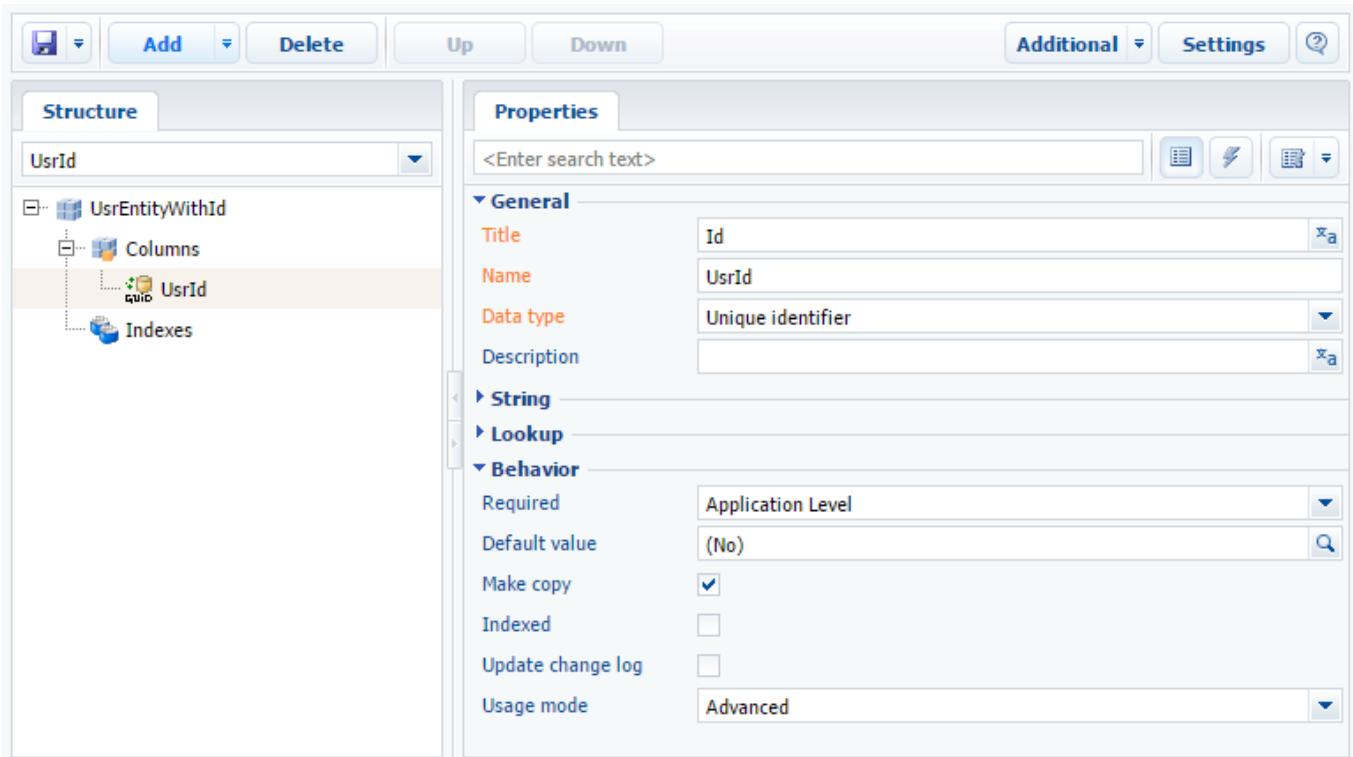
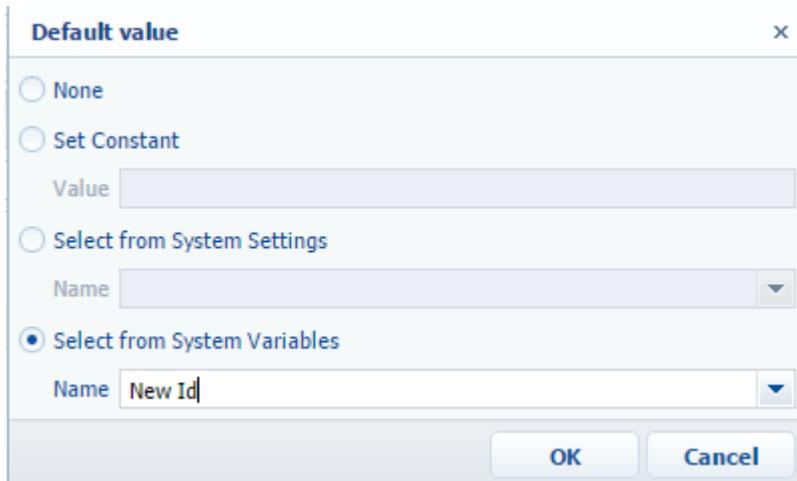


Fig. 10. Setting the default value



Save the schema after setting values for all required attributes.

Adding indexes to the object

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

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

You can create a composite index in a following way:

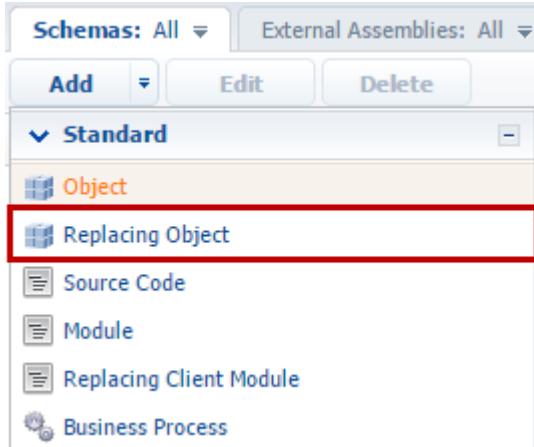
1. Select [Add] > [Index] in the context menu of the [Indexes] element. You can specify a custom name for the index or select the [Generate Name Automatically] option. After that, the unique index name will be generated by the system.
2. To implement an integrity constraint for the columns of the index,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

Creating the replacing object schema

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

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

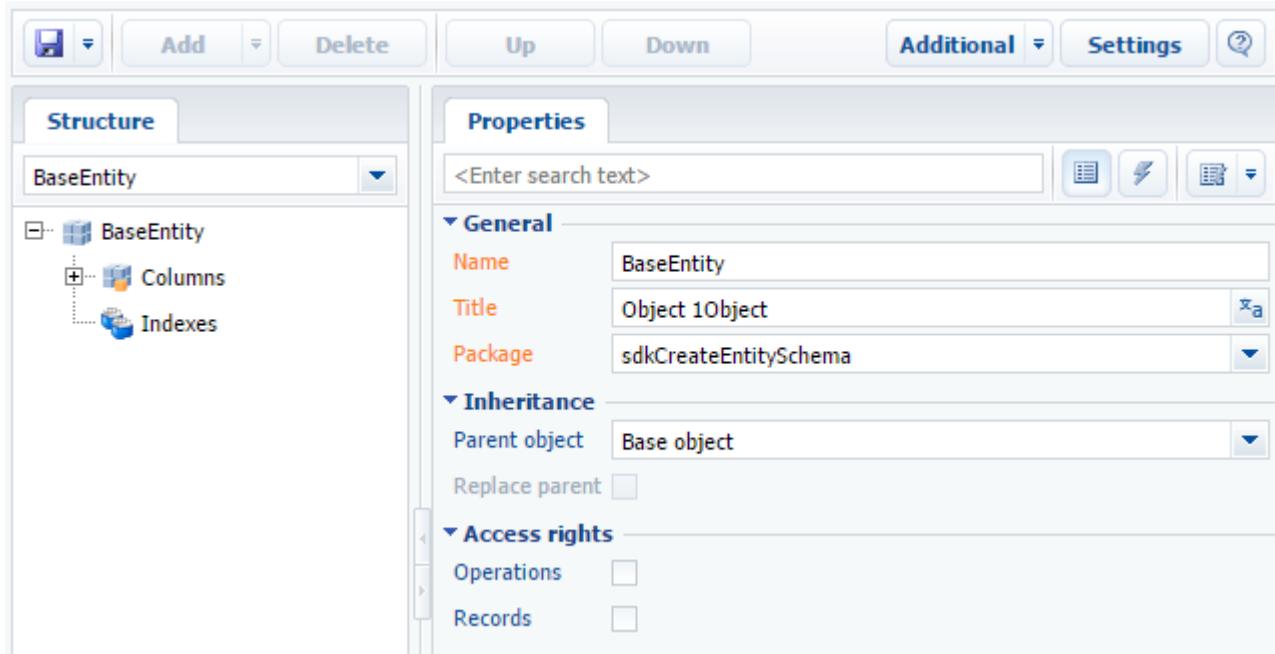
Fig. 11. Command for creating a replacing object schema



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

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

Fig. 12. Main properties of the replacing object schema



After implementing the changes, publish the replacing object schema.

Saving and publishing objects

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

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

designer.

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

Creating a custom client module schema

Beginner

Easy

Medium

Advanced

Introduction

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

The following client module types are available in Creatio:

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

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

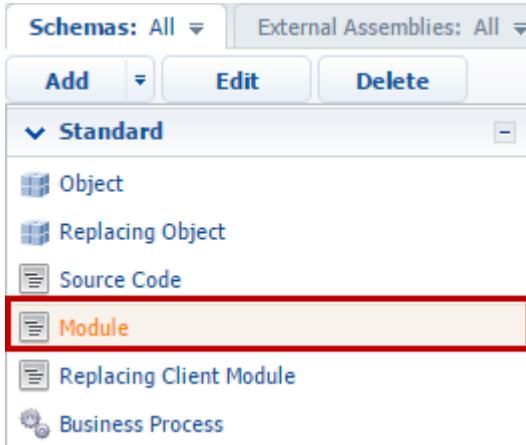
Creating a new schema of the non-visual module

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

To create a non-visual module schema:

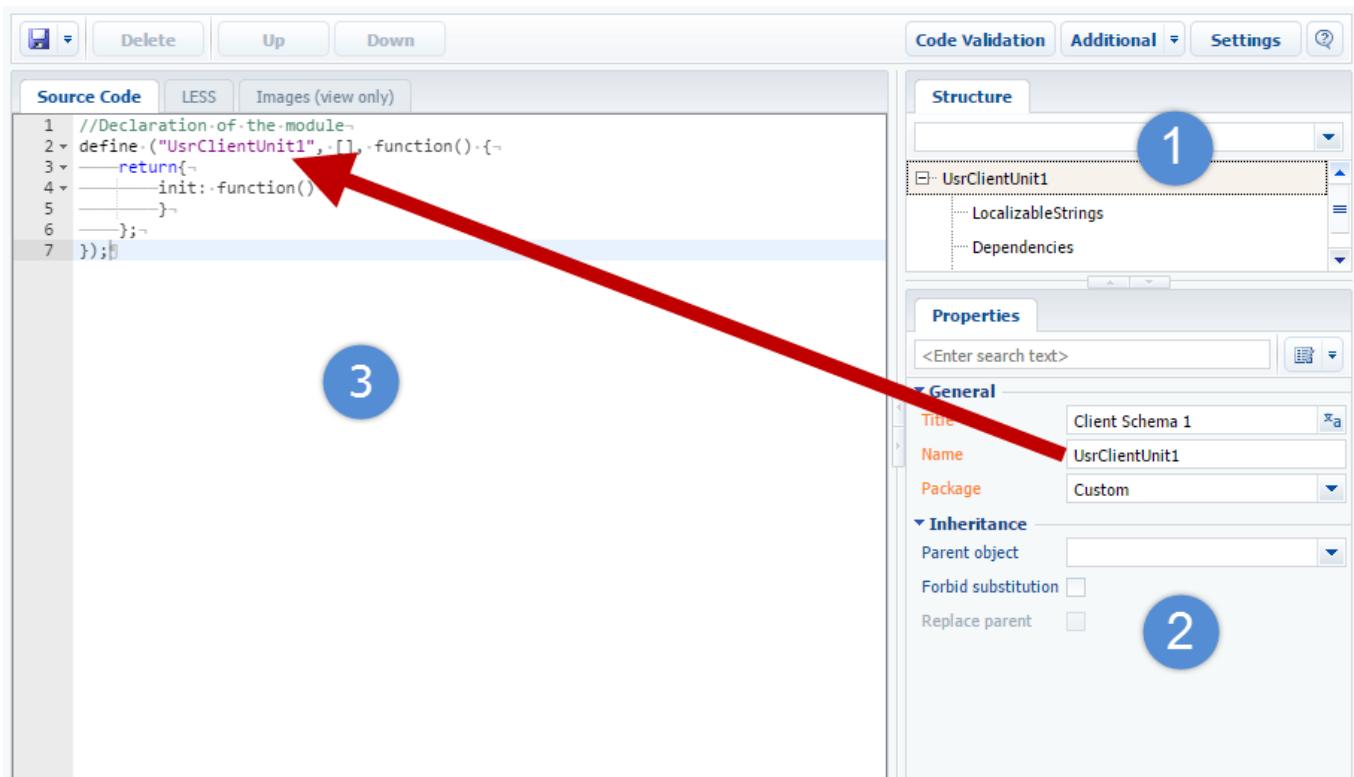
1. Go to the **[Configuration]** section and select custom package to add new schema.
2. On the **[Schemas]** tab, select **[Add] – [Module]** (Fig. 1).

Fig. 1. Adding new module schema



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

Fig. 2. Client module designer



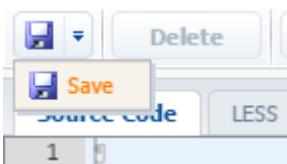
The main properties of the module schema are:

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

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

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

Fig. 3. Saving the client module schema



Creating a new view model schema.

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

To create a schema of the visual module:

1. Go to the **[Configuration]** section and select custom package to add a new schema.
2. Select one of the commands to add the schema from the “Additional” commands menu on the [Schemas] tab (Fig. 4):

Fig. 4. Commands for adding view model schemas

The screenshot shows the 'Schemas' section of the Creatio interface. At the top, there are dropdown menus for 'Schemas: All' and 'External Assemblies: All'. Below them are buttons for 'Add', 'Edit', and 'Delete'. The main area is divided into two sections: 'Standard' and 'Additional'. The 'Standard' section contains icons for Object, Replacing Object, Source Code, Module, Replacing Client Module, and Business Process. The 'Additional' section, which is expanded, contains four items: Schema of the Edit Page View Model, Schema of the Section View Model, Schema of the Detail View Model with List, and Schema of the Detail View Model with Fields. The last three items are highlighted with a red box.

You can add the following types of visual module schemas:

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

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

(Fig. 5). custom view model schemas designer

The screenshot shows the 'Source Code' tab of the 'Custom View Model Schemas Designer'. It displays a code editor with the following content:

```

1 define("UsrClientUnit2",[],function(){~
2   return{~
3     entitySchemaName:"ExampleEntity",~
4     mixins:{},~
5     attributes:{},~
6     messages:{},~
7     rules:{},~
8     modules:/**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/~,~
9     diff:/**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/~,~
10    };~
11 });

```

A red arrow points from the 'entitySchemaName' line in the code to a blue circle labeled '1' in the 'Structure' panel. The 'Structure' panel shows a tree view with 'UsrClientUnit2' as the root node, containing 'LocalizableStrings', 'Dependencies', and 'Images'. A blue circle labeled '2' is in the 'Properties' panel, which contains the following fields:

Title	Client Schema 2
Name	UsrClientUnit2
Package	Custom
Inheritance	
Parent object	
Replace parent	

A blue circle labeled '3' is in the code editor area.

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

mentioned above.

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

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

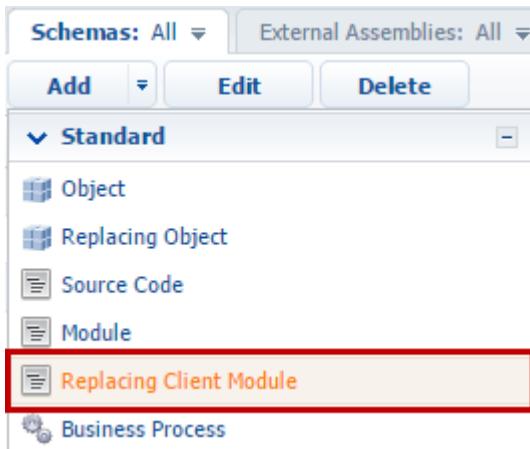
Creating a replacing schema

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

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

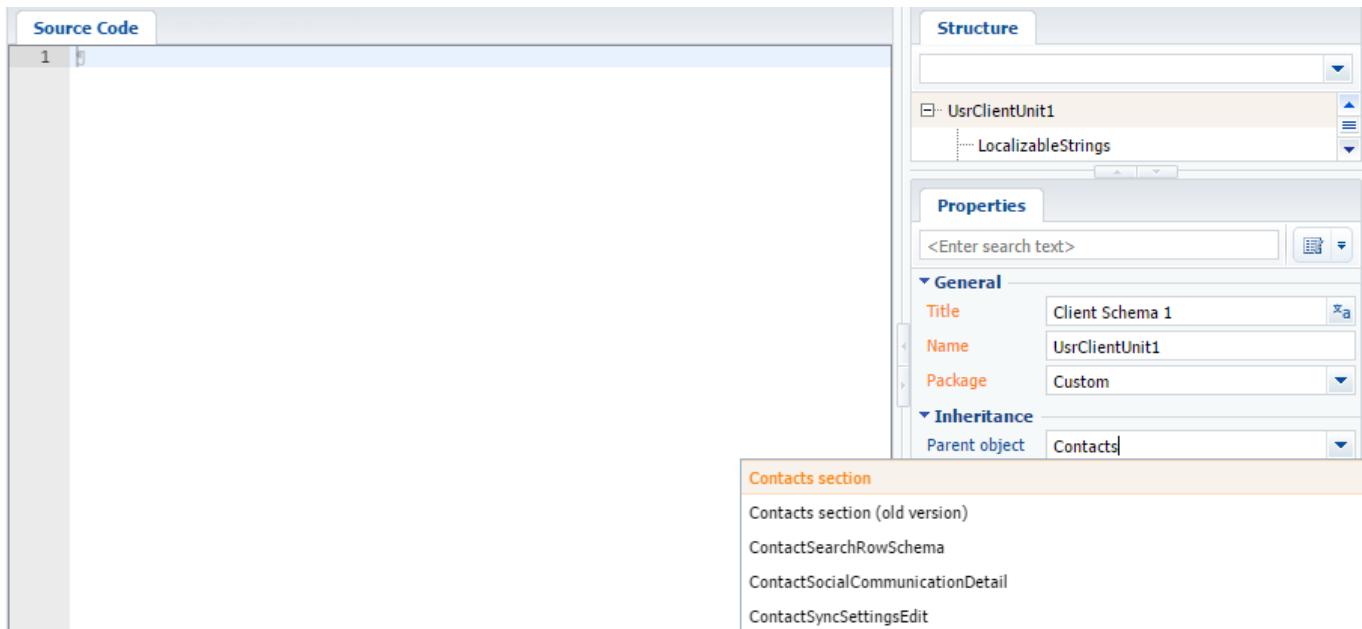
1. Go to the **[Configuration]** section and select custom package to add new module schema.
2. On the [Schemas] tab, select [Add] – [Replacing client module] (Fig. 6).

Fig. 6. Creating a replacing schema



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

(Fig. 7). Creating a replacing schema



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

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

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

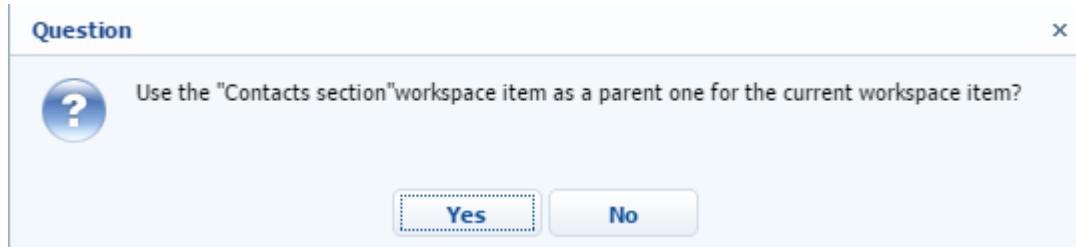
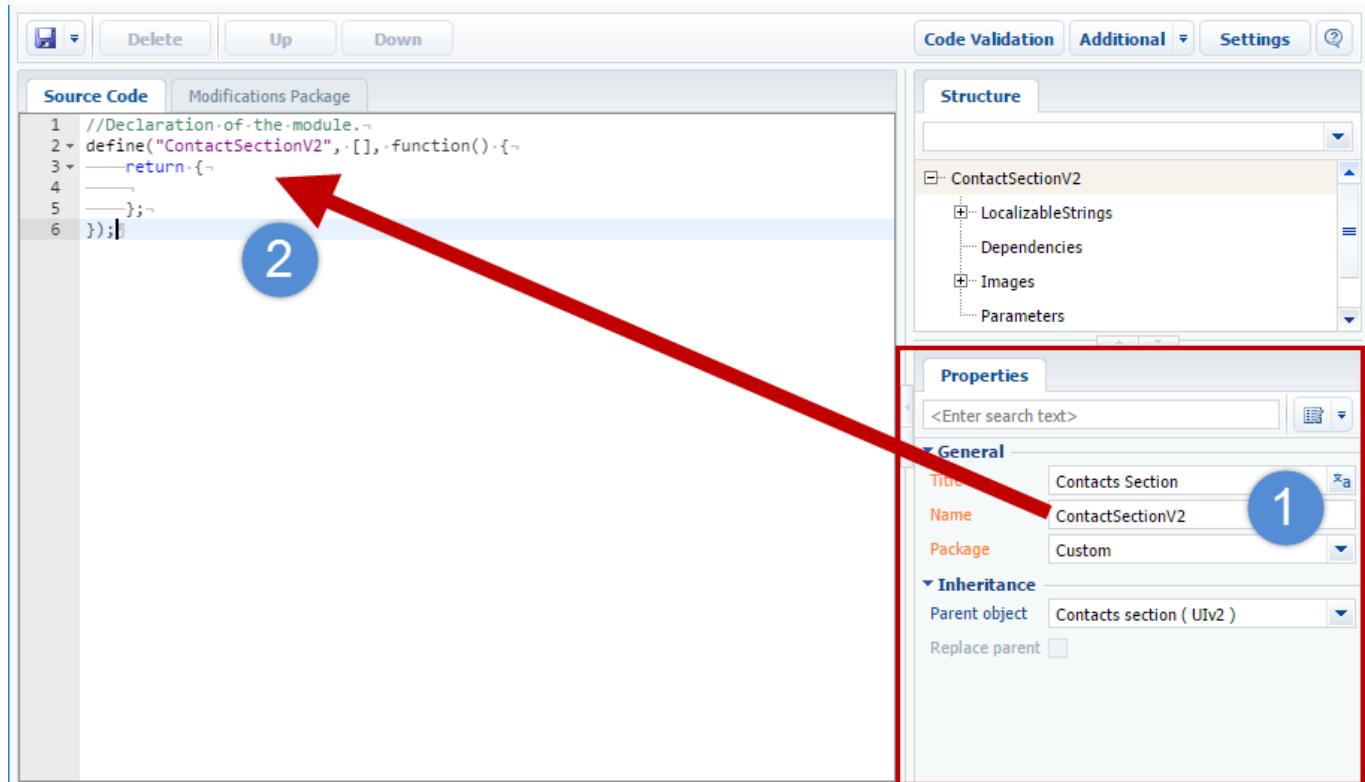


Fig. 9. A replacement client schema in the client schema designer



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

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

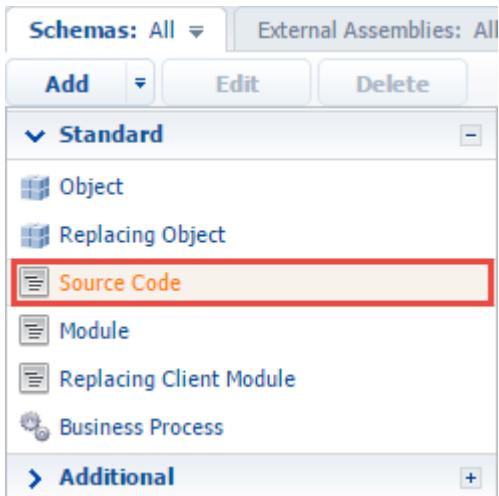
Creating the [Source code] schema

Beginner **Easy** Medium Advanced

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

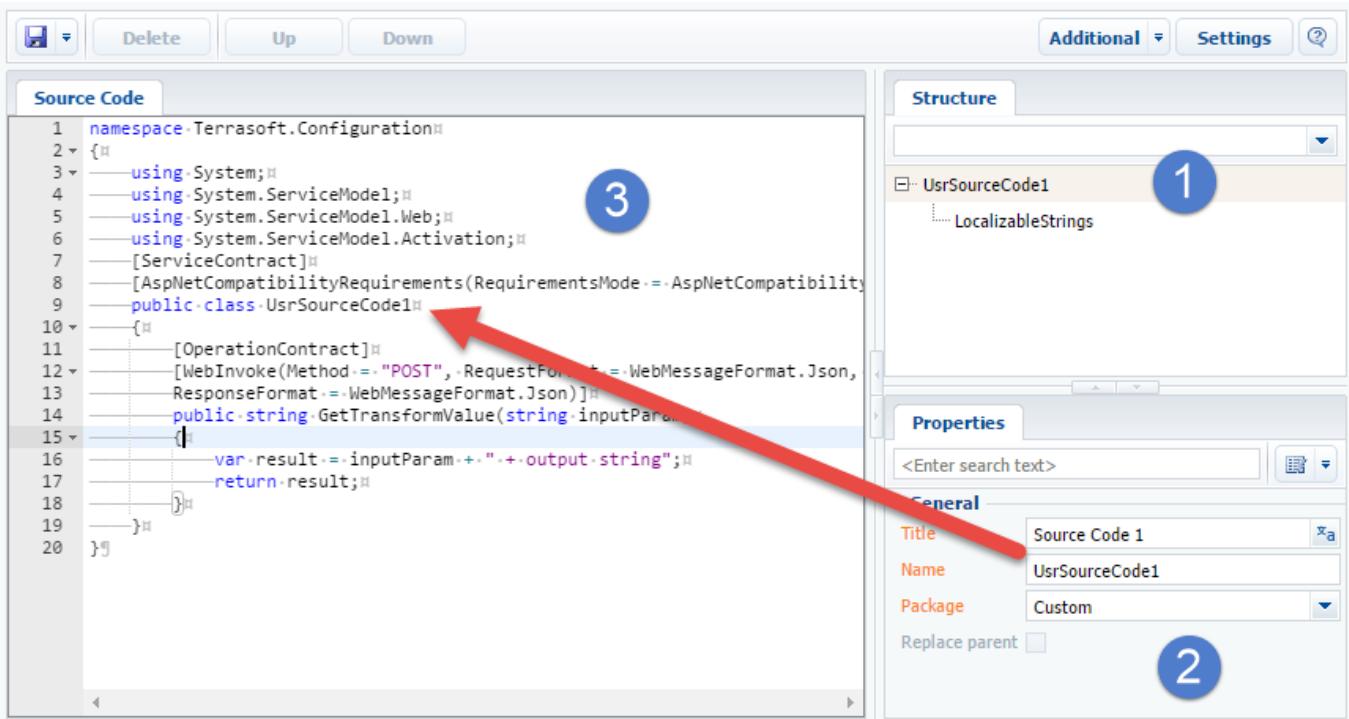
1. Go to the [Configuration] section and select the custom package to add a new schema.
2. On the [Schemas] tab, run the Add > Source Code command (Figure 1).

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



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

Fig. 2. The [Source code] schema designer



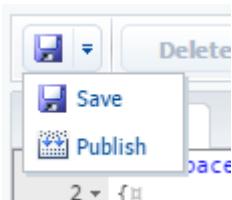
Main [Source code] schema properties:

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

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

5. Publish the schema (Fig. 3):

Fig. 3. Saving and publishing a schema



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

Development tools. Packages

Contents

- **Introduction**
- **Package structure and contents**
- **Package dependencies**
- **Package [Custom]**
- **Creating a package for development**
- **Binding data to packages**

Working with packages

Beginner

Easy

Medium

Advanced

Introduction

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

A Creatio package is a collection of configuration elements that implements particular block of functionality.

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

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

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

- **Creating a package for development**
- [Committing a package to repository](#)
- **Transferring changes using packages export and import**
- [Updating package from repository](#)
- [Installing marketplace applications from a zip archive](#)
- **Transferring changes using packages export and import**
- **Creating a package in the file system development mode**
- **Binding data to packages**

Package structure and contents

Beginner

Easy

Medium

Advanced

General information about packages

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

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

The Creatio packages can be divided into two types:

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

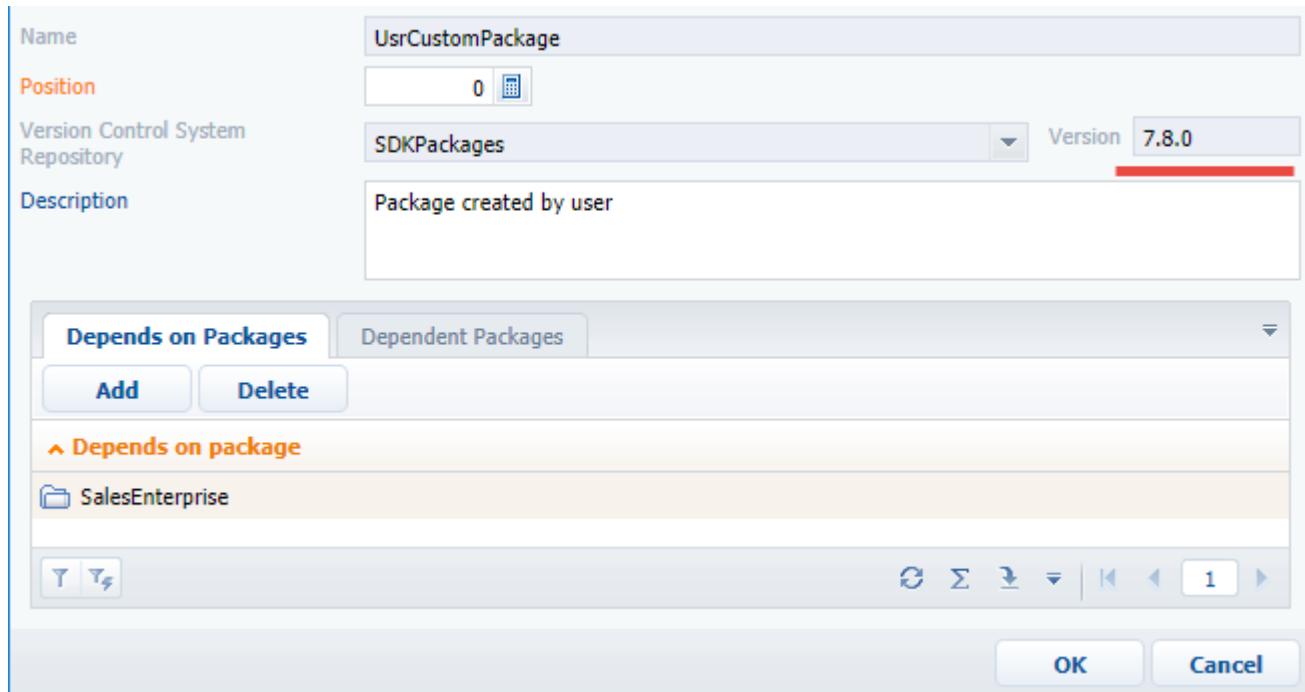
Configuration elements of the pre-installed packages cannot be modified. You can develop additional functionality or modify the existing functionality only via custom packages

Package version

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

The package versioning mechanism is deprecated and is not supported, starting with Creatio 7.9. Therefore, all pre-installed packages have a version no higher than 7.8.

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



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

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

```
        }
    ]
}
```

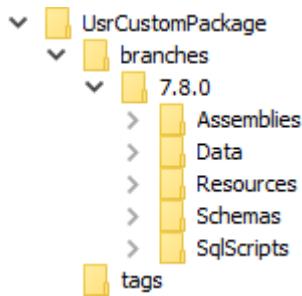
All elements of the package are of the same version as the package itself.

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

Package structure

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

Fig. 3. Package structure in the SVN



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

The structure that takes into account the package versions remained for compatibility with Creatio versions below 7.9.

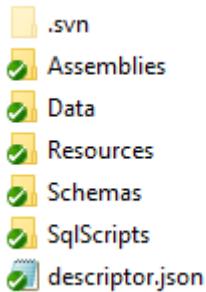
The *tags* directory stores tags. The tags in the version control system represent a "snapshot" of the project at a certain point in time, a static copy of the files required for saving some critical stage of development.

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

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

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

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



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

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

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

Package dependencies

Beginner

Easy

Medium

Advanced

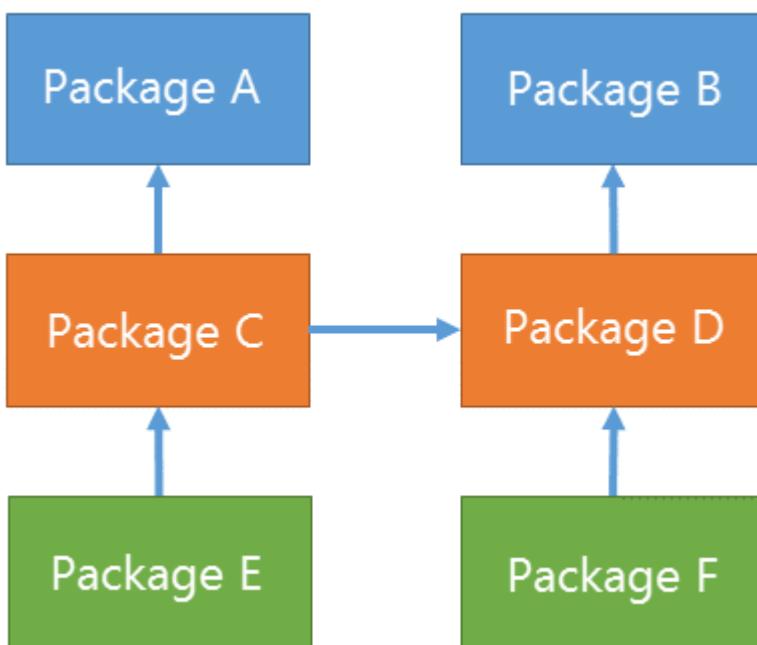
Introduction

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

Dependencies and package hierarchy

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

Fig. 1. Dependencies and package hierarchy

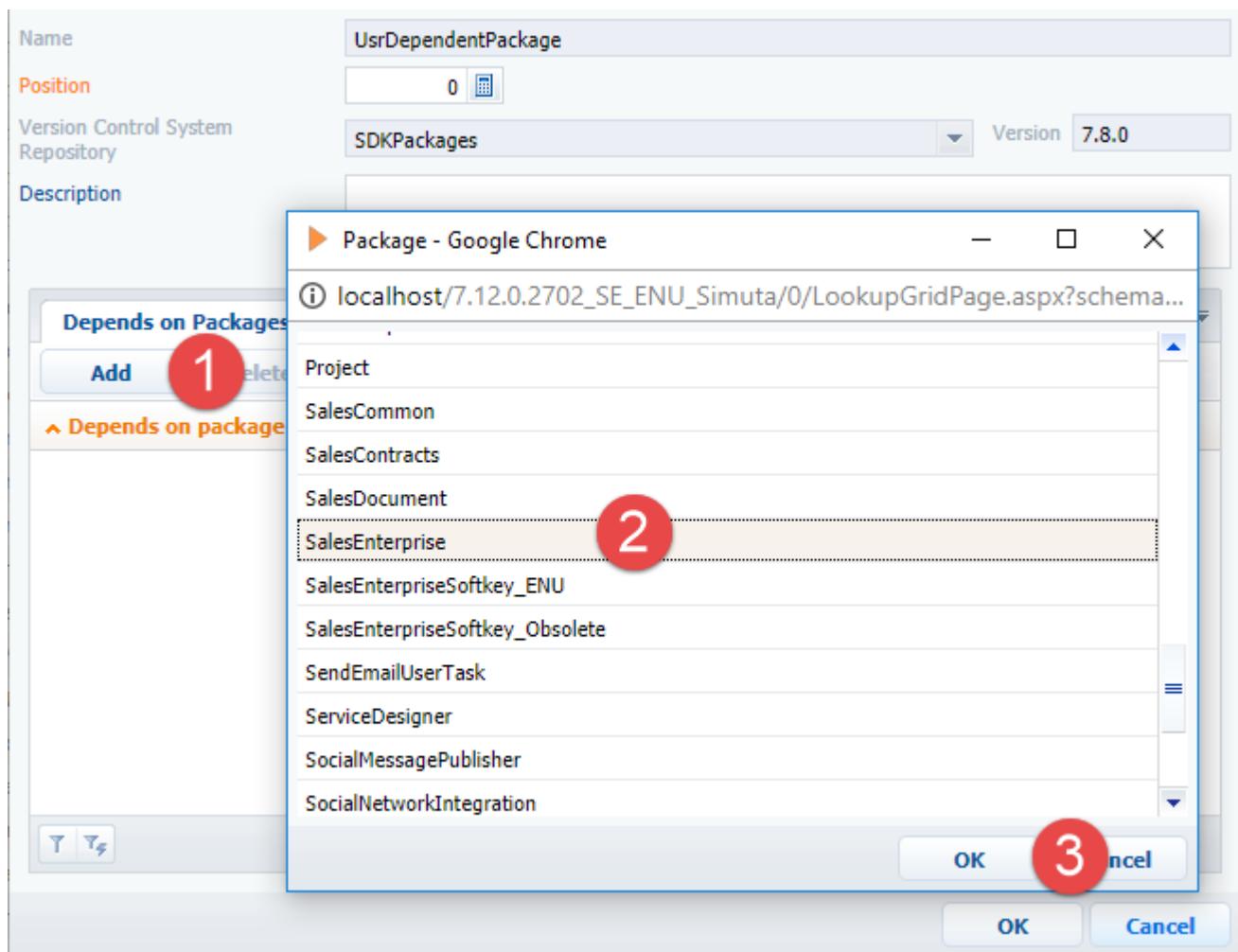


Package dependencies form a hierarchical chain. This means that if you add a package to the dependency of another package, the dependent package will contain all functionality of the added package as well as functionality of all packages that the added package depends on. The closest analogy of the package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

How to add package dependencies

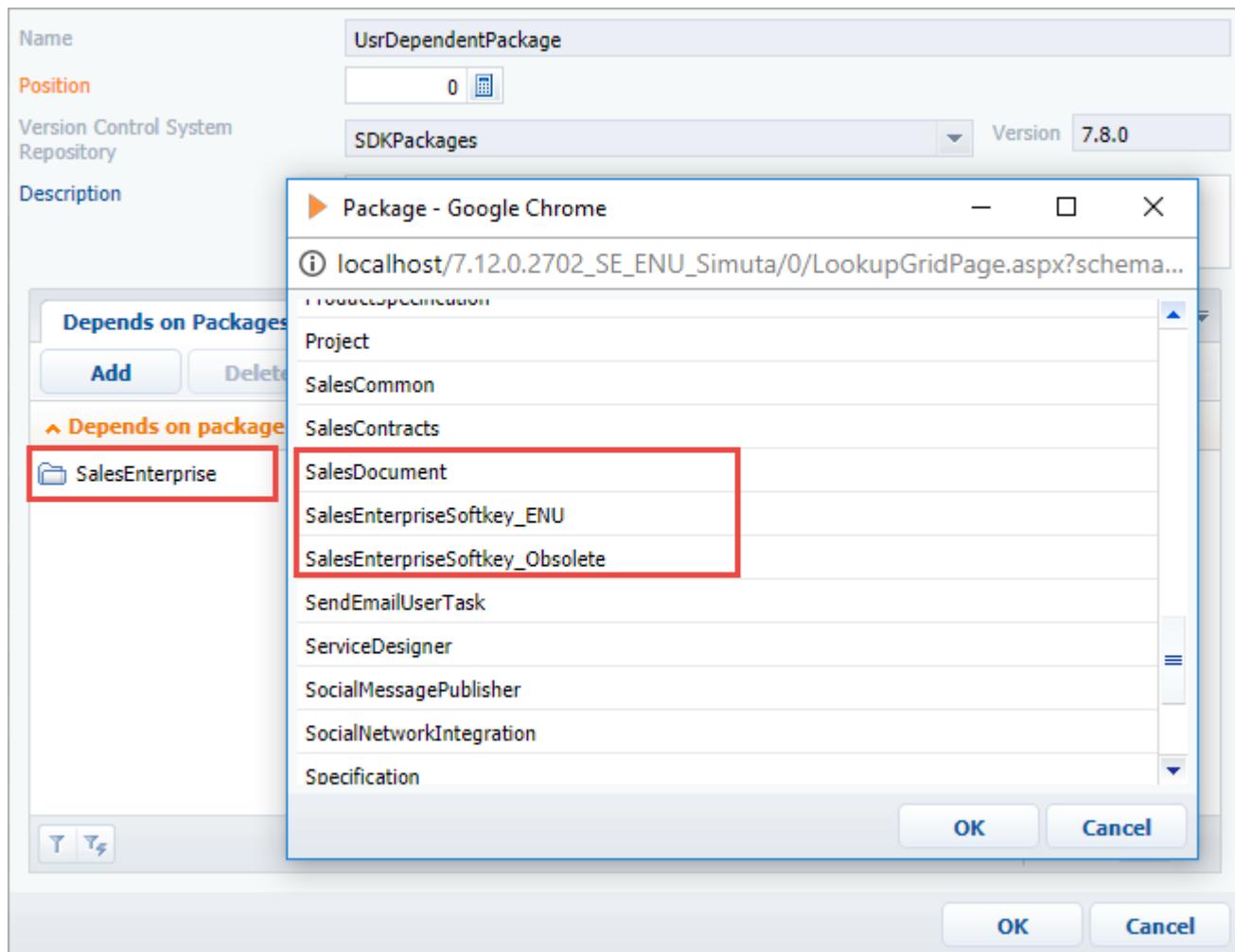
Dependencies can only be added to a custom package, and only after it has been created. To add dependencies, click the [Add] (1) button on the [Depends on packages] (Fig. 2) detail. In the opened dialog of the package lookup, select the required package (2) and click the [OK] button (3).

Fig. 2. Adding dependency to a custom package



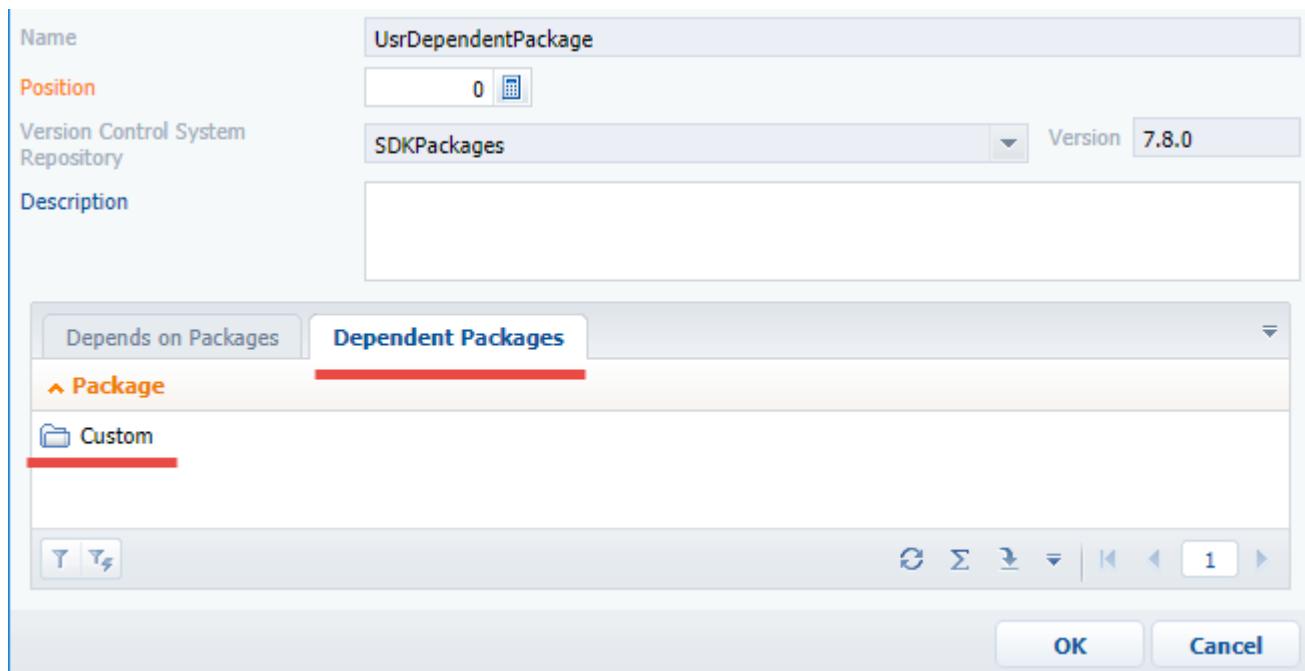
The selected package will be displayed in the list of dependencies of the current package. The packages that have been added to the dependencies are not displayed in the package lookup (Fig. 3).

Fig. 3. Added package dependency



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

Fig. 4. "Dependent Packages" tab



The list of dependencies in the metadata

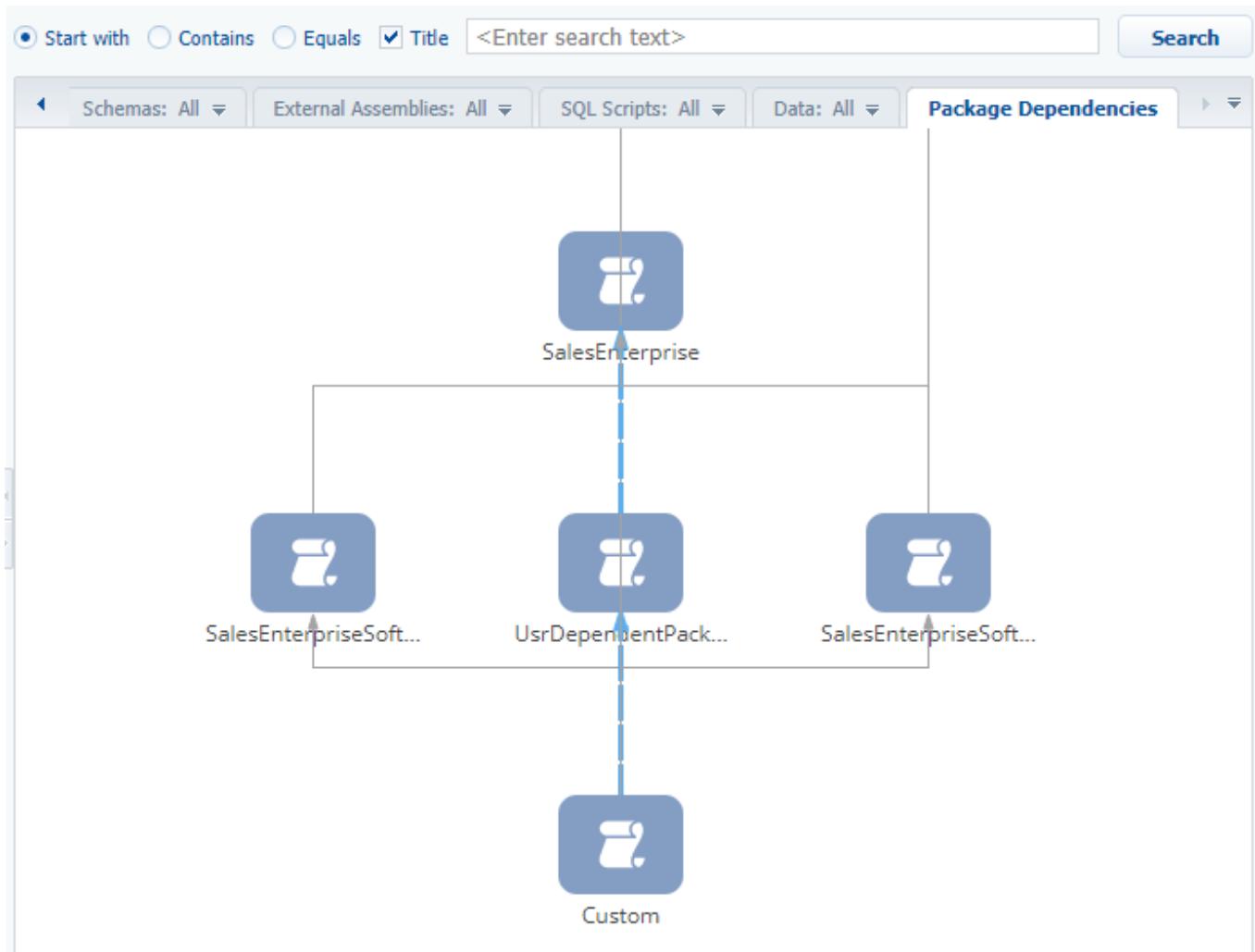
The list of package dependencies is stored in the package metadata in the *DependsOn* property of the object specified in the *descriptor.json* file. The *DependsOn* property is an array of objects that contain the package name, version and unique identifier by which the package can be identified in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

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

Application package hierarchy

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

Fig. 4. A fragment of package dependency hierarchy



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

Primary packages

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

Table 1. Basic application packages

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

Package [Custom]

Beginner

Easy

Medium

Advanced

Introduction

There are two types of Creatio packages:

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

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

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

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

Specifics of the “Custom” package

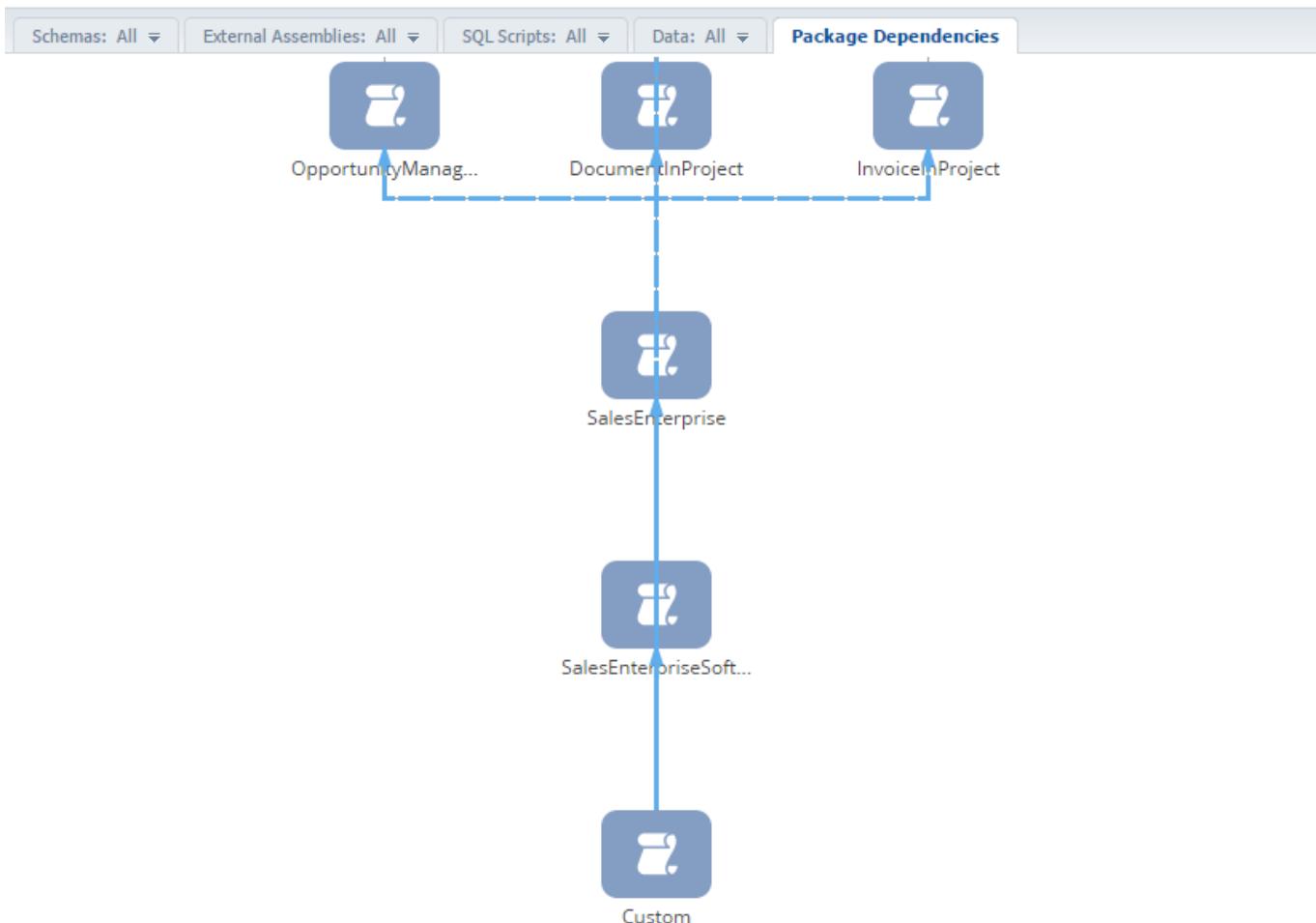
As a pre-installed package, “Custom” cannot be added to the SVN subversion control repository. The schemas can be transferred from the “Custom” package only by using the **export/import function**.

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

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

Custom packages cannot depend on the [Custom] package.

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



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

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

Recommendations

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

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

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

Creating a package for development

Beginner

Easy

Medium

Advanced

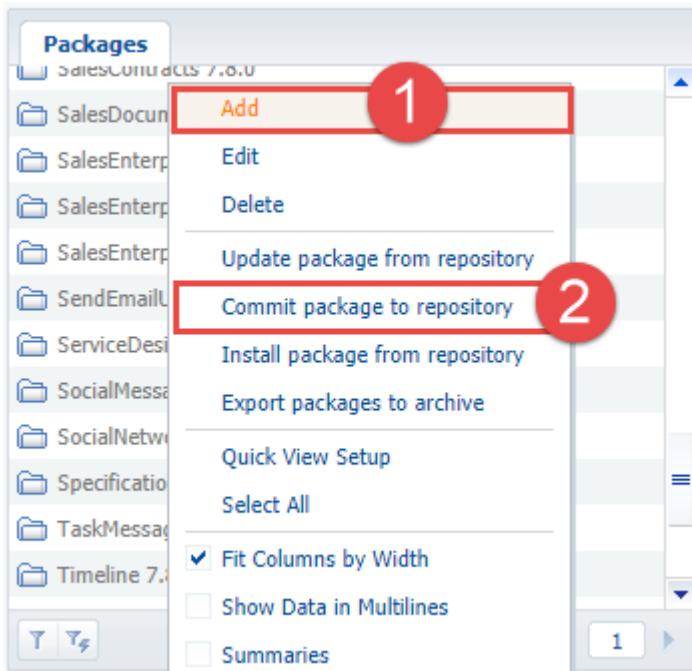
Introduction

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

How to create a custom package

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

Fig. 1. How to add a new package



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

Fig. 2. Package mini-page

Name	NewPackage
Position	0
Version Control System Repository	SDKPackages
Description	
<input type="button" value="Depends on Packages"/> <input type="button" value="Dependent Packages"/>	
<input type="button" value="Add"/> <input type="button" value="Delete"/>	
Depends on package	
<input type="button" value="OK"/> <input type="button" value="Cancel"/>	

Main fields of the package mini-page:

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

not marked as active will not appear in the drop-down list of available storages. This is a required field.

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

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

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

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

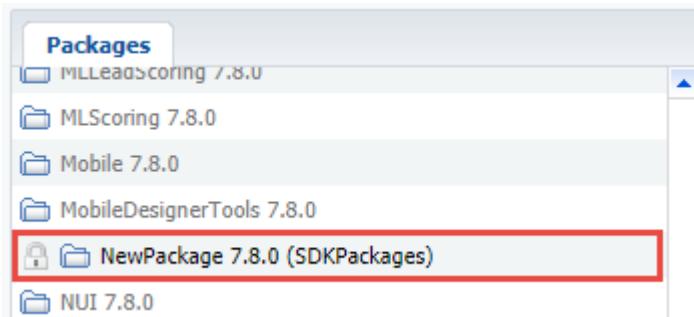
The contents of the key package mini-page fields pack will be saved in its metadata:

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

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

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

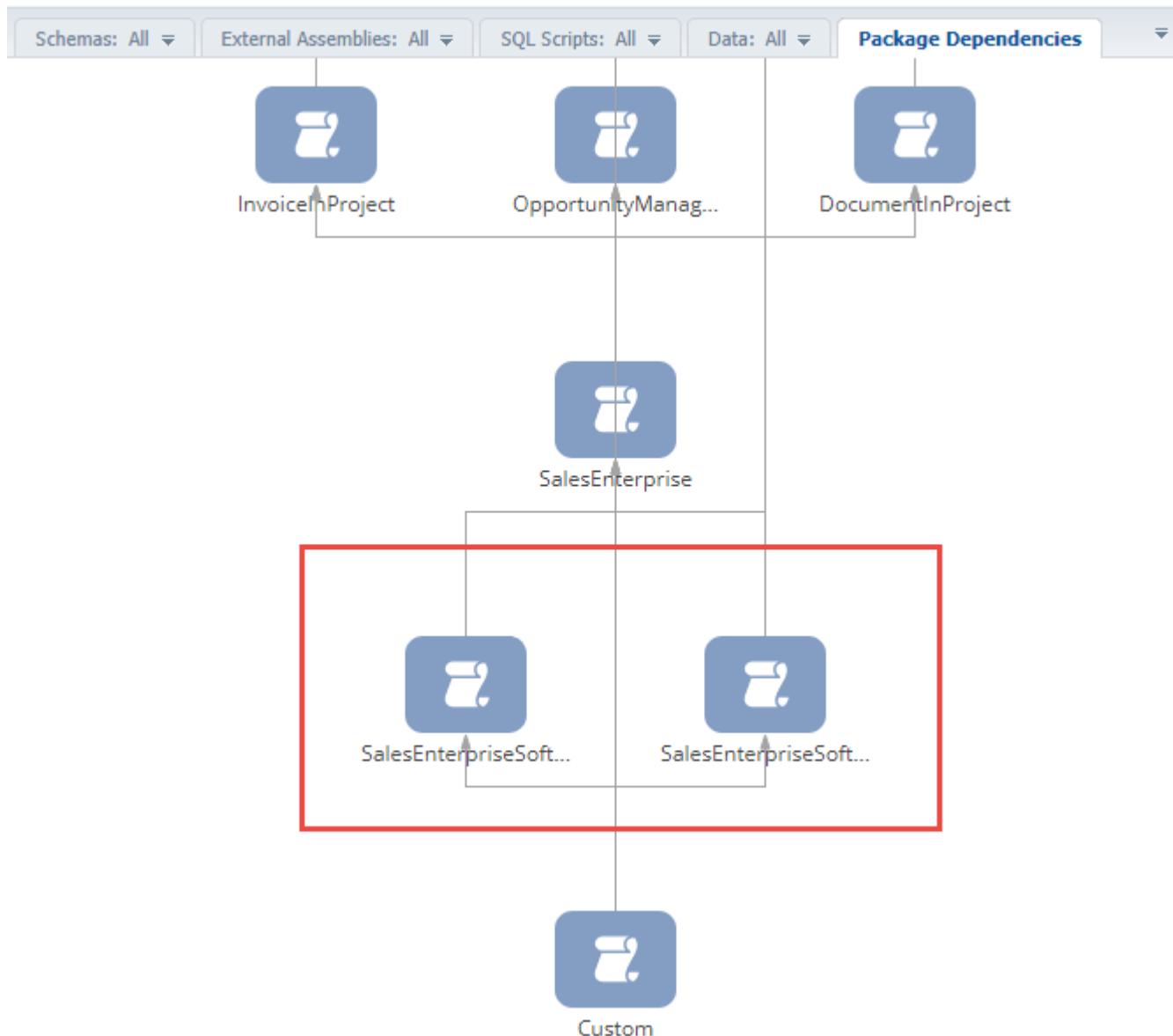
Fig. 3. New package on the [Packages] tab



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

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

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



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

Binding data to packages

Beginner

Easy

Medium

Advanced

Introduction

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

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

Case description

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

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

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

Name	Package	Schema
SysDetail_DetailManager_b78e...	sdkBookExample	SysDetail
SysImage_f8d3f0121ed2433a9...	sdkBookExample	SysImage
SysModule_SectionManager_8...	sdkBookExample	SysModule
SysModuleEdit_SysModuleEdit...	sdkBookExample	SysModuleEdit
SysModuleEntity_SysModuleEn...	sdkBookExample	SysModuleEntity
SysModuleInWorkplace_Sectio...	sdkBookExample	SysModuleInWorkplace

Source code

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

Case implementation algorithm

1. Adding a new [Books] section

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

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

Fig. 2. The [Books] section properties

Select basic properties for section:

Title* Books

Code* UsrBook

Menu icon 

Page settings:

- One page for all records
- Multiple pages

Fig. 3. Record edit page properties

Books: Page

SAVE CANCEL < SECTION PAGE BUSINESS RULES CASES BUSINESS PROCESSES >

Page elements

Books

- New column
- Boolean
- Date
- 0.5 Decimal
- 123 Integer
- Lookup
- String

T ISBN	T Name *
Author *	Description
Publisher *	
0.5 Price	

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

Table 1. Edit page column properties of section records

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

Price

UsrPrice

Decimal

2. Adding needed records to the section

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

Fig. 4. Section records

ISBN	Author	Publisher	Price
978-0596805524	David Flanagan	Apress	33.89
978-1484230176	Andrew Troelsen	Apress	56.99

3. Binding contact data to the package

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

1. [Name] – “ContactsInBooks”
2. [Object] – “Contact”
3. [Installation type] – “Installation”. Possible installation types are described in the **“The [Configuration] section. The [Data] tab” article**.
4. [Columns] – select only the populated columns. It is required to select the [Id] column.
5. Data filtering – filter the needed data, for example, by contact name.

Fig. 5. Page for binding contact data to package

Full name	Owner	Data entry compliance
Andrew Troelsen	Supervisor	10
David Flanagan	Supervisor	10

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

4. Binding account data to the package

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

1. [Name] – “AccountsInBooks”
2. [Object] – “Account”
3. [Installation type] – “Installation”
4. [Columns] – select only the populated columns. It is required to select the [Id] column.
5. Data filtering – filter the needed data, for example, by account identifier. You can determine the identifier from the browser address bar by opening the needed record edit page (Fig. 7)

Fig. 6. Page for binding account data to package

Name	Owner	Data entry compliance
Apress	Supervisor	10

Fig. 7. Determining the account identifier

5. Binding custom package data to the package

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

1. [Name] – “Books”
2. [Object] – “Books”
3. [Installation type] – “Installation”
4. [Columns] – select only the populated columns. It is required to select the [Id] column.
5. Data filtering – filter the needed data. If a section contains only two records you can avoid using the filter (Fig.8).

Fig. 8. The page of binding custom section data

Properties

Bound Data

Name	Books																								
Object	Books																								
Installation type	Installation																								
Columns	<input type="button" value="Add"/> <input type="button" value="Edit"/> <input type="button" value="Delete"/> <table border="1"> <thead> <tr> <th>Name</th> <th>Forced up...</th> <th>Key</th> </tr> </thead> <tbody> <tr> <td>Id</td> <td><input type="checkbox"/></td> <td><input checked="" type="checkbox"/></td> </tr> <tr> <td>Author</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Description</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>ISBN</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Name</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Price</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> <tr> <td>Publisher</td> <td><input type="checkbox"/></td> <td><input type="checkbox"/></td> </tr> </tbody> </table> <input type="checkbox"/> AND <input type="button" value="Add new condition"/>	Name	Forced up...	Key	Id	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Author	<input type="checkbox"/>	<input type="checkbox"/>	Description	<input type="checkbox"/>	<input type="checkbox"/>	ISBN	<input type="checkbox"/>	<input type="checkbox"/>	Name	<input type="checkbox"/>	<input type="checkbox"/>	Price	<input type="checkbox"/>	<input type="checkbox"/>	Publisher	<input type="checkbox"/>	<input type="checkbox"/>
Name	Forced up...	Key																							
Id	<input type="checkbox"/>	<input checked="" type="checkbox"/>																							
Author	<input type="checkbox"/>	<input type="checkbox"/>																							
Description	<input type="checkbox"/>	<input type="checkbox"/>																							
ISBN	<input type="checkbox"/>	<input type="checkbox"/>																							
Name	<input type="checkbox"/>	<input type="checkbox"/>																							
Price	<input type="checkbox"/>	<input type="checkbox"/>																							
Publisher	<input type="checkbox"/>	<input type="checkbox"/>																							

Display data Records total: 2

Name	ISBN	Author	Publisher	Description	Price
JavaScript: The Definitive Guide: Activate Your Web Pages	978-0596805524	David Flanagan	Apress	Since 1996, JavaScript: The Definitive Guide has been the bible for JavaScript programmers—a programmer's guide and comprehensive reference to the core language and to the client-side JavaScript APIs defined by web browsers	33.89
Pro C# 7: With .NET and .NET Core	978-1484230176	Andrew Troelsen	Apress	Dive in and discover why Pro C# has been a favorite of C# developers worldwide for over 15 years.	56.99

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

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

Start with Contains Equals Title <Enter search text>

Schemas: All External Assemblies: All SQL Scripts: All Data: All Packag

Name	Package	Schema
AccountsInBooks	sdkBookExample	Account
Books	sdkBookExample	UsrBook
ContactsInBooks	sdkBookExample	Contact
SysDetail_DetailManager_b78e...	sdkBookExample	SysDetail
SysImage_f8d3f0121ed2433a9...	sdkBookExample	SysImage
SysModule_SectionManager_8...	sdkBookExample	SysModule
SysModuleEdit_SysModuleEdit...	sdkBookExample	SysModuleEdit
SysModuleEntity_SysModuleEn...	sdkBookExample	SysModuleEntity
SysModuleInWorkplace_Sectio...	sdkBookExample	SysModuleInWorkplace

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

Version control system. Built-in IDE

Contents

- **Committing a package to repository**
- **Installing packages from repository**
- **Updating package from repository**

Committing a package to repository

Beginner

Easy

Medium

Advanced

Introduction

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

Packages are committed manually. Modifications of other configuration packages are not committed.

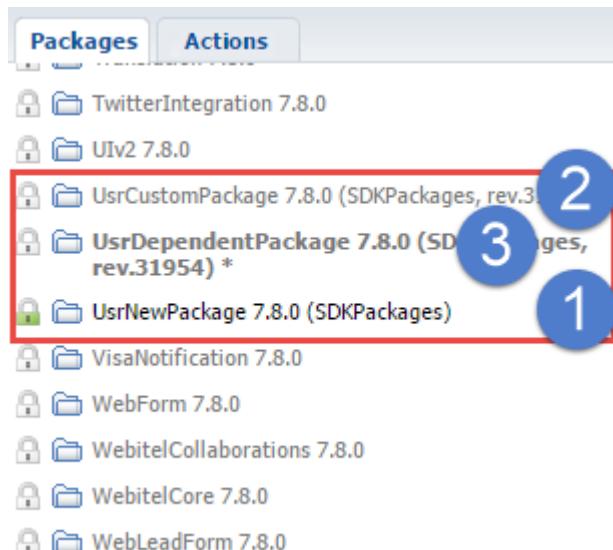
Package committing is required when:

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

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

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

Fig. 1. Package display



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

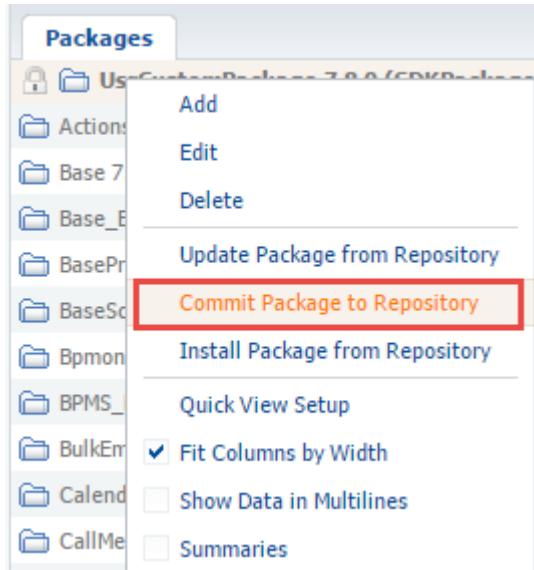
If an element was deleted from a package (for example, schema or SQL script), then those modifications won't affect the package display.

Committing a package to storage

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

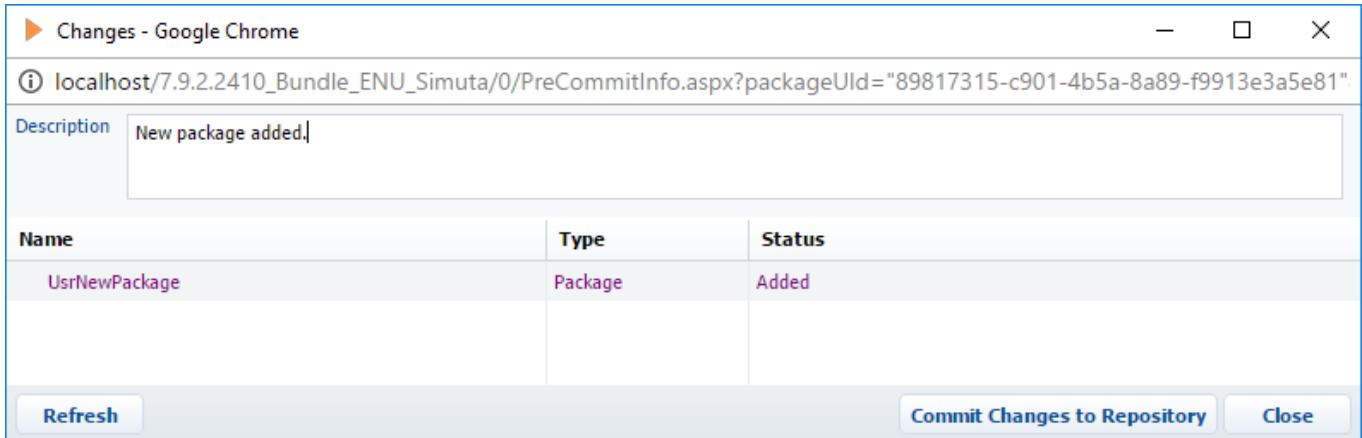
When the **file system development mode is enabled**, the SVN integration mechanism is turned off. Therefore, the [Commit package to repository] action is unavailable.

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



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

Fig. 4. The [Changes] window



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

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

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

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

Installing packages from repository

Beginner

Easy

Medium

Advanced

Introduction

Installing a package from the repository is adding the package and all its dependencies from the version control

system repository (SVN) to Creatio.

Package installation is required when:

- Multiple developers work on the package functionality.
- Changes are transferred ('**Transferring changes between the working environments' in the on-line documentation**) between applications.

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

Package installation sequence

Important!

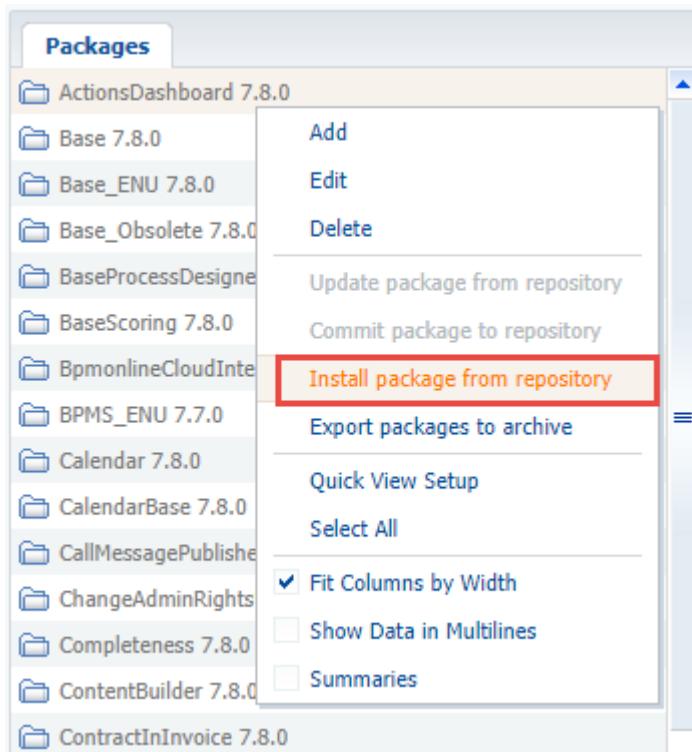
Before updating the application via SVN, you must back up the database. If the application is deployed in the cloud, you should contact support.

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

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

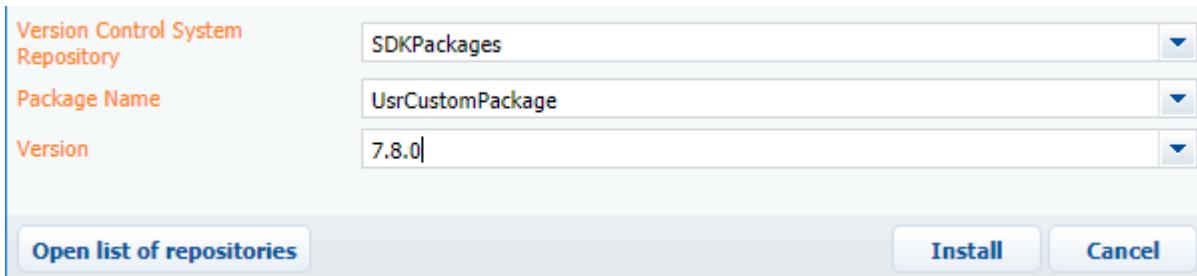
To install the package from the repository, go to the [Configuration] section, right-click the [Packages] tab and select the [Install Package from Repository] option (Fig. 1).

Fig. 1. The [Packages] tab context menu



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

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



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

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

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

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

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

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

Changes in package hierarchy

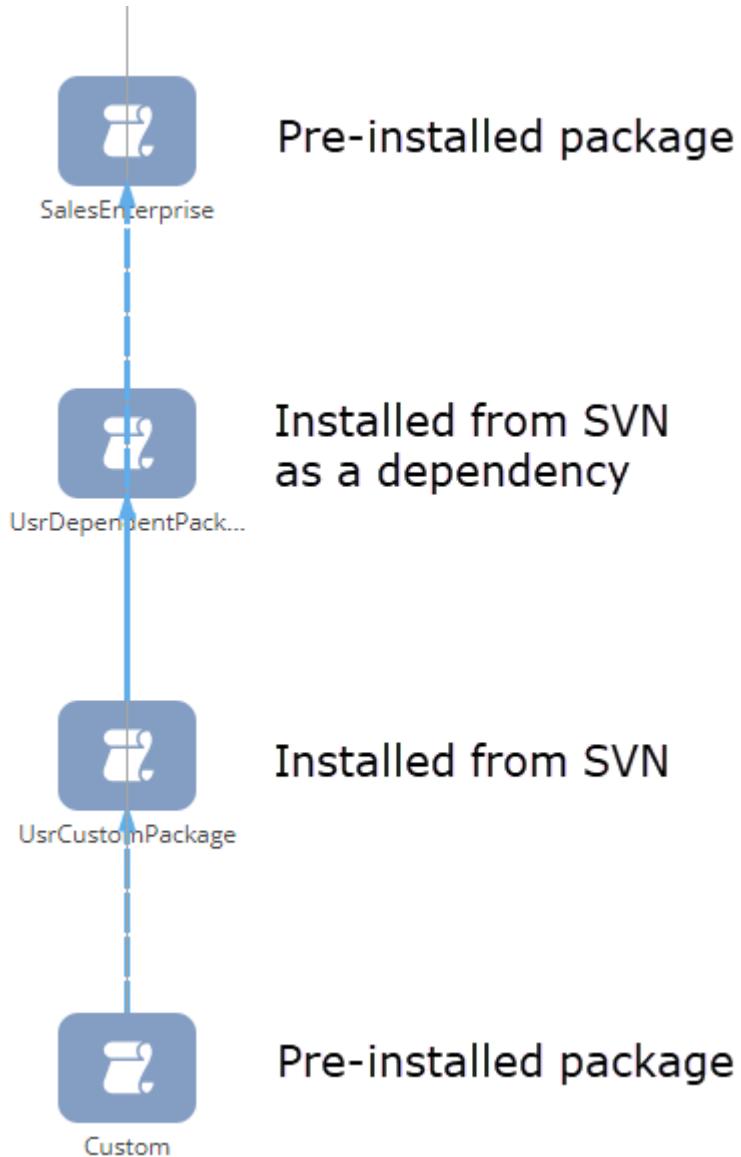
When you install a custom package, the system checks its dependencies and optionally installs or upgrades all the packages the current package depends on. For example, when you install the [UsrCustomPackage] package from the repository, the [UsrDependentPackage] package dependency will also be installed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

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

Name	Type	Status
+... UsrCustomPackage	Package	Added
UsrDependentPackage	Package	Added

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

Fig. 4. New application package hierarchy



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

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

```
{  
  "Descriptor": {  
    "UIId": "8bc92579-92ee-4ff2-8d44-1ca61542aa1b",  
    "PackageVersion": "7.8.0",  
    "Name": "UsrCustomPackage",  
    "ModifiedOnUtc": "\/Date(1522671879000) \/",  
    "Maintainer": "Customer",  
    "Description": "Package created by user",  
    "DependsOn": [  
      {  
        "UIId": "51b3ed42-678c-4da3-bd16-8596b95c0546",  
        "PackageVersion": "7.8.0",  
        "Name": "UsrDependentPackage"  
      },  
      {  
        "UIId": "e14dcfb1-e53c-4439-a876-af7f97083ed9",  
        "PackageVersion": "7.8.0",  
        "Name": "SalesEnterprise"  
      }  
    ]  
  }  
}
```

```
        }  
    ]  
}  
}
```

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

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

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

If the workspace is missing any pre-installed dependency packages that were installed from zip files, the package installation will fail.

You must first install the packages on which the installed custom package or its dependencies depend on.

Updating package from repository

Beginner

Easy

Medium

Advanced

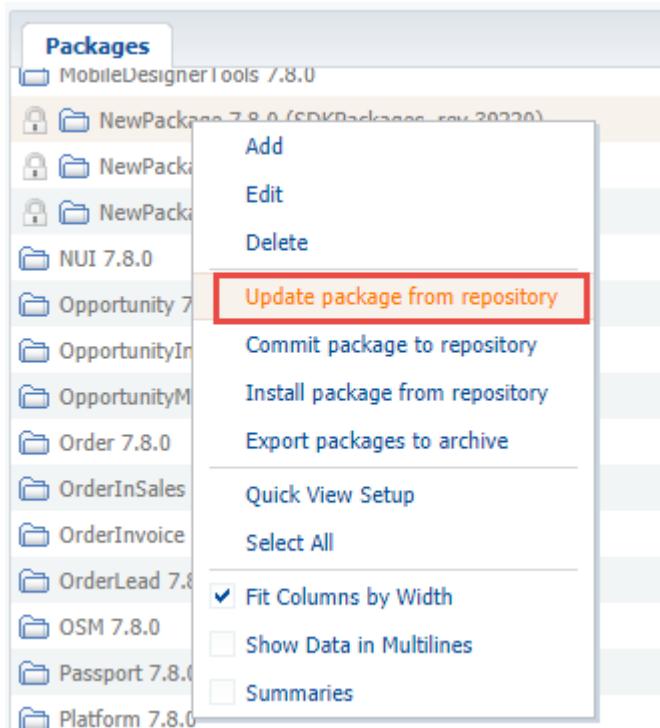
Introduction

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

The sequence of package update from SVN

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

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



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

The system will detect all dependencies of the installed package specified in its metadata in the *DependsOn*

property. The metadata and the package properties are described in detail in the "**Package structure and contents**" article.

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

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

Delivery tools. Built-in IDE

Contents

- **Transferring changes using packages export and import**
- **Transferring changes using schema export and import**
- **Transferring changes using SVN**

Transferring changes using packages export and import

Beginner

Easy

Medium

Advanced

Introduction

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

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

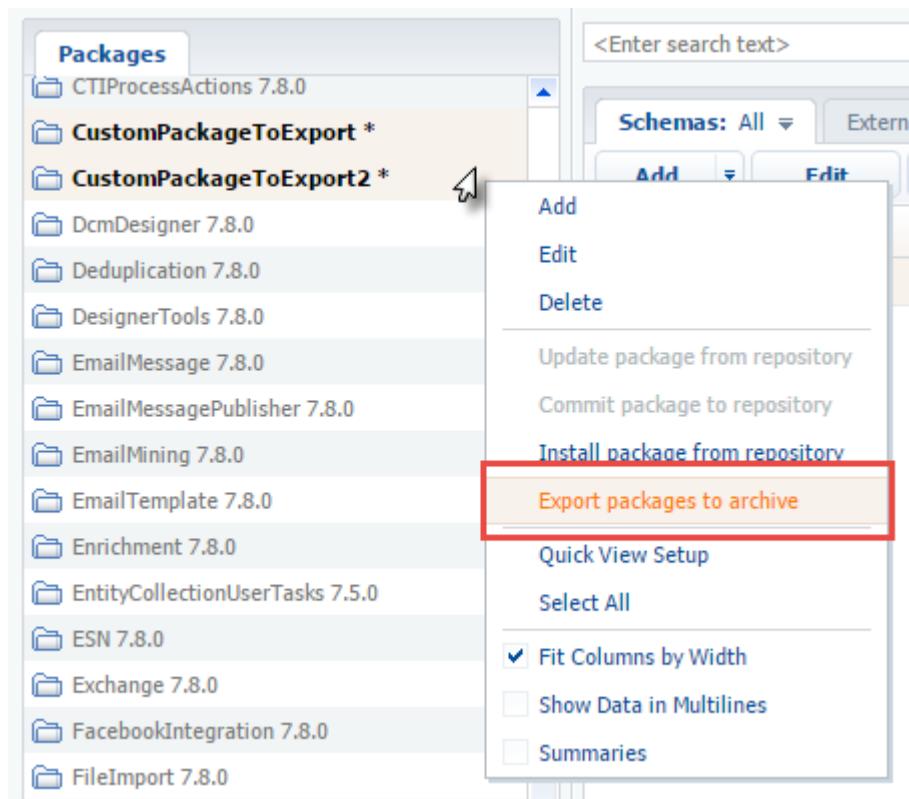
The [Custom] package cannot be transferred between applications. Learn more about this package in the "**Package [Custom]**" article.

Exporting packages

To install packages from the application interface:

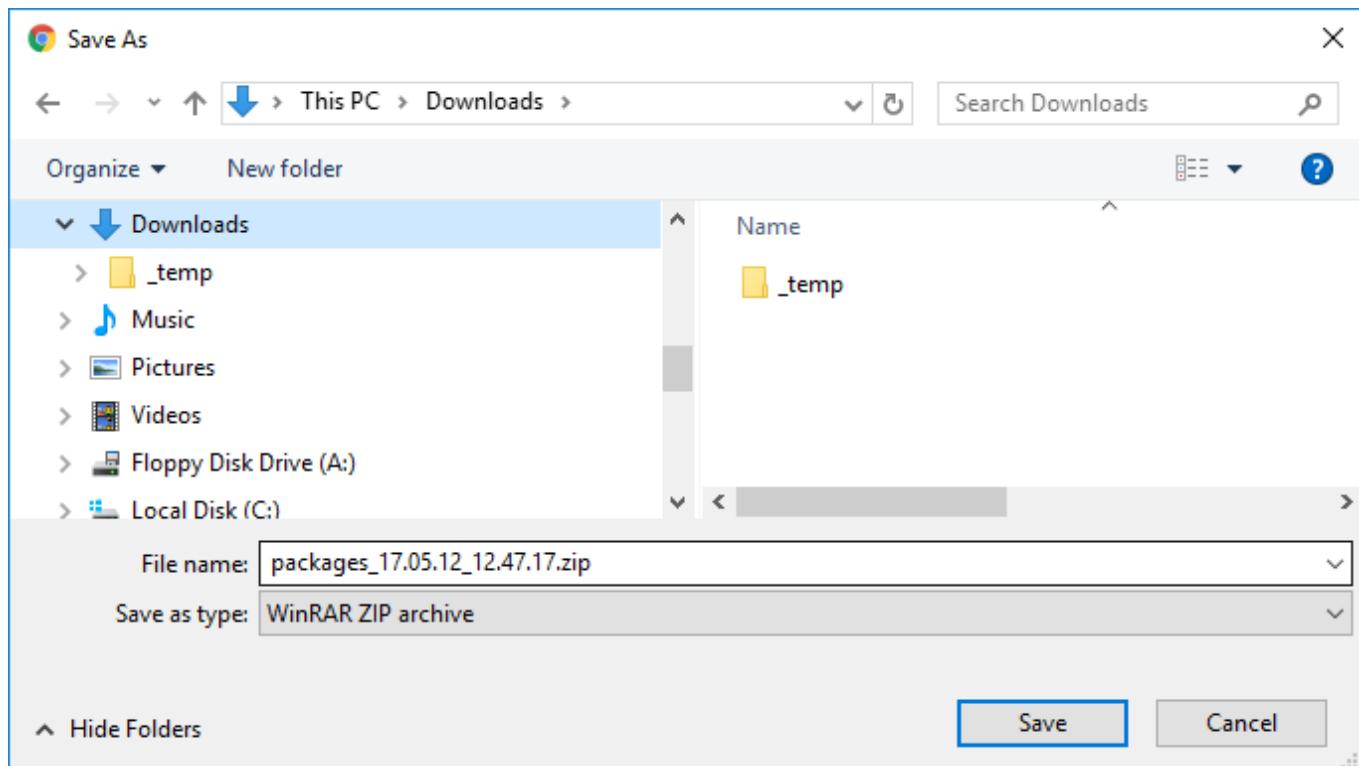
1. Go to the **[Configuration] section**.
2. On the **[Packages]** tab, select one or multiple packages (hold Ctrl or Shift to select multiple packages).
3. Trigger the **[Export packages to archive]** action (Fig. 1).

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



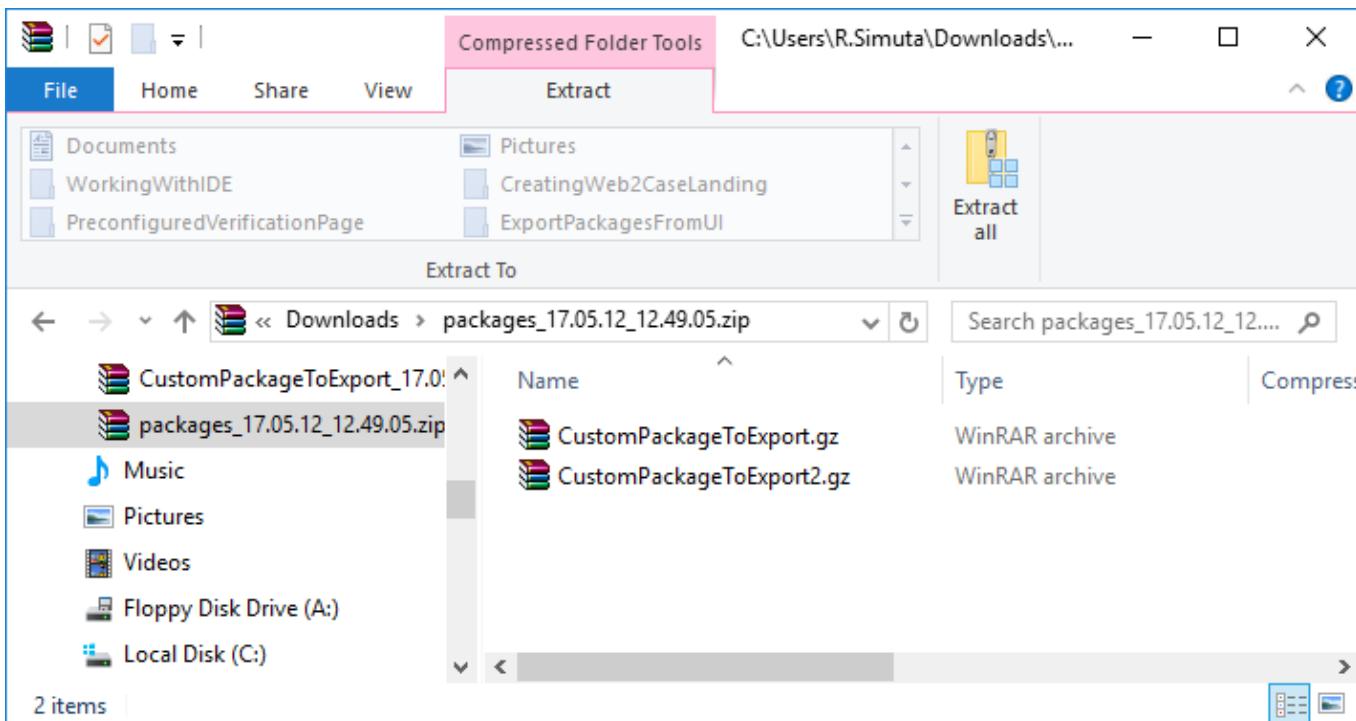
Depending on the browser settings, the zip-archive with packages will either be saved to the default downloads folder, or the browser will display a dialog box for selecting a folder for the archive (Fig. 2).

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



The zip-archive will contain one or multiple packages (Fig. 3) and can be imported into another Creatio application.

Fig. 3. A zip-archive with packages



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

Importing packages

Since application version 7.11, marketplace applications can be uploaded and installed directly in the application interface. This functionality is available in the [[Installed applications](#)] section. Installing the application directly from the marketplace is described in more detail in the "[Installing applications from the marketplace](#)" article.

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

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

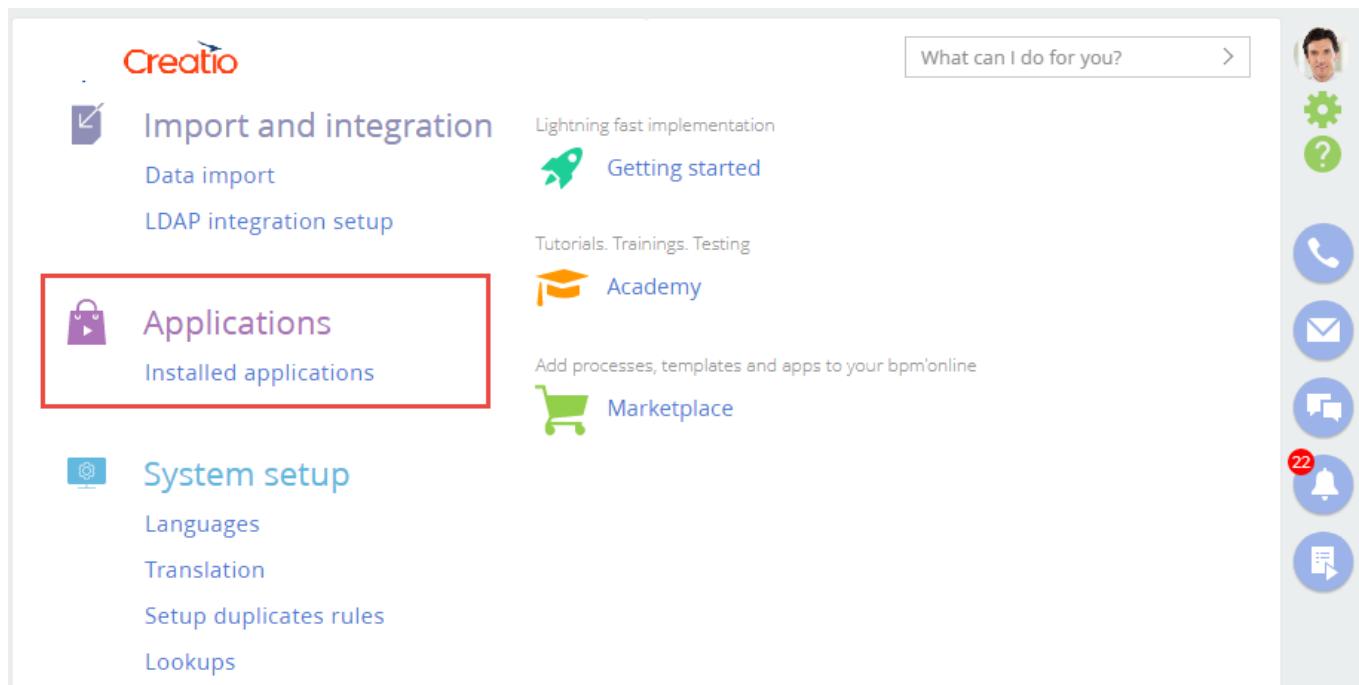
You cannot use the [Custom] package in the marketplace application. For more information about a [Custom] package, please see the "[Package \[Custom\]](#)" article.

Installing an applications from a zip archive

To install packages from the application interface:

1. Go to the [System Designer] section and in the [Admin area] group, click the [[Installed applications](#)] link (Fig. 4). The [[Installed applications](#)] section will open in a separate window.

Fig. 4. System designer



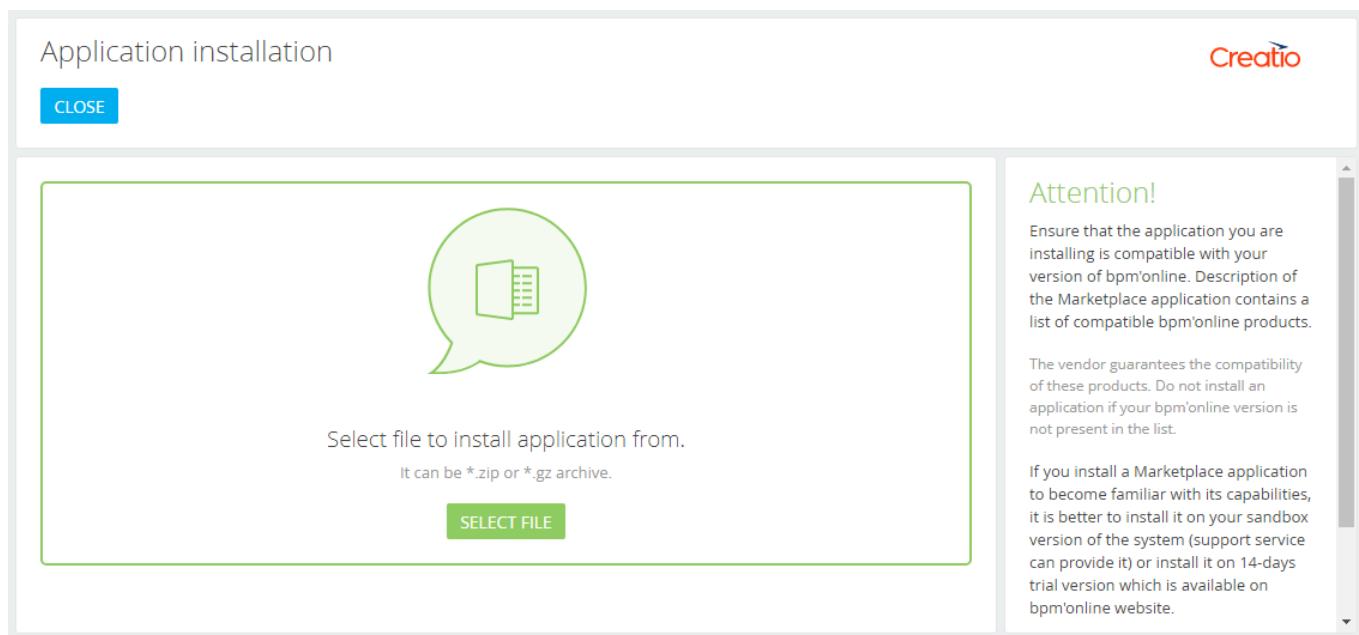
2. In the [Installed applications] section, select the [Install from file] command from the [Add application] drop-down menu (Fig. 5).

Fig. 5. The marketplace application installation menu

This screenshot shows the 'Installed applications' page. At the top, it says 'Installed applications'. Below that is a green button labeled 'ADD APPLICATION ▾'. Underneath the button, there are two options: 'Choose from Marketplace' and 'Install from file'. The 'Install from file' option is highlighted with a red border.

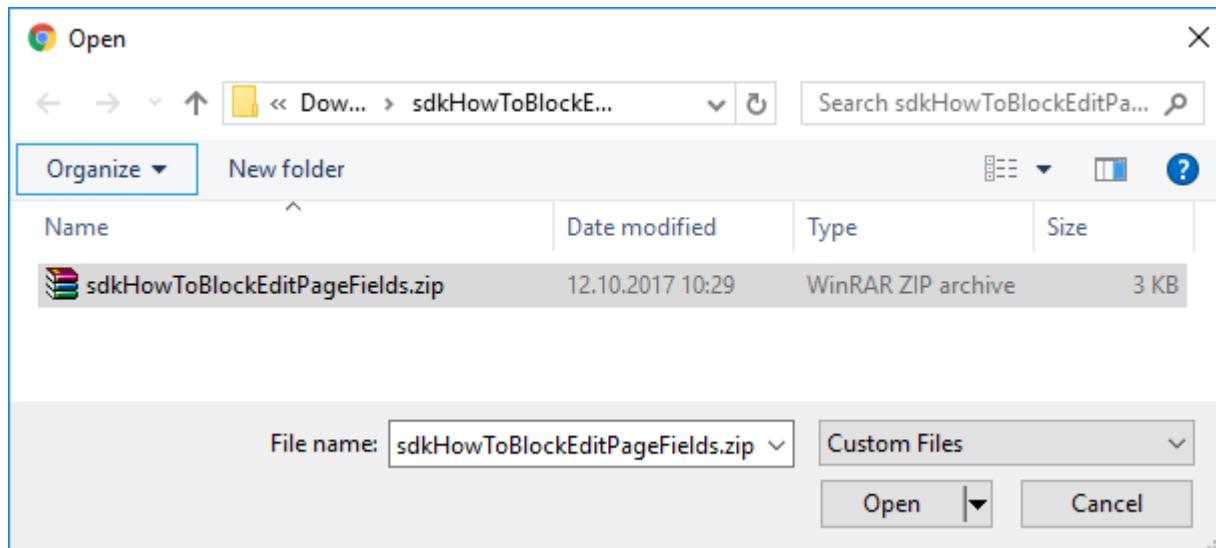
A page for installing an application will open in a separate window (Fig. 6).

Fig. 6. Application installation page



3. Click on the [Select file] button, and select the necessary *.zip archive (Fig. 7).

Fig. 7. Selecting a package for import



The system will be backed up (Fig. 8) and the application will be installed (Fig. 9).

Fig. 8. System backup

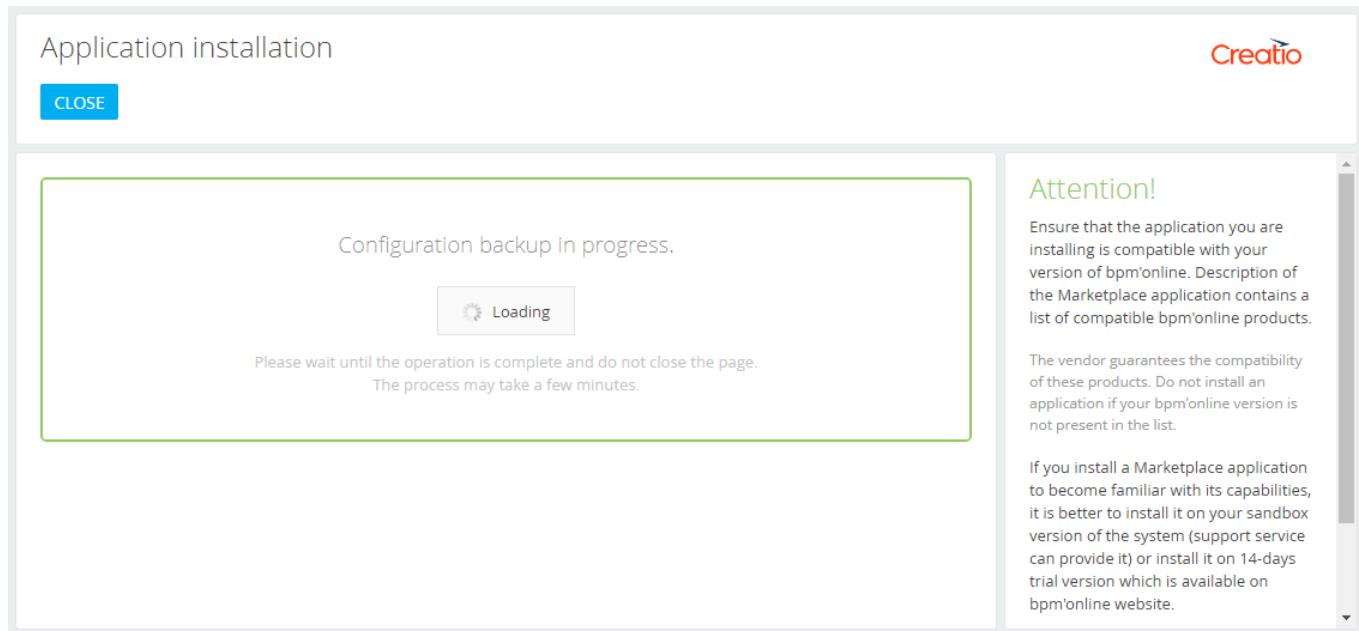
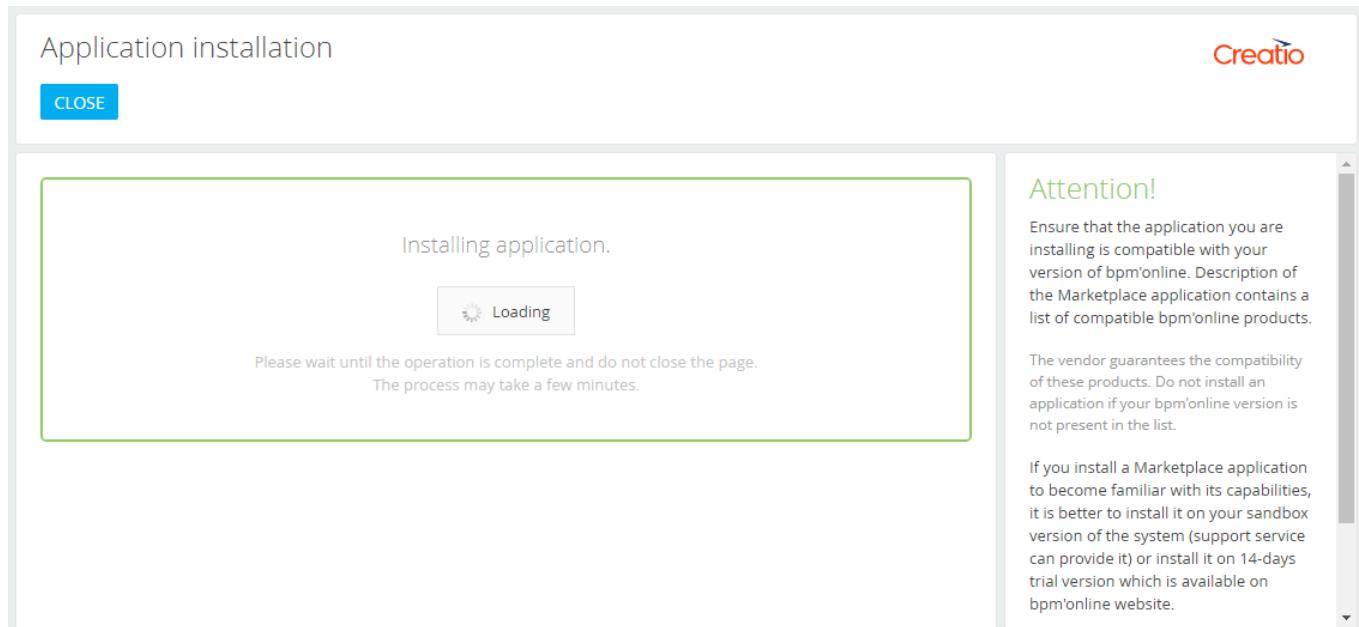


Fig. 9. Application setup



A corresponding message will be displayed (Fig. 10, Fig. 11).

Fig. 10. Successful installation message

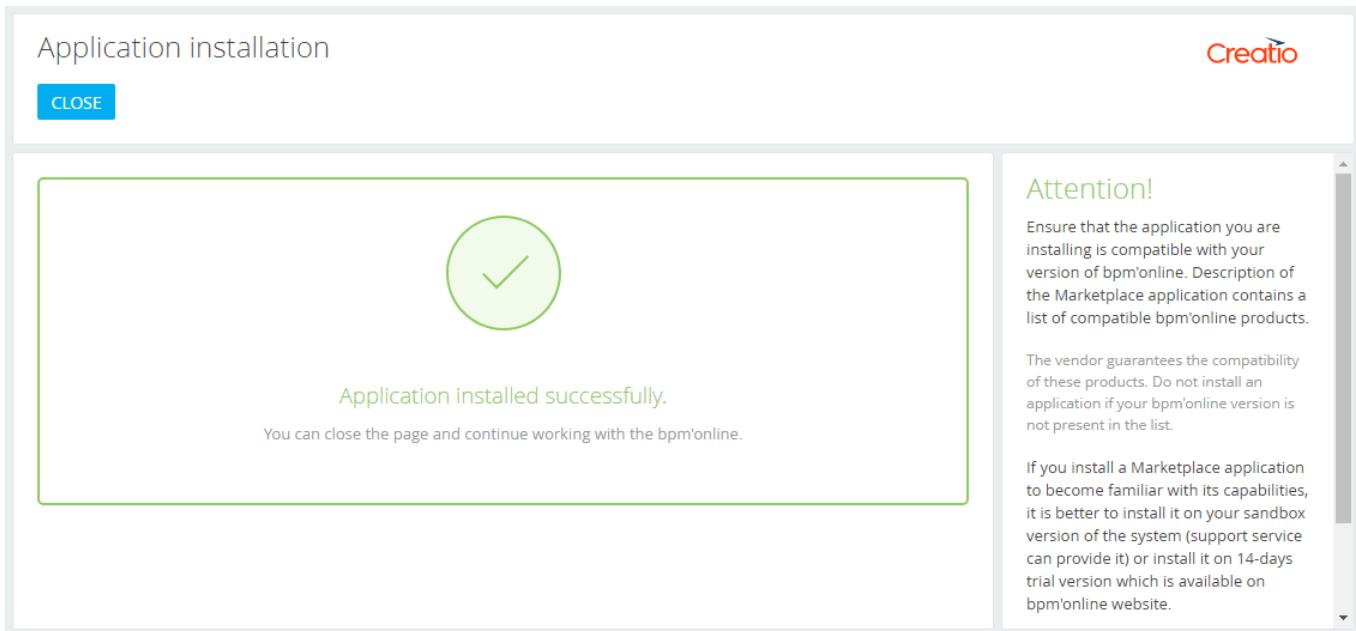
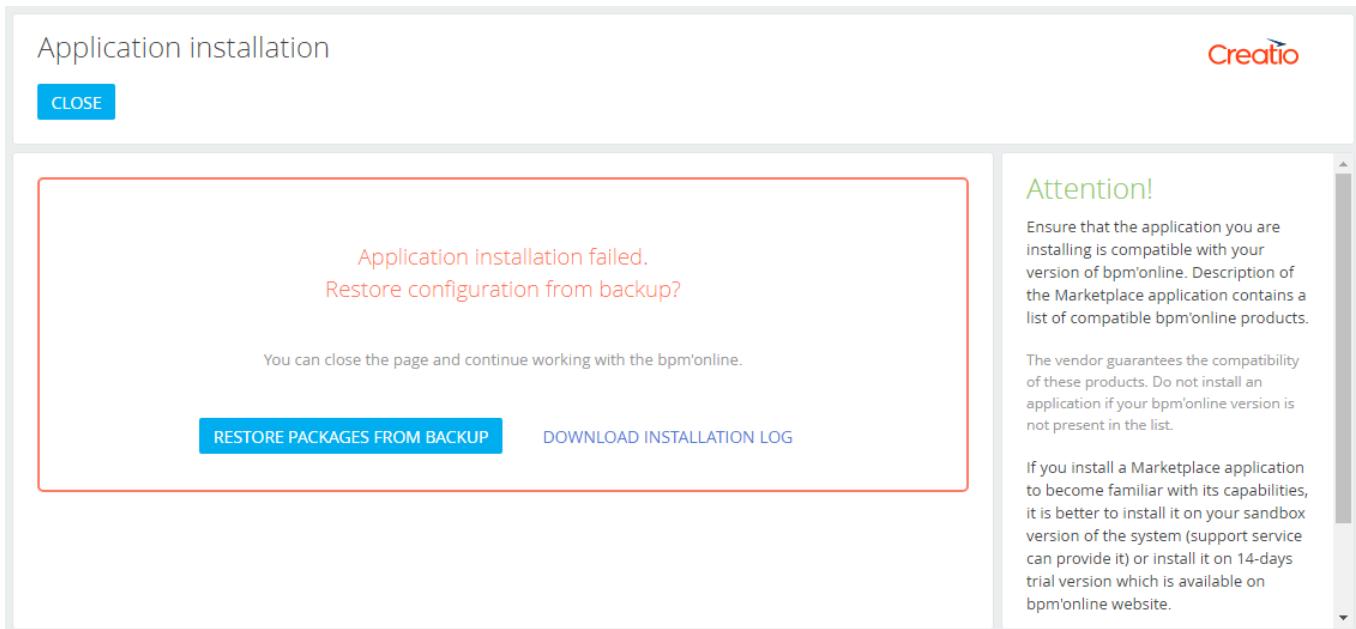


Fig. 11. Unsuccessful package installation message



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

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

Restoring from backup

If you encounter an error during the installation process, you can restore the previous configuration by clicking the [Restore packages from backup] button (Fig. 11). After the backup is restored, a corresponding message will be displayed (Fig. 12). You can select another package file for import.

Fig. 12. Restoring a backup copy message

Application installation

CLOSE

The configuration has been successfully restored from backup.

You can close this page and continue with the bpm'online.

[DOWNLOAD INSTALLATION LOG](#)

Attention!

Ensure that the application you are installing is compatible with your version of bpm'online. Description of the Marketplace application contains a list of compatible bpm'online products.

The vendor guarantees the compatibility of these products. Do not install an application if your bpm'online version is not present in the list.

If you install a Marketplace application to become familiar with its capabilities, it is better to install it on your sandbox version of the system (support service can provide it) or install it on 14-days trial version which is available on bpm'online website.

Transferring changes using schema export and import

Beginner

Easy

Medium

Advanced

Introduction

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

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

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

Advantages:

Ability to replace contents of a schema quickly.

Disadvantages and limitations:

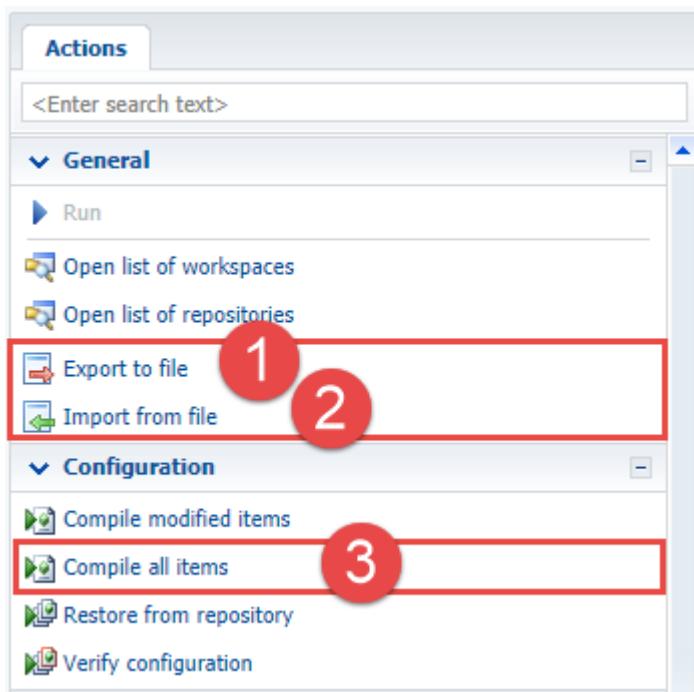
- The mechanism enables you to export and import only schemas. You cannot export or import packages. In addition, it is not possible to transfer data connected to the packages.
- Administrator access rights in the application are required.
- It is not possible to load several schemas at the same time.

Exporting schemas

To export a schema:

1. Go to the **[Configuration]** section of the system designer.
2. Select the package in which the schema is located.
3. Select a schema to export.
4. Click the **[Export to file]** on the **[Actions]** tab (Fig. 1,1).

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



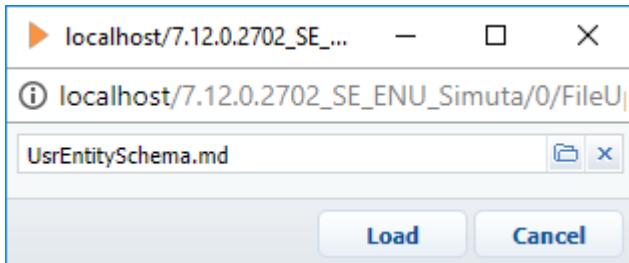
A file with the schema name and *.md extension saved on your hard drive as a result.

Importing schemas

To import a schema:

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

Fig. 2. Import file selection window



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

Transferring changes using SVN

Beginner

Easy

Medium

Advanced

Introduction

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

The purpose of version control system in Creatio is:

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

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

Benefits of transferring changes via SVN

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

The recommended steps for transferring changes via SVN

Using SVN is not recommended for transferring changes to the production environment. The best way for this is using **packages export and import**. Transferring changes with SVN can only be used in the development environment.

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

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

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

2. Enable mechanisms for automatic application of changes.

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

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

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

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

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

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

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

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

5. Install the package from the repository

The sequence of package installation is described in detail in the "[Installing packages from repository](#)".

Important!

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

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

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

Debugging tools

Contents

- **Client code debugging**
- **IsDebug mode**

Client code debugging

Beginner Easy Medium Advanced

Introduction

The client part of the Creatio application is represented by configuration schemas (modules), described in JavaScript language. Debugging of the source code of configuration schemas is executed directly from the browser. Developer tools that provide for debugging for all browsers, supported by Creatio, are used for this purpose.

To run tools for client debugging, execute the following command in a browser:

- Chrome: *F12* or *Ctrl + Shift + I*.
- Firefox: *F12*.
- Internet Explorer: *F12*.

Possibilities for debugging of Creatio client code

All supported browsers provide mostly similar capabilities for debugging client code. Most common and frequently used debugging methods are listed below. For more details about debugging with browser tools, see the following documentation:

- [Chrome developer tools](#)
- [Firefox developer tools](#)
- [Internet Explorer developer tools](#)

Scripts and breakpoints

You can view the full list of scripts, connected to the page and downloaded to a content by means of developer tools. Open any script to set a breakpoint in the place where you want to stop execution of a source code. In the stopped code, you can view current values of variables, execute commands etc.

To set a viewpoint, take the following actions:

- open necessary script file (for example, execute name lookup by combination of buttons *Ctrl+O* and *Ctrl+P*);
- go to code string where you want to set a breakpoint (for example, execute script lookup on the basis of method name);
- set a breakpoint by one of the following methods: click string name, press *F9* button or select "Add breakpoint" item in right-click menu (cursor should be in the string, to which you want to add breakpoint).

You can also set a conditional breakpoint, for which you should set a condition for activation of the breakpoint.

You can also break an execution process directly from the code by the debugger command:

```
function customFunc (args) {  
    ...  
    debugger; // <-- debugger stops here.  
    ...  
}
```

Execution control

The debugging process is reduced to breaking of code execution at the breakpoint, verification of variable values and call stack. Code tracing is executed further for detection of fragments where program behavior deviates from predicted behavior.

The following command is used in browser debuggers for code-based turn-by-turn navigation (figure 1, figure 2 and figure 3):

- suspend/continue script execution (1);
- perform step without entering the function (2);
- perform step with entering the function (3);
- perform step before exiting from current function (4).

Fig. 1. — Navigation panel in Chrome browser debugger



Fig. 2. — Navigation panel in Firefox browser debugger



Fig. 3. — Navigation panel in Internet Explorer browser debugger



Chrome browser provides an additional two commands for execution control:

- deactivate all breakpoints (5);
- deactivate/activate automatic break in case of error (6).

For more information about possibilities and commands of navigation panel for a browser, see corresponding documentation.

Browser console use

In the course of debugging, you can execute JavaScript commands, display debugging, trace information, execute measurements and code profiling. The `console` object is used for this purpose.

JavaScript commands calling

To start operation of the browser console, you should open it by going to the [Console] tab or opening it in addition to the debugger, using the [Esc] button. You can then enter commands in JavaScript and start their execution by pressing [Enter].

Debug information output

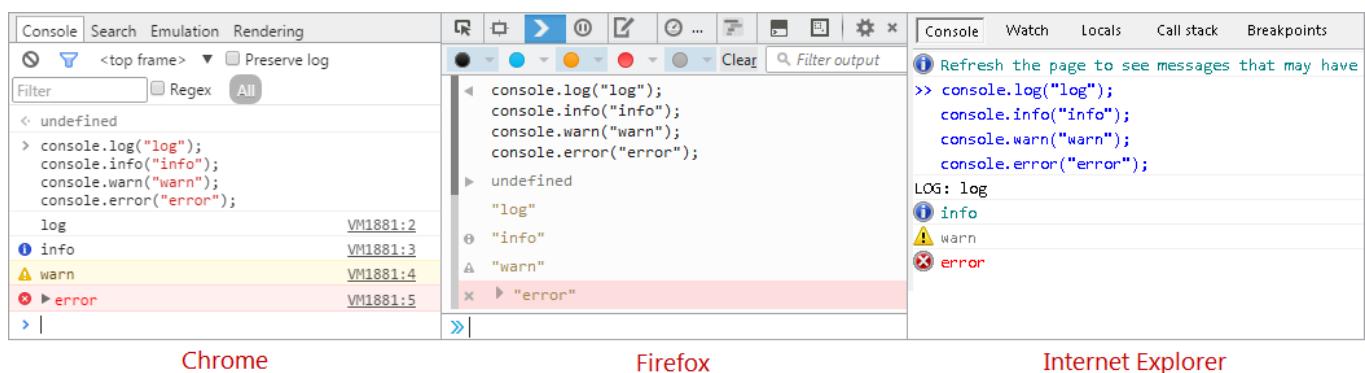
You can enable debugging information of a different nature, i.e. info messages, warnings and error messages, in the console. For this purpose you can use corresponding `console` object methods (table 1).

Table 1. — Console methods for output of debug messages.

Method	Description	Chrome	Firefox	Internet Explorer
console.log(object [, object, ...])	Outputs arguments in console and separate them with comma. Used for enabling different general messages.	+	+	+
console.info(object [, object, ...])	Similar to log() method but outputs messages in other style (figure 4) and emphasizes their significance.	+	+	+
console.warn(object [, object, ...])	Outputs warning message in console.	+	+	+
console.error(object [, object, ...])	Outputs error message in console.	+	+	+ (8+)

An individual style is used for each type of outputted message (figure 4).

Fig. 4. — Styles of different types of console messages



The represented console methods support formatting of outputted messages. This means that you can use special controlling sequences (templates) that will be replaced by corresponding values (arguments, additionally transferred to the function).

Console methods support the following formation templates (table 2).

Table 2. — Console message formation templates

Template	Data type	Example of use
%s	String	console.log("%s is one of flagship products of a company %s", "Sales Creatio", "Terrasoft");
%d, %i	Number	console.log("Platform %s was issued for the first time ever in %d year", "Creatio", 2011);
%f	Float	console.log("Pi character is equal to %f", Math.PI);
%o	DOM-item (it is not supported by IE)	console.log("DOM-View of item <body/>: %o", document.getElementsByTagName("body")[0]);
%O	JavaScript Object (is not supported by IE and Firefox)	console.log("Object: %O", {a:1, b:2});
%c	CSS style (is not supported by IE)	console.log("%cGreen text, %cRed Text on a blue background, %cCapital letters, %cPlain text", "color:green;", "color:red; background:blue;", "font-size:20px;", "font:normal; color: normal; background: normal");

Tracing and validations

Table 3 shows console methods for tracing and verification of expressions.

Table 3. — Console methods for tracing and verification

Method	Description	Chrome	Firefox	Internet Explorer
console.trace()	Outputs call stack from code point where method was called. Call stack includes file names, string numbers and also call counters of trace() method from one and the same point.	+	+	+ (11+)
console.assert(expression[, object, ...])	Verified expression, transferred as an expression parameter and, if the expression is false, outputs error with (console.error ()) call stack in the console, otherwise it outputs nothing.	+	+ (28+)	+

Console.trace() method outputs informative stack-trace with full list of functions and their arguments at the moment of call.

Due to the console.trace() method you can comply with rules in the code and ensure that code execution results meet expectations. Using console.assert () you can execute code testing, i.e. if execution result is unsatisfactory, the corresponding value will be discarded.

An example of the console.assert() method for testing of results:

```
var a = 1, b = "1";
console.assert(a === b, "A is not equal to B");
```

Profiling and measurement

You can measure code execution time with browser console methods (table 4).

Table 4. – Console methods for measurement of code execution time

Method	Description	Chrome	Firefox	Internet Explorer
console.time(label)	Starts counter (milliseconds) with label.	+	+	+ (11+)
console.timeEnd(label)	Stops counter (milliseconds) with label and plans result in console.	+	+	+ (11+)

An example of console.time() and console.timeEnd() methods in code:

```
var myArray = new Array();
// Starts counter with Initialize myArray tag.
console.time("Initialize myArray");
myArray[0] = myArray[1] = 1;
for (i = 2; i<10; i++)
{
    myArray[i] = myArray[i-1] + myArray[i-2];
}
// Stops counter with Initialize myArray tag.
console.timeEnd("Initialize myArray");
```

You also can execute code profiling and output profiling stacks that contain detailed information about how much time was spent by a browser for definite operations.

Table 5. – Console methods for code profiling

Method	Description	Chrome	Firefox	Internet Explorer
--------	-------------	--------	---------	-------------------

console.profile(label)	Runs Java Script profiler and displays results, marked with label.	+	+ (when DevTools panel is opened)	+ (10+)
console.profileEnd(label)	Stops Java Script profiler.	+	+ (when DevTools panel is opened)	+ (10+)

You can view profiling results in:

- Chrome — Profiles tab;
- Firefox — Performance tab;
- Internet Explorer — Profiler tab.

IsDebug mode

Beginner

Easy

Medium

Advanced

Introduction

You need the ‘isDebug’ mode for getting detailed data about errors and tracing them in the source code. You can debug the source code using built-in browser tools. For more information about the built-in developer tools, please see the "**Client code debugging**" article.

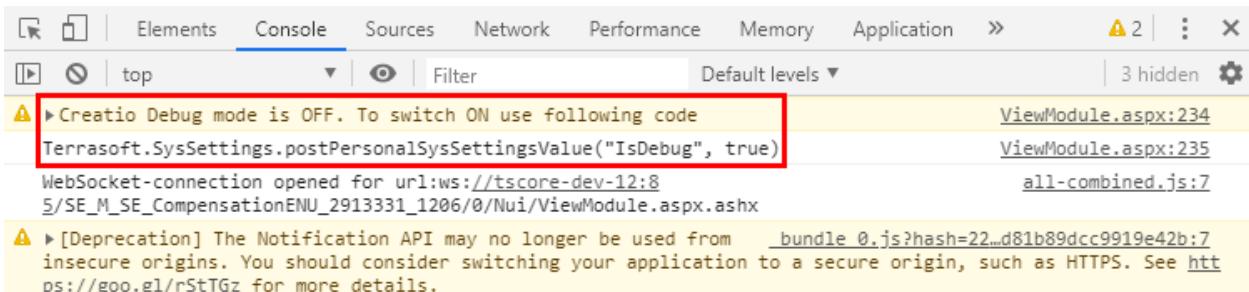
In the normal mode, the code in the browser is **minified**. This means that the client scripts are compiled in the *all-combined.js* file that contains the core functionality. The file is updated during the build compilation. The ‘isDebug’ mode disables compilation and compression of the core JS files and allows retrieving the client scripts as a series of individual files.

The ‘isDebug’ mode is implemented in version 7.13.0 and up.

Enabling the client debugging mode

To check the current status of the client debugging mode, open the browser console (fig. 1) by pressing the F12 key or pressing Ctrl+Shift+I. Aside from the current status of the client debugging mode, the console will display the code to enable or disable debugging.

Fig. 1. – ‘isDebug’ mode status



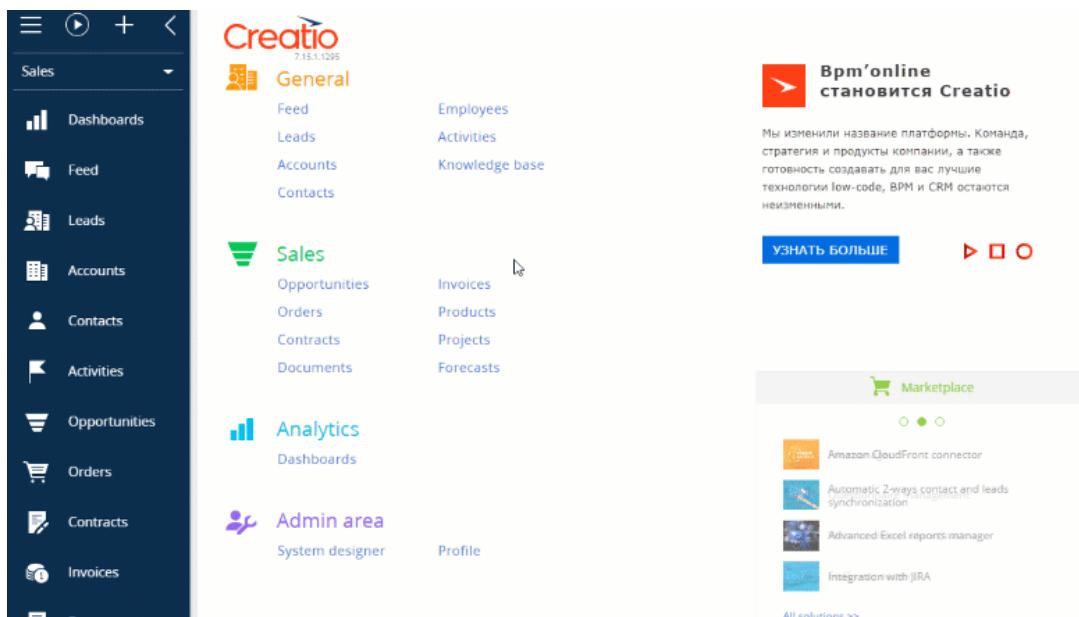
You can enable the client debugging mode using the following methods:

- Execute the following code in the browser console:

```
Terrasoft.SysSettings.postPersonalSysSettingsValue("IsDebug", true)
```

- Change the value of the [Debug mode] system setting (fig. 2).

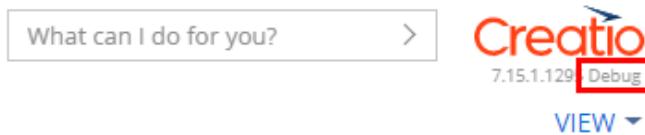
Fig. 2. – Changing the ‘isDebug’ system setting value



To apply the changes, refresh the page or hit F5.

Upon activating the client debugging mode, you will see the Debug indicator next to the site's version number.

Fig. 3. – The client debugging mode indicator



Enabling the client debugging mode will affect site performance. For instance, it can increase the time needed for the pages to load.

Example. The browser console displaying the error details.

The figures below show examples of errors displayed in the console with the 'isDebug' mode disabled (fig. 4) and enabled (fig. 5).

Fig. 4. – Displaying an error ('isDebug' disabled)

```
✖ ▼Uncaught TypeError: Cannot read property 'value' of undefined InvoicePageV2.js?has...c0f99002aff011:1741
  at i.setCardLockoutStatus (InvoicePageV2.js?has...c0f99002aff011:1741)
  at i.onEntityInitialized (InvoicePageV2.js?has...c0f99002aff011:1752)
  at Object.callback (all-combined.js:6)
  at i.<anonymous> (BasePageV2.js?hash=6...c0f99002aff011:1108)
  at i.e (all-combined.js:7)
  at Object.callback (all-combined.js:6)
  at i.<anonymous> (all-combined.js:7)
  at Object.callback (all-combined.js:6)
  at i.<anonymous> (all-combined.js:7)
  at Object.callback (all-combined.js:6)

setCardLockoutStatus      @ InvoicePageV2.js?has...c0f99002aff011:1741
onEntityInitialized        @ InvoicePageV2.js?has...c0f99002aff011:1752
callback                  @ all-combined.js:6
(anonymous)                @ BasePageV2.js?hash=6...c0f99002aff011:1108
e                         @ all-combined.js:7
callback                  @ all-combined.js:6
(anonymous)                @ all-combined.js:7
callback                  @ all-combined.js:6
(anonymous)                @ all-combined.js:7
callback                  @ all-combined.js:6
_parseGetEntityResponse   @ all-combined.js:7
(anonymous)                @ all-combined.js:7
callback                  @ all-combined.js:7
e.callback                @ all-combined.js:7
callback                  @ all-combined.js:6
onComplete                @ all-combined.js:6
onStateChange              @ all-combined.js:6
(anonymous)                @ all-combined.js:6

XMLHttpRequest.send (async)
request                   @ all-combined.js:6
request                   @ all-combined.js:7
executeRequest            @ all-combined.js:7
callParent                @ all-combined.js:6
executeRequest            @ all-combined.js:7
executeQuery               @ all-combined.js:7
getEntity                 @ all-combined.js:7
load                      @ all-combined.js:7
loadEntity                @ all-combined.js:7
(anonymous)                @ BasePageV2.js?hash=6...c0f99002aff011:1104
e                         @ all-combined.js:7
callback                  @ all-combined.js:6

XMLHttpRequest.send (async)
request                   @ all-combined.js:6
request                   @ all-combined.js:7
executeRequest            @ all-combined.js:7
```

Fig. 5. – Displaying an error ('isDebug' enabled)

```
✖ ▼Uncaught TypeError: Cannot read property 'value' of undefined InvoicePageV2.js?has...c0f99002aff011:1741
  at constructor.setCardLockoutStatus (InvoicePageV2.js?has...c0f99002aff011:1741)
  at constructor.onEntityInitialized (InvoicePageV2.js?has...c0f99002aff011:1752)
  at Object.callback (extjs-base-debug.js:11584)
  at constructor.<anonymous> (BasePageV2.js?hash=6...c0f99002aff011:1108)
  at constructor.nextFn (commonutils.js?hash=...7c0f99002aff011:130)
  at Object.callback (extjs-base-debug.js:11584)
  at constructor.<anonymous> (entity-base-view-mod...7c0f99002aff011:977)
  at Object.callback (extjs-base-debug.js:11584)
  at constructor.<anonymous> (entity-data-model.js...7c0f99002aff011:177)
  at Object.callback (extjs-base-debug.js:11584)
setCardLockoutStatus          @ InvoicePageV2.js?has...c0f99002aff011:1741
onEntityInitialized            @ InvoicePageV2.js?has...c0f99002aff011:1752
callback                      @ extjs-base-debug.js:11584
(anonymous)                   @ BasePageV2.js?hash=6...c0f99002aff011:1108
nextFn                        @ commonutils.js?hash=...7c0f99002aff011:130
callback                      @ extjs-base-debug.js:11584
(anonymous)                   @ entity-base-view-mod...7c0f99002aff011:977
callback                      @ extjs-base-debug.js:11584
(anonymous)                   @ entity-data-model.js...7c0f99002aff011:177
callback                      @ extjs-base-debug.js:11584
_parseGetEntityResponse       @ entity-schema-query...7c0f99002aff011:487
(anonymous)                   @ entity-schema-query...7c0f99002aff011:558
callback                      @ base-service-provide...7c0f99002aff011:126
config.callback                @ ajax-provider.js?has...7c0f99002aff011:157
callback                      @ extjs-base-debug.js:11584
onComplete                     @ extjs-base-debug.js:46413
onStateChange                  @ extjs-base-debug.js:46349
(anonymous)                   @ extjs-base-debug.js:3278
XMLHttpRequest.send (async)
request                        @ extjs-base-debug.js:45742
request                        @ ajax-provider.js?has...7c0f99002aff011:177
executeRequest                 @ base-service-provide...7c0f99002aff011:289
callParent                     @ extjs-base-debug.js:6836
executeRequest                 @ service-provider.js?...97c0f99002aff011:73
executeQuery                   @ data-provider.js?has...7c0f99002aff011:138
getEntity                      @ entity-schema-query...7c0f99002aff011:556
load                           @ entity-data-model.js...7c0f99002aff011:174
loadEntity                     @ entity-base-view-mod...7c0f99002aff011:971
(anonymous)                   @ BasePageV2.js?hash=6...c0f99002aff011:1104
nextFn                        @ commonutils.js?hash=...7c0f99002aff011:130
callback                      @ extjs-base-debug.js:11584
XMLHttpRequest.send (async)
request                        @ extjs-base-debug.js:45742
request                        @ ajax-provider.js?has...7c0f99002aff011:177
executeRequest                 @ base-service-provide...7c0f99002aff011:289
```

For solving complex customization tasks and server-side development

Contents

- **Introduction**
- **Development tools. IDE Microsoft Visual Studio**
- **Development tools. SQL**
- **Version control systems**

- **Testing tools.** **NUnit**
- **Logging tools**
- **Server code debugging**
- **Delivery tools**

Complex customization tasks and working with the server code

Beginner

Easy

Medium

Advanced

Introduction

Complex customization tasks require working with the server code and developing projects with a wide range of functions. Unlike basic tasks, the implementation of complex customization tasks requires involvement from several developer teams, as well as enabling file system development mode.

The file system development mode enables using the following tools:

- development tools
- version control systems
- testing tools
- logging tools
- server code debugging tools
- solution transferring tools.

Team development implies using an integrated development environment, such as **Microsoft Visual Studio**, which enables both the coding and **debugging** of the code, using version control systems and more. You can use any suitable version control system (for instance, **SVN**, **Git**, etc.) to keep track of the change history, as well as add, remove, and move files and directories.

To test the isolated program components, we recommend using a .NET-application Unit-testing framework, **NUnit**. To enable logging in Creatio, we recommend using third-party libraries (for instance, **NLog**).

During the implementation of complex development tasks by a team of developers, it often becomes necessary to transfer changes between different development environments. We recommend using built-in tools to avoid errors during the migration. Aside from built-in tools, you can use additional transferring tools (**the WorkspaceConsole utility** and the **command line utility ('Command Line Interface' in the on-line documentation)**).

Development tools. IDE Microsoft Visual Studio

Contents

- **Introduction**
- **IDE settings for development**
- **Working with the server side source code**
- **Working with the client code**
- **File content**

Development in the file system

Beginner

Easy

Medium

Advanced

Introduction

Using an [Integrated Development Environment \(IDE\)](#) maximizes development speed. Examples of the IDE include Visual Studio, WebStorm and other tools. An IDE usually enables you to create, modify and compile the source code, debug it, run team development, use version control systems, etc. IDEs usually use text files stored in the file system to work with the source code.

For development in the file system, you can use Microsoft Visual Studio Community, Professional and Enterprise

version 2017 (with latest updates) and higher.

You can configure Creatio **configuration packages** in the file system. With this mechanism, you can export the packages from the database to a set of files, edit the package content using an IDE and upload the updated packages back to the database. Using Visual Studio, you can debug custom source code of the schemas of the “Source code” (*SourceCodeSchema*) type.

Use Creatio [built-in tools](#) if there is no need or possibility to develop in the file system.

Main limitations of development in the file system

When the development in the file system mode is enabled, a full-fledged development is supported only for schemas of the “Source code” (*SourceCodeSchema*) and “Client schema” (*ClientUnitSchema*) types.

For other package elements (such as resources and SQL scripts), the following rules are used:

- When exporting packages from the database to the file system, the package elements that are stored in the database will replace the corresponding items in the file system. The source code schemas and client schemas will not be replaced.
- When uploading packages to the database, source code schemas and client schemas will replace the corresponding items in the database. The application will keep using source code schemas and client schemas from the file system.

Starting with version 7.11.2, when the file system development mode is enabled, resources of these schemas are also saved in the file system after saving client schemas (*ClientUnitSchema*) and source code schemas (*SourceCodeSchema*) in the corresponding designers (see the [\[Configuration\] section](#)).

Integration with the version control system (SVN) with enabled development in the file system is performed with third-party tools. Creatio built-in mechanism of working with SVN is not used. It is still possible to install packages from the SVN repository in the [Configuration] section (this simplifies working with related packages). Use third-party utilities, such as [TortoiseSVN](#) to install separate packages.

To use the built-in capabilities of working with the SVN repository, disable the development mode in the file system.

Application settings for development in the file system

To enable development in the file system, edit the Web.config file (located in the root folder with the installed application) and set the *enabled* attribute of the *fileDesignMode* element to *true*.

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

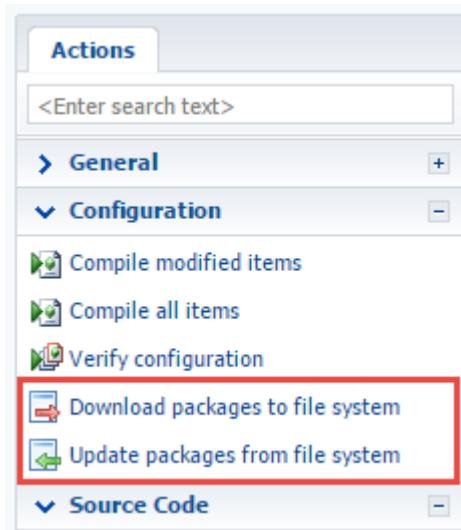
Currently, the development in the file system is not compatible with getting client content from preliminary generated files. For the correct work of the development in the file system you need to disable getting static client content from the file system. Set the “false” for the *UseStaticFileContent* flag in the Web.config file to disable this function.

```
<fileDesignMode enabled="true" />
...
<add key="UseStaticFileContent" value="false" />
```

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

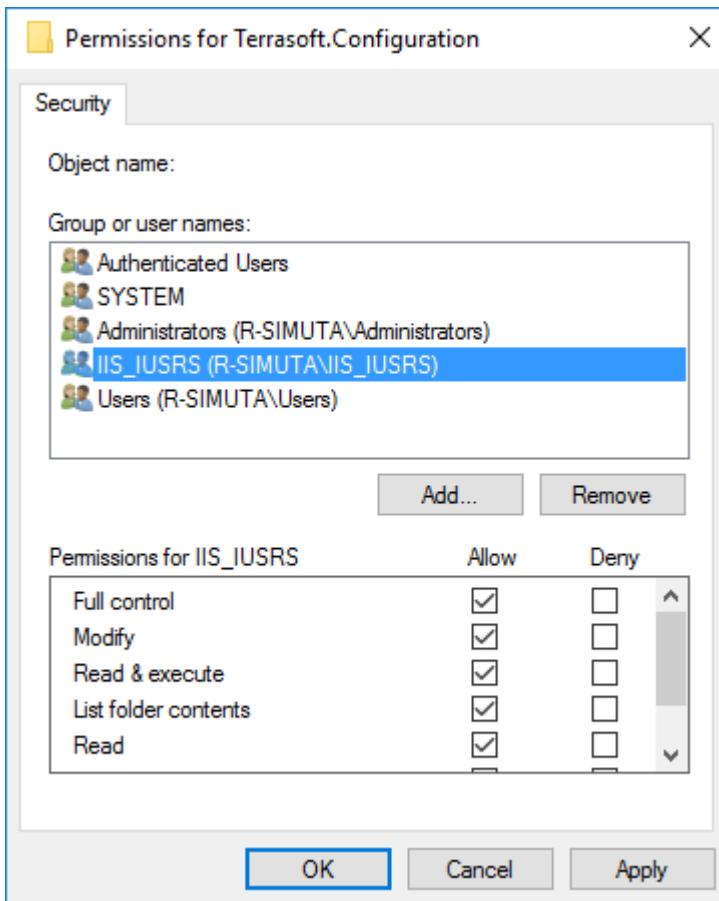
- [Download packages to file system] – exports the packages from the database to the following folder: *[path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg*.
- [Update packages from file system] – uploads the packages to the database from the following folder: *[path to the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

Fig. 1 Actions of the [Configuration] section for development in the file system



To integrate the application with the configuration project, grant full access to the *[path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg* folder for the OS user, who runs the IIS application pool (Fig. 2). Usually, this is the built-in IIS_IUSRS user.

Fig. 2 Setting up access rights for the Terrasoft.Configuration folder



Terrasoft.Configuration package

A configuration project is a Visual Studio solution supplied with Creatio setup files. The solution can be found here: *[path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration*.

To start development in the file system, open the following file in Visual Studio: *[path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Terrasoft.Configuration.sln*.

The configuration project structure is available in table 1.

Table 1. Configuration project structure

Folder	Purpose
Lib	The folder where package-bound third-party class libraries are exported.
Autogenerated\Src	The folder with files that contain auto-generated source code of the preset package schemas. These files cannot be edited.
Pkg	The folder where the packages for development in the file system are exported from the database.
bin	A folder for compiled configuration and third-party libraries.

Getting Started with the configuration

Creating a package

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

The working with SVN mode is enabled in the Creatio by default. If the [Version control system repository] field is empty when creating a package, then the package will not be bound to the repository. The versioned development of this package can be performed only after manually binding it to the repository from the file system.

More information about creating a custom package and binding it to the SVN repository can be found in the "**Creating a package in the file system development mode**" article.

Working with new package elements

It is recommended to add new elements (schema or resource) to the package only from the **[Configuration] section**. To create and edit a new item in a custom package:

- Select a custom package in the [Configuration] section and add a new element in it (see **Creating a custom client module schema**, "**Creating the [Source code] schema**").
- Add resources (such as localized strings) to the schema if needed.
- Click [Download packages to file system] (Fig. 1).
- Use an IDE (such as Visual Studio) to edit the source code of the schema or localized resource in the files (located in the [Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\[Package name] folder). The package properties are described in the "**Package structure and contents**" article.
- Click [Update packages from file system] to upload changes to the application database (Fig. 1).

Changes made in client schemas are available in the application immediately, without uploading to the database. You only need to update the page in the browser.

- If you changed a source code schema, then you must compile the application.

When developing source code schemas in C #, compile them directly in Visual Studio. More information about compilation and debugging in Visual Studio can be found in the "**Working with the server side source code**" article.

See also

- **IDE settings for development**
- **Working with the client code**
- **Working with the server side source code**
- **Developing the configuration server code in the user project**
- **Working with the client code. Automatic displaying of changes**
- **Working with SVN in the file system**
- **Packages file content**
- **Localization of the file content**

- **Testing tools. NUnit**
- **How to use TypeScript when developing custom functions**

IDE settings for development

Beginner

Easy

Medium

Advanced

Introduction

Using an [Integrated Development Environment](#) (IDE), Microsoft Visual Studio maximizes development speed. Microsoft Visual Studio IDE usually enables you to create, modify and compile the source code, debug it, run team development, use version control systems, etc.

Development in Microsoft Visual Studio became possible after the implementation of the **configuration packages in the file system** mechanism in Creatio. With this mechanism, you can export the packages from the database to a set of files, edit the package source code using an IDE and upload the updated packages back to the database.

For development in the file system, you can use Microsoft Visual Studio Community, Professional and Enterprise version 2017 (with latest updates) and higher.

The WorkspaceConsole utility integrated into Visual Studio is used to compile applications. The WorkspaceConsole has the following benefits:

- Significantly speeds up the compilation process, because the whole configuration assembly is split into independently compiled modules. Only the modules that contain modified packages are compiled.
- Compilation does not require exiting the debugging mode or disconnecting from the IIS process.

You can also compile the configuration project using the Visual Studio compiler. However, it may not take into account the dependencies and the position of packages.

Visual Studio settings for development in the file system:

1. Enable compilation mode in the IDE.
2. Configure the WorkspaceConsole to compile the application.
3. Configure Microsoft Visual Studio.

Visual Studio configuration steps

1. Enable compilation mode in the IDE

To enable compilation mode in the IDE, edit the Web.config file (located in the root folder with the installed application) and set the *enabled* attribute of the *fileDesignMode* element to *true*.

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

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

Currently, the development in the file system is no compatible with getting client content from preliminary generated files. For the correct work of the development in the file system you need to disable getting static client content from the file system. Set the "false" for the *UseStaticFileContent* flag in the Web.config file to disable this functions.

```
<fileDesignMode enabled="true" />
...
<add key="UseStaticFileContent" value="false" />
```

If the development mode in the file system is enabled, IIS will use the *Terrasoft.Configuration.dll* library only from the file system.

After switching to the file system development mode for the first time, upon logging in, the user is redirected to the "Configuration" section. At this time, "The "Default" workspace assembly is not initialized" error appears. To eliminate this error, run the "Compile all items" action.

2. Configure the WorkspaceConsole to compile the application.

Configuration projects are compiled via the **WorkspaceConsole** utility, which is included in the application setup files. The utility should be configured before using. In addition to the settings covered in the "**WorkspaceConsole settings**" article, you must also enable the development mode in the file system in the *Terrasoft.Tools.WorkspaceConsole.exe.config* configuration file.

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

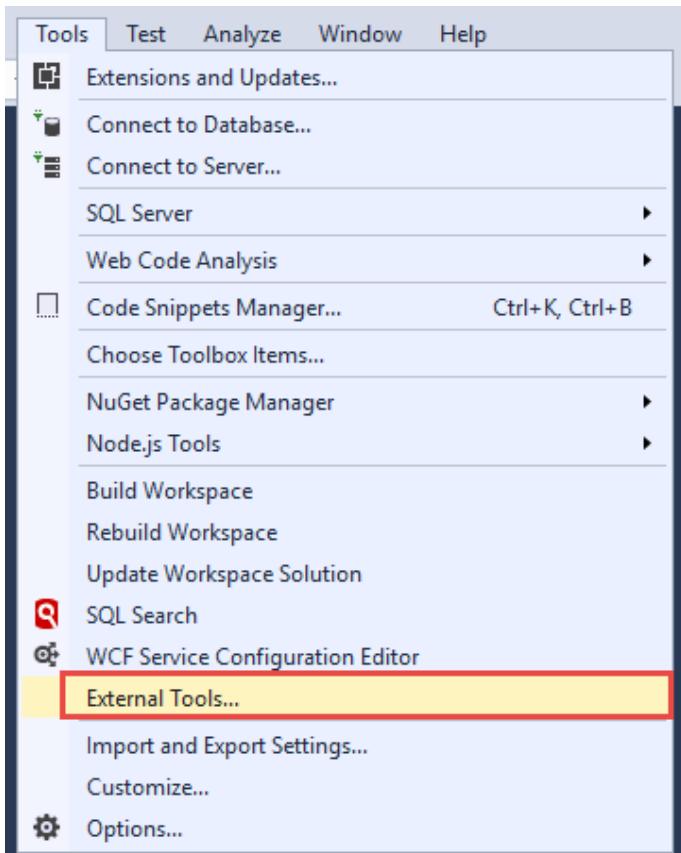
To speed up the compilation by splitting the configuration assembly into independent compiled modules, set the *CompileByManagerDependencies* setting to “true” in the “internal” Web.Config (located in the Terrasoft.WebApp directory) and in the Terrasoft.Tools.WorkspaceConsole.exe.config file of the WorkspaceConsole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

```
<appSettings>
  ...
    <add key="CompileByManagerDependencies" value="true" />
  ...
</appSettings>
```

3. Configure Microsoft Visual Studio

To use the WorkspaceConsole utility for compilation in Visual Studio, set up *External Tools*. To do this, execute the Tools > External Tools... command in the Visual Studio (Fig. 1).

Fig. 1 Adding external tools in the Visual Studio



In the opened dialog window (Fig 2), add and set up three commands for calling the *WorkspaceConsole* utility. The *Build Workspace* and *Rebuild Workspace* commands initiate compilation of changes and full compilation of configuration projects. The *Update Workspace Solution* command updates the Visual Studio solution of the configuration package from the application database. It applies all changes made by the users within the application. The properties and arguments of commands are available in tables 1, 2 and 3. Select the *Use Output window* checkbox (Fig. 2) for all three commands.

Fig. 2 Setting properties and arguments for an external Visual Studio command

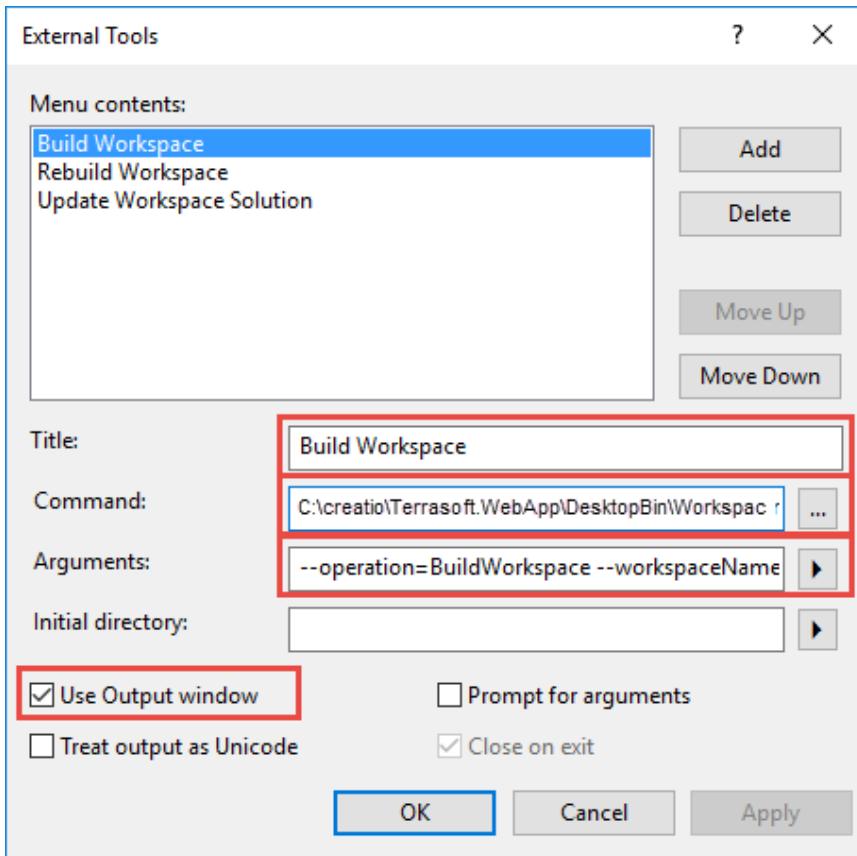


Table 1. Update Workspace Solution command properties

Title	Update Workspace Solution
Command	[Path to the catalog with installed application]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe Example: C:\Creatio\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe
Arguments	--operation=UpdateWorkspaceSolution --workspaceName=Default --webApplicationPath="[Path to the catalog with installed application]\Terrasoft.WebApp" --confRuntimeParentDirectory="[Path to the catalog with installed application]\Terrasoft.WebApp" Example: --operation=UpdateWorkspaceSolution --workspaceName=Default --webApplicationPath="C:\Creatio\Terrasoft.WebApp" --confRuntimeParentDirectory="C:\Creatio\Terrasoft.WebApp"

Table 2. Build Workspace command properties

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

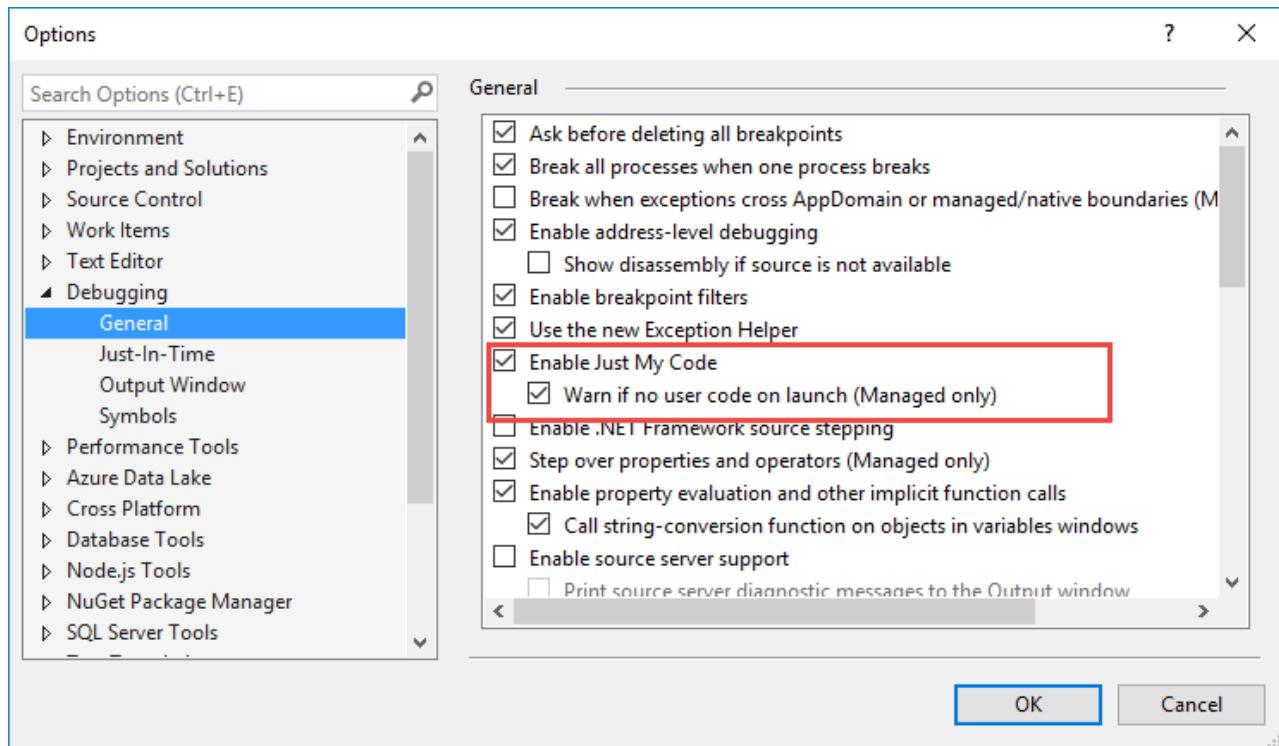
```
webApplicationPath="C:\creatio\Terrasoft.WebApp" --
confRuntimeParentDirectory="C:\creatio\Terrasoft.WebApp"
```

Table 3. Rebuild Workspace command properties

Title	Rebuild Workspace
Command	[Path to the catalog with installed application]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe Example: C:\creatio\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe
Arguments	--operation=RebuildWorkspace --workspaceName=Default --webApplicationPath="[Path to the catalog with installed application]\Terrasoft.WebApp" --confRuntimeParentDirectory="[Path to the catalog with installed application]\Terrasoft.WebApp" Example: --operation=RebuildWorkspace --workspaceName=Default --webApplicationPath="C:\creatio\Terrasoft.WebApp" --confRuntimeParentDirectory="C:\creatio\Terrasoft.WebApp"

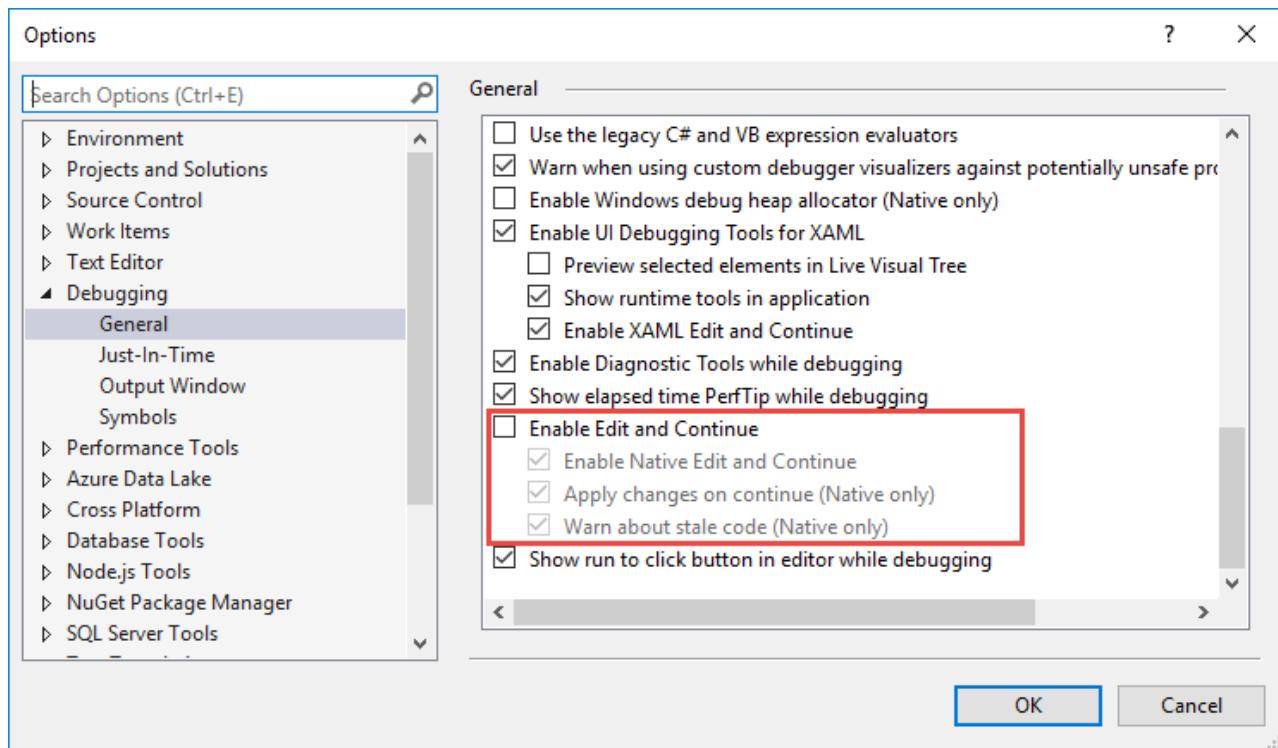
To prevent the debugger from accessing the source code that is disabled in the project, execute the Debug > Options... menu command and enable the *Enable Just My Code* option in the opened dialog (Fig. 3). For more information about the *Enable Just My Code*, please refer to [this page](#).

Fig. 3 Enable Just My Code option



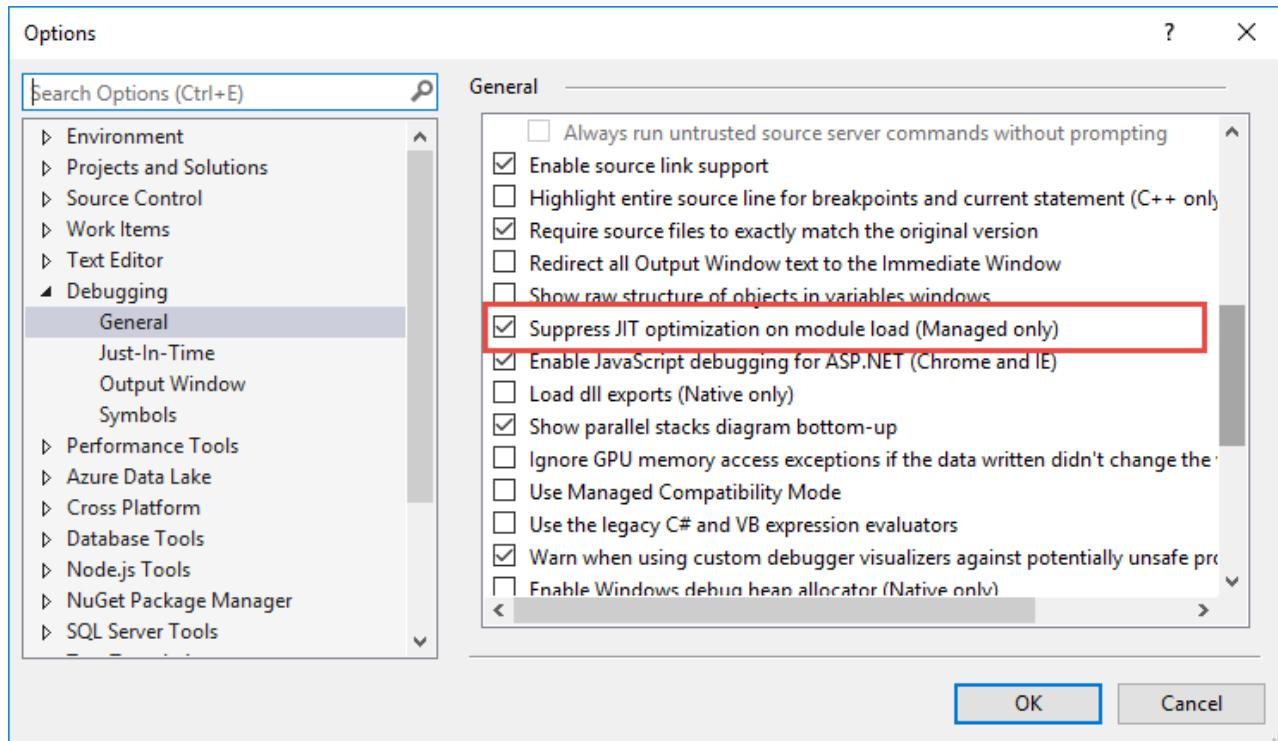
After the configuration is compiled, the application is automatically restarted. The *Edit and Continue* option is not supported and should be disabled (Fig. 4).

Fig. 4 Edit and Continue option



For the debugger to stop correctly on breakpoints, make sure that the *Suppress JIT optimization on module load* option is enabled (Fig. 5).

Fig. 5 Suppress JIT optimization on module load



Working with the server side source code

Contents

- **Working with the server side source code**
- **Developing the configuration server code in the user project**

Working with the server side source code

Beginner

Easy

Medium

Advanced

Introduction

Creatio has the ability to debug program code using the integrated functions of the Visual Studio development environment. The Visual Studio debugger enables developers to freeze the execution of program methods, check variable values, modify them and monitor other activities performed by the program code.

The general procedure for Creatio development in Visual Studio is as follows:

1. Perform preliminary settings in Creatio and Visual Studio.
2. Create, obtain or update a package from the SVN repository.
3. Create a [Source code] schema for development.
4. Perform development in Visual studio.
5. Save, compile and debug the source code.

After successful compilation, the resulting *Terrasoft.Configuration.dll* assembly will be placed to the Bin catalog, while IIS will automatically use it in Creatio application.

General procedures for developing Creatio solutions in Visual Studio

1. Perform preliminary settings

Creatio and Visual Studio setup for development in the file system is described in the “**Development in the file system**” and “**IDE settings for development**” articles.

For development in the file system, you can use Microsoft Visual Studio Community, Professional and Enterprise version 2017 (with latest updates) and higher.

2. Create, obtain or update a package from the SVN repository

Creating a custom package with or without SVN is described in the “**Creating a package for development**” and “**Creating a package in the file system development mode**” articles. Installing and updating packages is described in the “[Installing packages from repository](#)” and “[Updating package from repository](#)” articles.

We recommend using [Tortoise SVN](#) or [Git](#) for working with version control repositories.

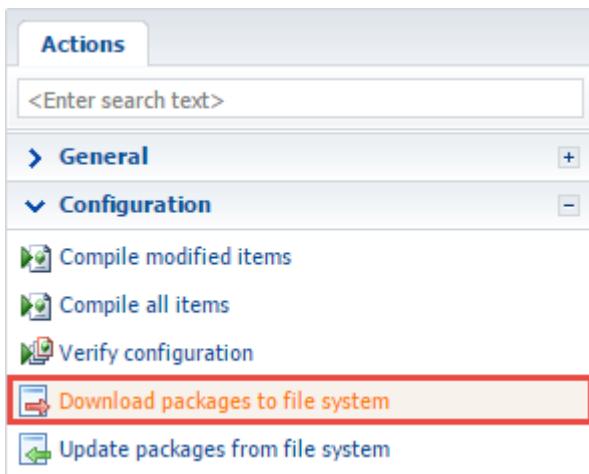
3. Create a [Source code] schema

Learn more about the process of creating the [Source code] schema in the “**Creating the [Source code] schema**” article.

4. Conduct development in the Visual Studio

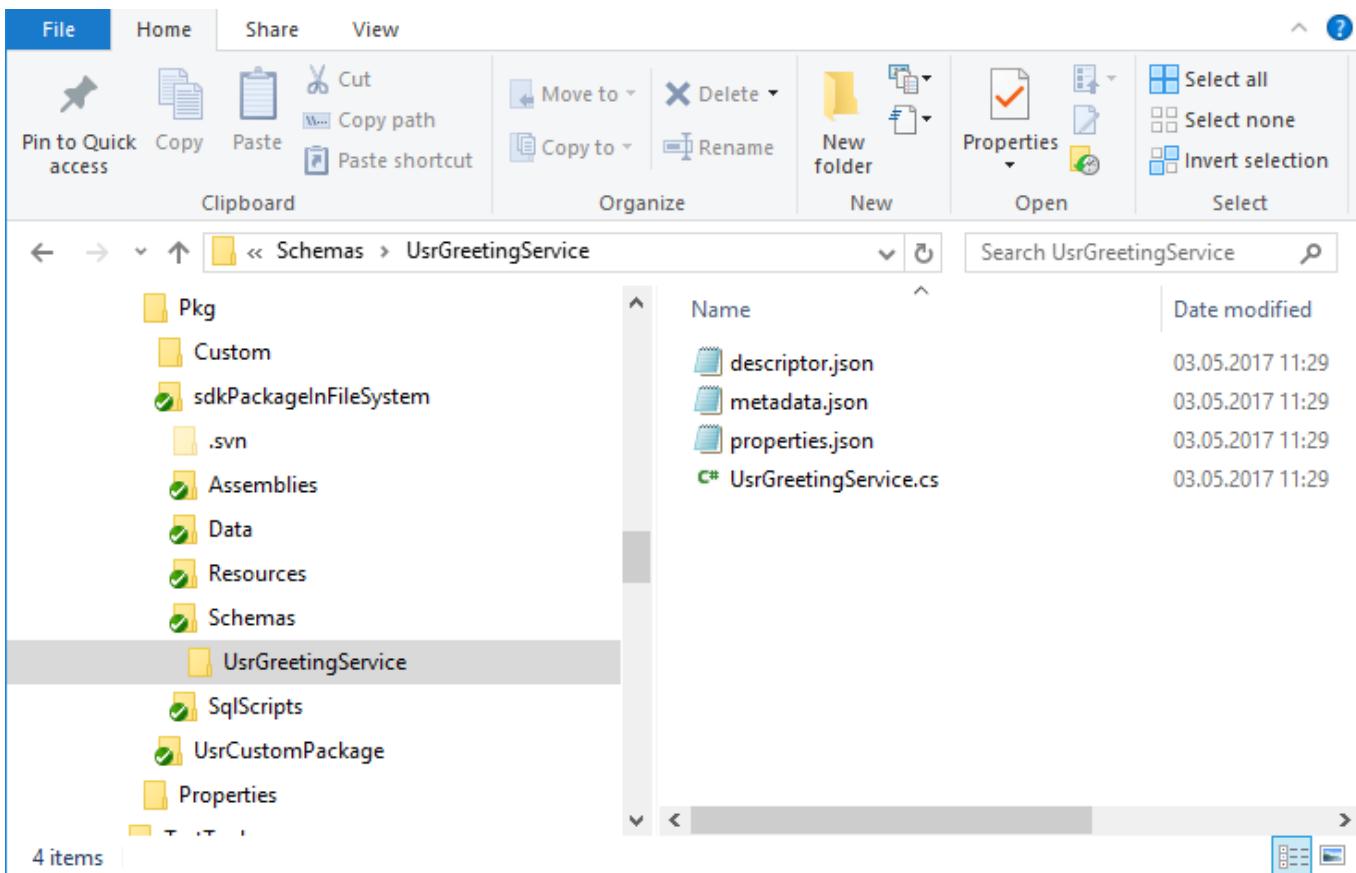
Before you start development in Visual Studio, make sure that you export existing schemas from the database to the file system.

Fig. 1 The [Download packages to file system] action



For example, if the [Source code] schema with the name *UsrGreetingService* was created in the *sdkPackageInFileSystem* package, the file of the source code of the *UsrGreetingService.cs* schema appears in the file system in the *Pkg|sdkPackageInFileSystem|Schemas|* directory (Fig. 2). In this case, the system generated *UsrGreetingServiceSchema.sdkPackageInFileSystem_Entity.cs* file will be placed in the *Autogenerated|Src* directory.

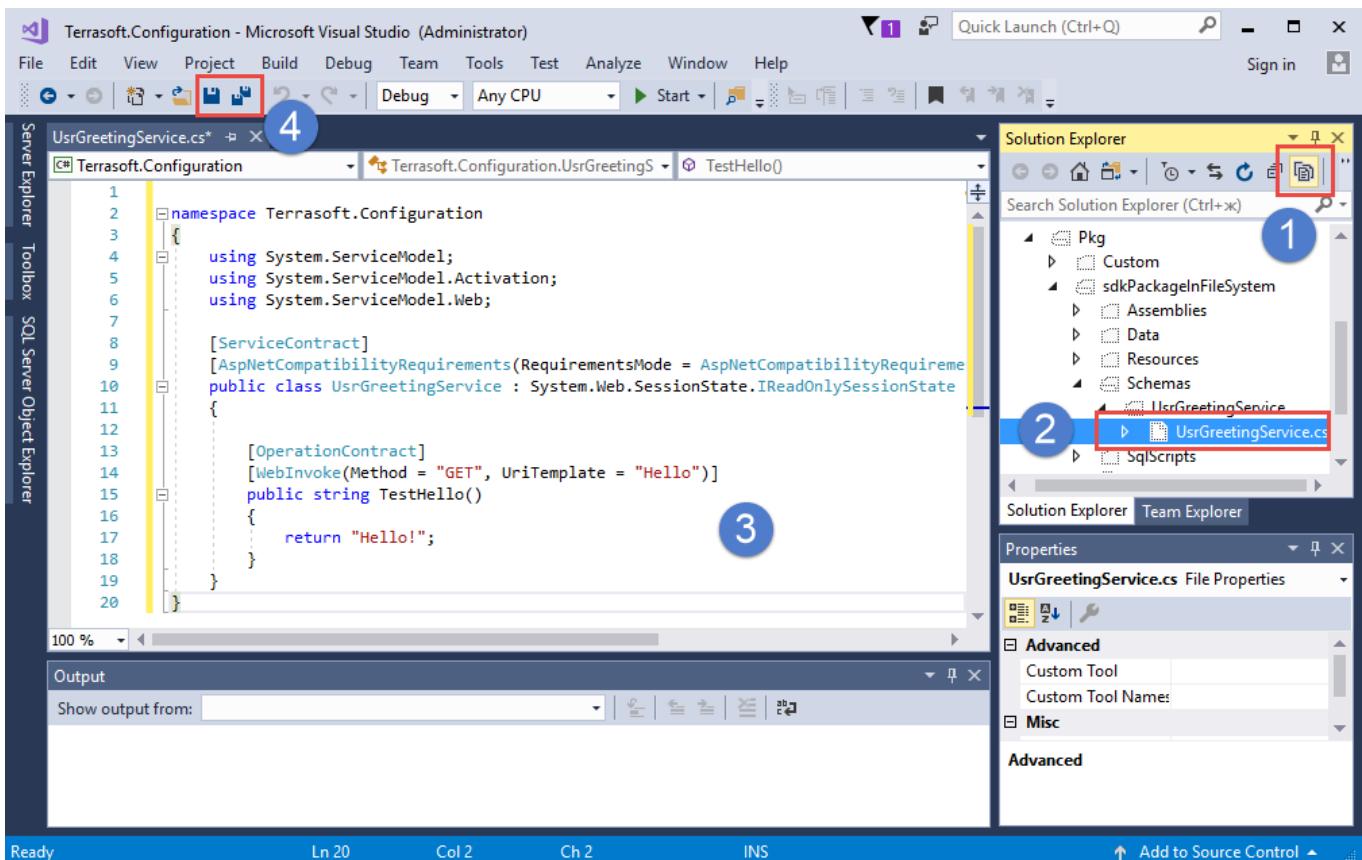
Fig. 2 The source code schema file



To add a schema to SVN, you must add the entire *UsrGreetingService* directory, including the JSON files.

Open the *Terrasoft.Configuration.sln* solution in Visual Studio to start the development (see “**Development in the file system**”). In Visual Studio Solutions Explorer, enable the display of all file types (Fig. 3, 1), open the *UsrGreetingService.cs* file (Fig. 3, 2) and add the required source code (Fig. 3, 3).

Fig. 3 Working with the schema file in Visual Studio



Below is an example of source code implementation, which must be added to the contents of the *UsrServiceGreeting.cs* file, using Visual Studio:

```
namespace Terrasoft.Configuration
{
    using System.ServiceModel;
    using System.ServiceModel.Activation;
    using System.ServiceModel.Web;
    // Class that implements configuration service.
    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Required)]
    public class UsrGreetingService : System.Web.SessionState.IReadOnlySessionState
    {
        // Service operation.
        [OperationContract]
        [WebInvoke(Method = "GET", UriTemplate = "Hello")]
        public string TestHello()
        {
            return "Hello!";
        }
    }
}
```

For more information about the purpose of the attributes of the class that implements configuration services, please refer to the “**Creating a user configuration service (on-line documentation)**”.

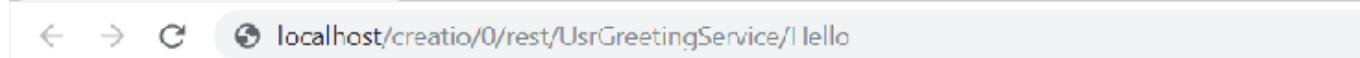
5. Save, compile and debug source code

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

After saving, the source code must be compiled with the “Build Workspace” or “Rebuild Workspace” commands (see “**IDE settings for development**”). If the compilation is successful, the code becomes available. In the example described previously, the service will become available at the following address (Fig. 4):

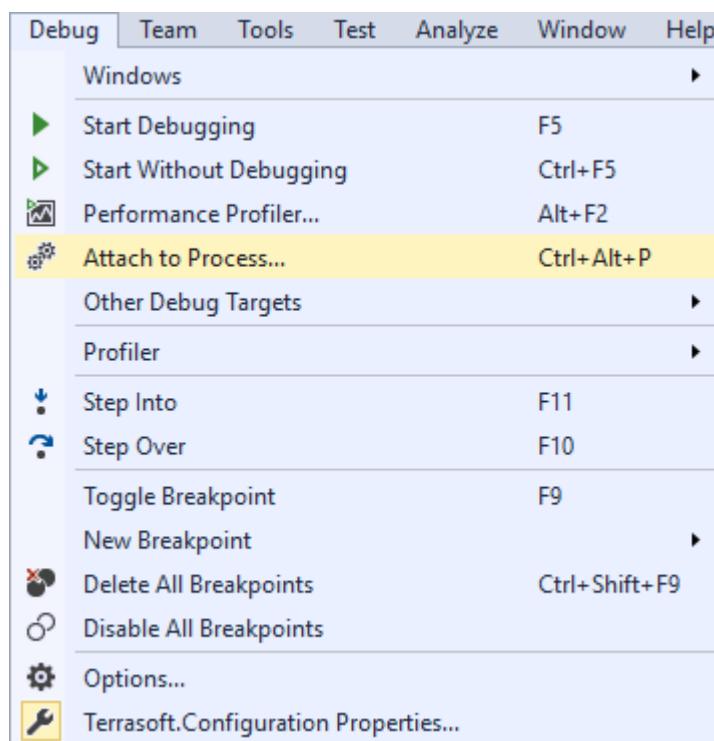
`http://[Application URL]/0/rest/UsrGreetingService>Hello`

Fig. 4 Checking service workability



To begin debugging, attach to the IIS server process, where the application runs. Execute the *Debug > Attach to process* menu command (Fig. 5).

Fig. 5 The [Attach to process] command



In the opened window, select the working IIS process in the list of processes, where the application pool is running (Fig. 6).

Fig. 6 Attaching to an IIS process



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

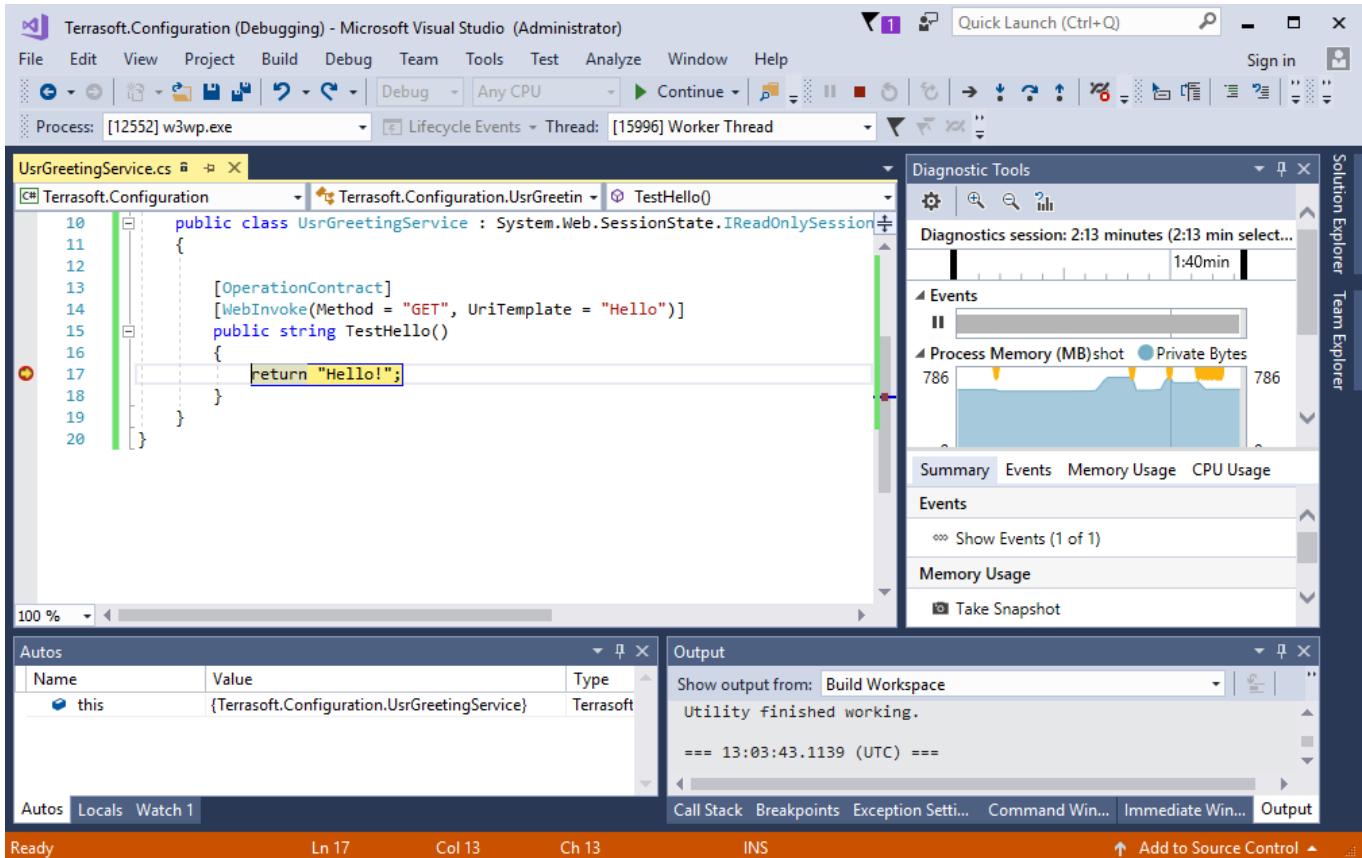
By default, the IIS working process is run under the account whose name matches the name of the application pool. To display processes of all users, set *Show processes from all users* checkbox (Fig. 6).

After attaching to a working IIS process, execute compilation one more time. After that, begin the debugging process using the Visual Studio debugger. For example, you can set the stop points, view variable values, call stacks, etc. For more information on the Visual Studio debugger, please refer to the [corresponding documentation](#).

For example, after setting up the breakpoint on the return line from the `TestHello()` method, and re-compiling the

application and executing the service request, the debugger will stop the program execution on the breakpoint (Fig. 7).

Fig. 7 Stopping an application on the breakpoint



The debugging feature depends on the **correct configuration of Visual Studio**.

Developing the configuration server code in the user project

[Beginner](#)

[Easy](#)

[Medium](#)

[Advanced](#)

Introduction

Before the 7.11.1 version, only the preconfigured Visual Studio solution which is distributed with Creatio, was used to develop configuration server code in the file system. More information about the Terrasoft.Configuration.sln solution and development of the server code in the file system is given in the "[Development in the file system](#)" and "[Working with the server side source code](#)" articles.

This development approach is inconvenient because of low performance connected with recompilation of all Creatio configuration (Terrasoft.Configuration.dll). This is significantly if the application contains several Creatio products. In addition, the development of server code in the file system could only be performed by interacting with the database of the application deployed on-site.

Due to the described inconveniences, this approach can be efficiently used to perform complex configuration revision of the Creatio. It is more efficient to use the built-in Creatio development tools to develop a simple server code (see the "[Built-in development tools](#)" article). But the built-in development tools do not support full IDE functions: debugging, Intellisense, Refactoring, etc.

Starting with version 7.11.1 the Creatio you can develop simple server code in the Visual Studio custom projects.

To develop and debug separate classes or small blocks of server functionality, you can create a separate class library project and configure it. Then, connect corresponding Creatio class libraries (for example, Terrasoft.Core) and perform development and debugging of the server code. To debug and test the development result you can use a local database (use the WorkspaceConsole utility to connect to the database) or an application located in the cloud

by connecting to it via the Executor utility.

Advantages of this approach:

- High speed of testing modifications, compiling and execution
- Full usage of the Visual Studio functions
- Ability of using any tools for [Continuous Integration](#), for example Unit testing.
- Simplicity of configuration – you do not need configuration source codes
- You can use the database of an application deployed on-site or in Cloud.

Preliminary settings

For connecting the libraries of the Creatio classes, deploying the local database from an archive copy and working with the WorkspaceConsole utility, you can use the Creatio installed locally. In all examples of this article used the Creatio installed to the *C:\creatio* local folder.

The Executor utility located in the *C:\Executor* folder is used as an example of working with the Creatio Cloud service. You can use following [link](#) to download the utility configured for processing the example.

Development of the configuration server code for on-site application

If you have an access to the Creatio local database, to develop configuration server logic do the following:

1. Restore the database from a backup (if need)

The process of restoring the Creatio database from backup is described in the "[Installing Creatio application](#)" article. Backup of the application database is located in the *db* folder of the application (for example, *C:\creatio\db*).

2. Configure the WorkspaceConsole utility

To operate with the database, you need to configure the WorkspaceConsole utility using the application files. More information about configuration of the utility is described in the "[WorkspaceConsole settings](#)" article. To configure the utility:

- Open the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder of the application (for example, *C:\creatio\Terrasoft.WebApp\DesktopBin\WorkspaceConsole*).
- Execute one of the .bat files: *PrepareWorkspaceConsole.x64.bat* or *PrepareWorkspaceConsole.x86.bat*, depending on the Windows version.

Ensure that the *SharpPlink-*xXX.svnExe** and *SharpSvn-DB44-20-*XXX.svnDll** files were copied to the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder from the corresponding folder (x64 and x86) after executing the .bat file.

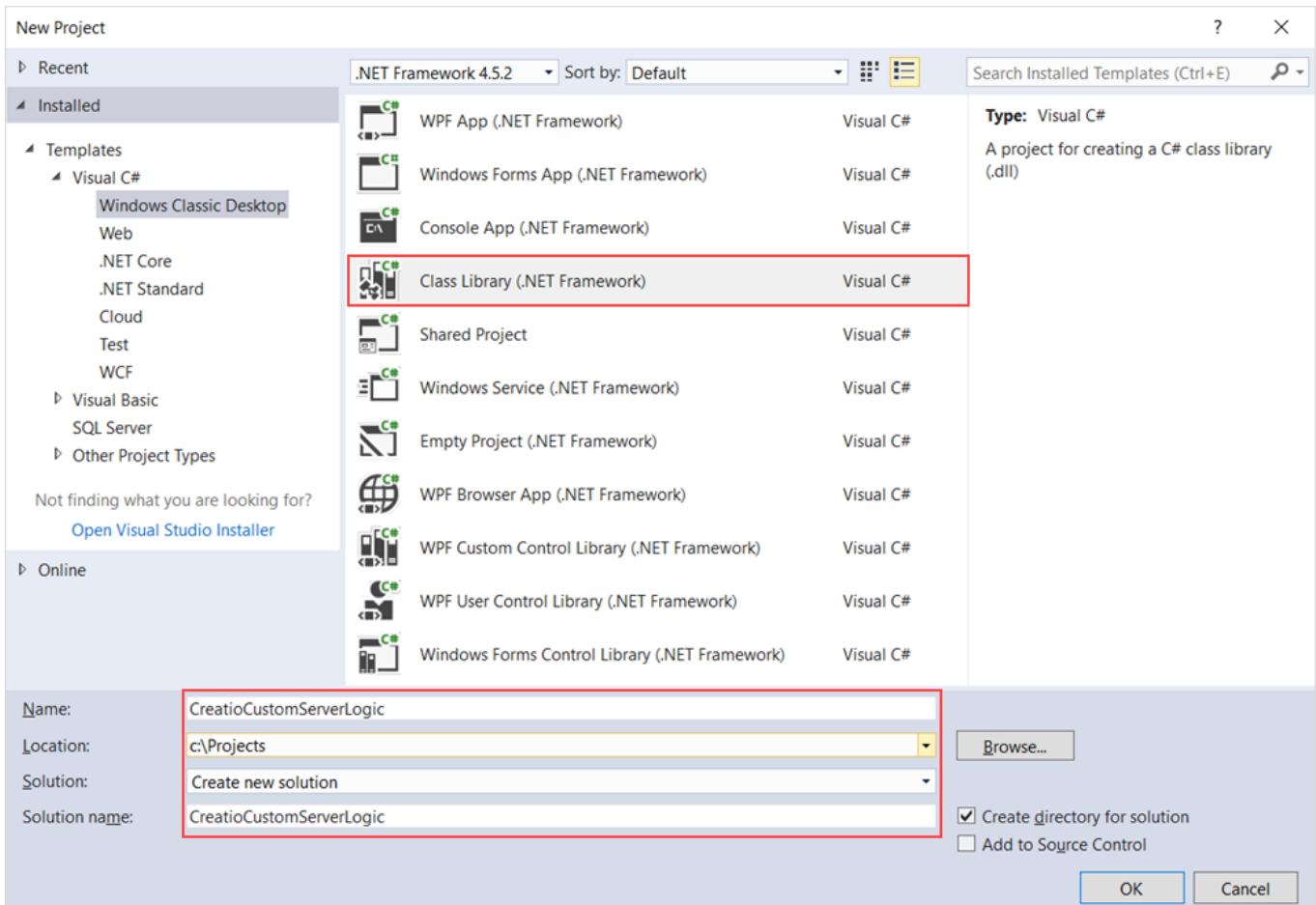
- Specify parameters of connection to the database in the *Terrasoft.Tools.WorkspaceConsole.exe.config* file from the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder of the application (for example, *C:\creatio\Terrasoft.WebApp\DesktopBin\WorkspaceConsole*). For example, if the *creatioDB* database is deployed on the *dbserver* server, the connection string will be as follows:

```
<connectionStrings>
<add name="db" connectionString="Data Source=dbserver; Initial Catalog=creatioDB;
Persist Security Info=True; MultipleActiveResultSets=True; Integrated Security=SSPI;
Pooling = true; Max Pool Size = 100; Async = true; Connection Timeout=500" />
</connectionStrings>
```

3. Create and configure Visual Studio project

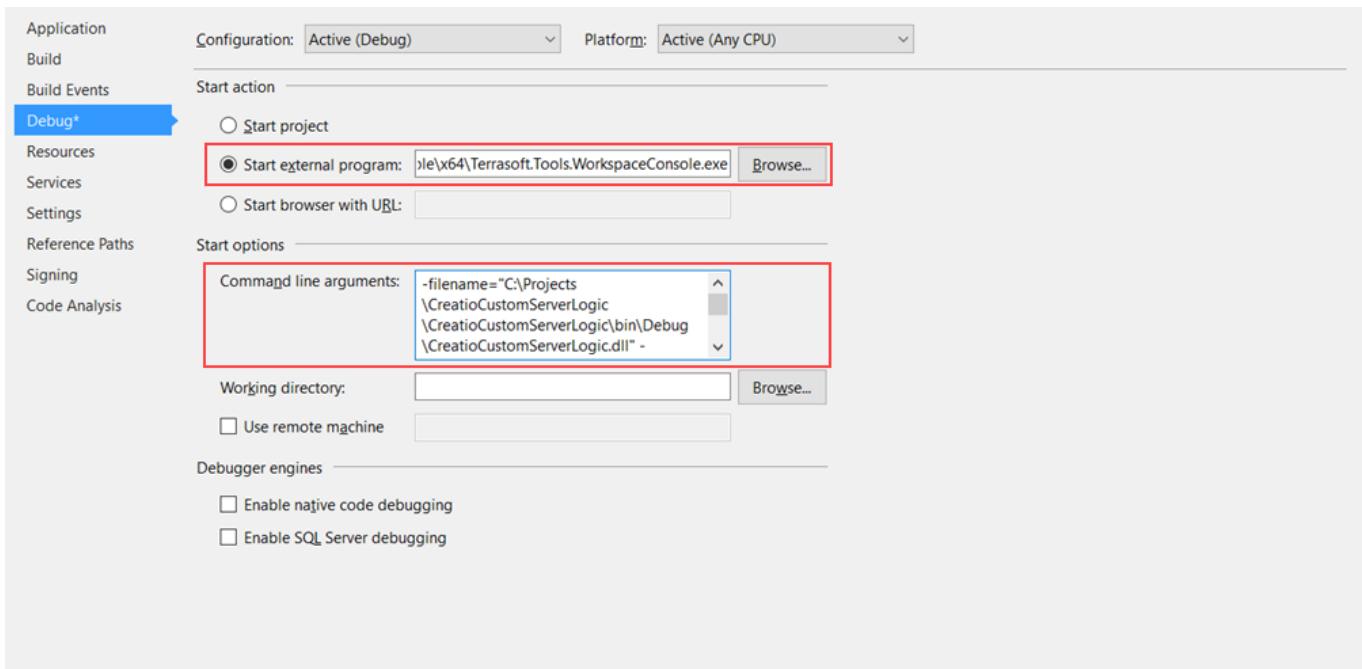
For this, create standard class library project (Fig. 1). More information about creating a new Visual Studio solution and managing projects is described in the "[Solutions and Projects in Visual Studio](#)" Microsoft documentation article.

Fig. 1. Creating the solution and project of the classes library in the Visual Studio



On the [Debug] tab of the properties window of the created class library project, specify the full path to the configured WorkspaceConsole utility in the [Start external program] property (Fig. 2). The WorkspaceConsole is used as the external application for debugging the developed program logic.

Fig. 2. The [Debug] tab properties



In the [Command line arguments] properties specify following launch arguments of the WorkspaceConsole:

- *-filename* – full path to the debug version of developed class library.
- *-typeName* – full name of the class in which the program logic (including the names of all namespaces) is being developed. For example, *CreatioCustomServerLogic.MyContactCreator*.
- *-operation* – WorkspaceConsole operation. The "ExecuteScript" value should be specified.
- *workspaceName* – the workspace name. The "Default" value should be specified.
- *-confRuntimeParentDirectory* – path to the parent catalog of the *conf* directory (see "**Client static content in the file system**" and "**Server content in the file system (on-line documentation)**"). This directory is usually located in the *Terrasoft.WebApp* catalog of the deployed application.

The example of the WorkspaceConsole launch arguments:

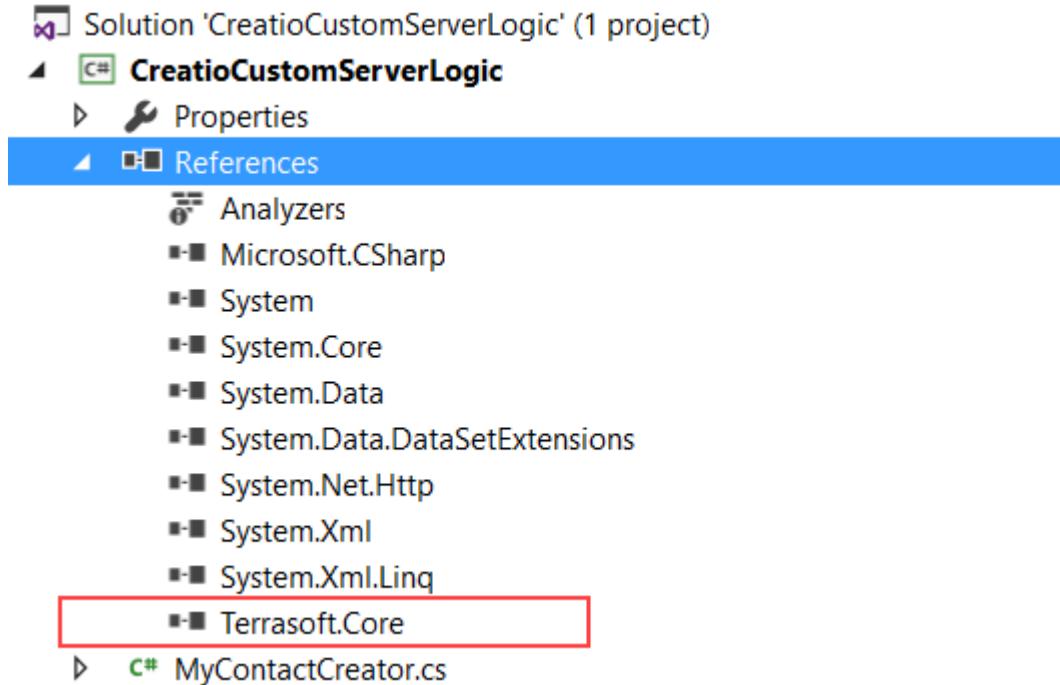
```
filename="C:\Projects\CreatioCustomServerLogic\CreatioCustomServerLogic\bin\Debug\CreateioCustomServerLogic.dll" -typeName=CreatioCustomServerLogic.MyContactCreator -operation=ExecuteScript -workspaceName=Default -logPath=C:\Projects\Logs -confRuntimeParentDirectory=C:\creatio\Terrasoft.WebApp -autoExit=true
```

More information can be found in the "**WorkspaceConsole parameters**" article.

In the properties of the Visual Studio project that operates with the application 7.11.0 or higher, you need to specify the version of the .NET Framework 4.7 (the [Target framework] property of the [Application] tab).

To work with the classes of the server side of Creatio core, set the dependencies from the necessary Creatio class libraries in the created project. For example, add the dependency from the Terrasoft.Core.dll library (Fig. 3). More information about adding the dependencies can be found in the "[Managing references in a project](#)" Microsoft documentation article.

Fig. 3. Terrasoft.Core library in the project dependencies



Class libraries of the Creatio namespace can be found in the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder of the application.

Class libraries are being copied to the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder when executing the .bat files (see Step 2. Configure the WorkspaceConsole utility").

4. Develop the functions

For this, add a new class to the created class library project. The name of the class should match the name specified in the *typeName* launch argument of the WorkspaceConsole (for example,

CreatioCustomServerLogic.MyContactCreator). Class should implement the *Terrasoft.Core.IExecutor* interface.

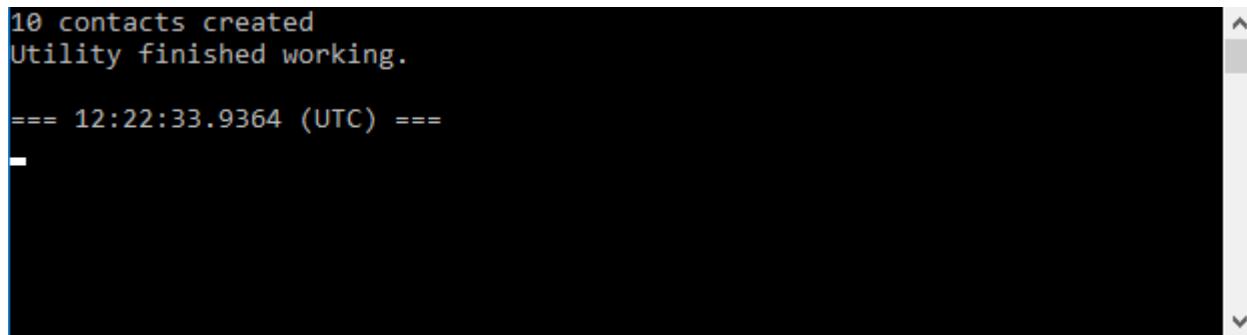
The implementation of the class is available below:

```
using System;
using Terrasoft.Core;

namespace CreatioCustomServerLogic
{
    public class MyContactCreator : IExecutor
    {
        public void Execute(UserConnection userConnection)
        {
            // Getting an instance of the [Contacts] schema.
            var schema =
userConnection.EntitySchemaManager.GetInstanceByName("Contact");
            var length = 10;
            for (int i = 0; i < length; i++)
            {
                // Create a new contact.
                var entity = schema.CreateEntity(userConnection);
                // Set contact properties.
                entity.SetColumnValue("Name", string.Format("Name {0}", i));
                entity.SetDefColumnValues();
                // Save the contact to the database.
                entity.Save(false);
            }
            // Output message to the console.
            Console.WriteLine($"{length} contacts created");
        }
    }
}
```

After running the project (F5 key) the WorkspaceConsole window with the corresponding message will be displayed (Fig. 4).

Fig. 4. Displaying the result of running the program in the WorkspaceConsole window.



The screenshot shows a black terminal window with white text. It displays the following output:
10 contacts created
Utility finished working.
--- 12:22:33.9364 (UTC) ---
-
The window has scroll bars on the right side.

You can also set a breakpoint on any line of the source code and view the current values of variables at the time of program execution (ie, debugging). More information about breakpoints in the Visual Studio can be found in the [Use Breakpoints in the Visual Studio Debugger](#) Microsoft documentation article.

The result of execution the above code can be found in the [Contacts] section of the Creatio application (Fig. 5) or by executing the request to the database (Fig. 6).

Fig. 5. Added contacts

Name	Last Activity
Name 0	3/30/2020 6:16 PM
Name 1	3/30/2020 6:16 PM
Name 2	3/30/2020 6:16 PM
Name 3	3/30/2020 6:16 PM
Name 4	3/30/2020 6:16 PM
Name 5	3/30/2020 6:16 PM
Name 6	3/30/2020 6:17 PM
Name 7	3/30/2020 6:17 PM

Fig. 6. Request to the table of contacts of the database

```

1 select Name from Contact
2 where Name like 'Name%'

```

	Name
1	Name 0
2	Name 1
3	Name 2
4	Name 3
5	Name 4
6	Name 5
7	Name 6

Development of the configuration server code for Cloud application

To develop configuration server logic without direct access to the Creatio database:

1. Create class library project.

Create standard class library project (Fig. 1). More information about creating a new Visual Studio solution and managing projects is described in the "[Solutions and Projects in Visual Studio](#)" Microsoft documentation article. Set the name of the project (for example, "CreatioCustomServerLogic.Cloud").

To work with the classes of the server side of Creatio core, set the dependencies from the necessary Creatio class libraries in the created project. For example, add the dependency from the Terrasoft.Core.dll library (Fig. 3). More information about adding the dependencies can be found in the "[Managing references in a project](#)" Microsoft documentation article.

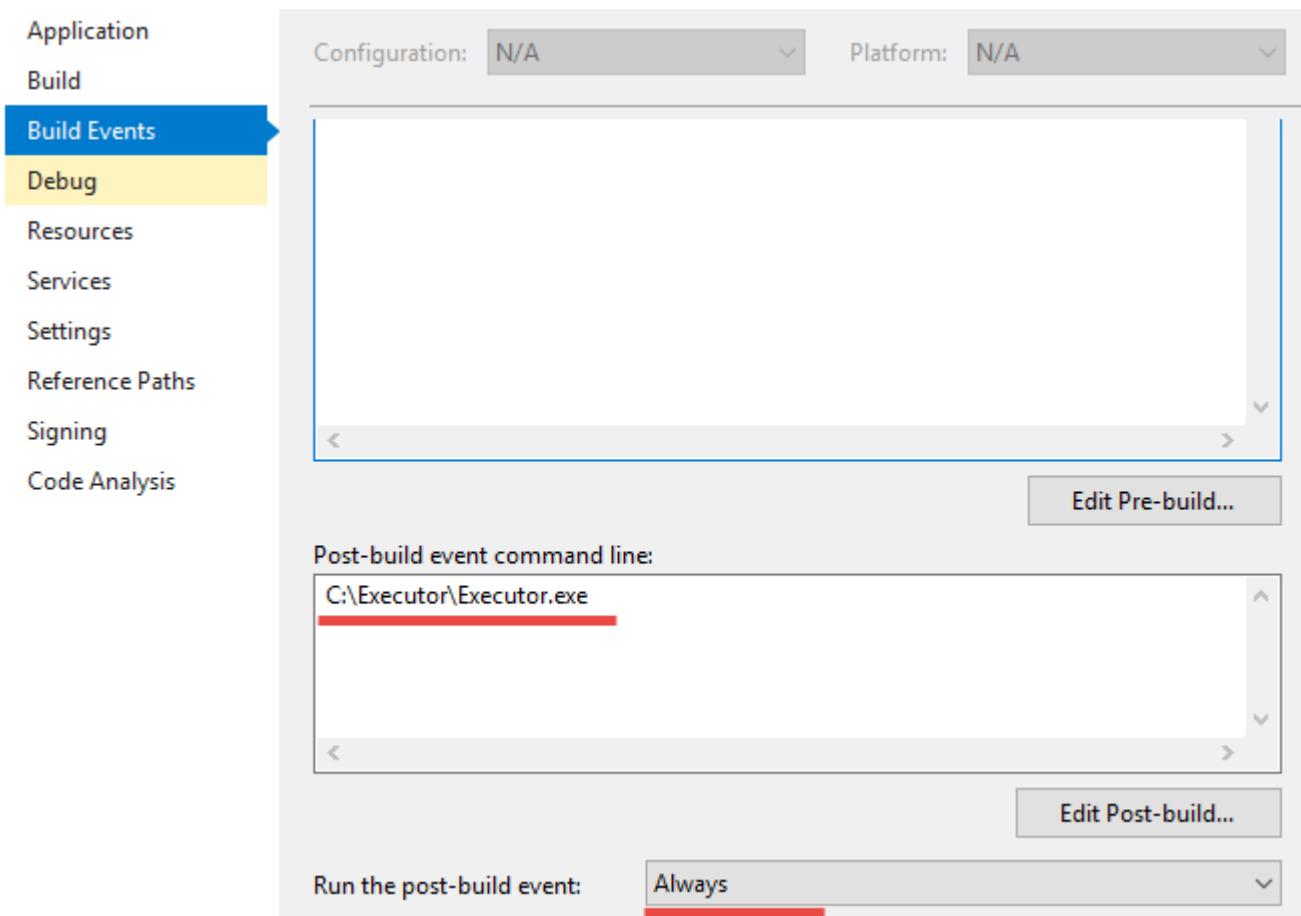
Class libraries of the Creatio namespace can be found in the *bin* folder of the application. Class libraries are being copied to the *Terrasoft.WebApp\DesktopBin\WorkspaceConsole* folder when executing the .bat files (see Step 2. "Configure the WorkspaceConsole utility" of the example of configuration server code development for on-site application).

In the created class library project, specify the full path to the configured Executor utility in the [Post-build event command line] property on the [BuildEvents] tab of the properties window (Fig. 7), for example

C:\Executor\Executor.exe. Also, you must select the condition for starting the library build event on this tab.

The configuration process is given below on the Step "3. Executor utility configuration".

Fig. 7. [Build Events] tab properties



2. Develop the functions

For this, add the class that will implement the *Terrasoft.Core.IExecutor* interface to the created library project. The implementation of the class is available below:

```
using System;
using System.Web;
using Terrasoft.Core;
using Terrasoft.Core.Entities;

namespace CreatioCustomServerLogic.Cloud
{
    public class MyContactReader : IExecutor
    {
        public void Execute(UserConnection userConnection)
        {
            // Getting an instance of the [Contacts] schema.
            var entitySchema =
userConnection.EntitySchemaManager.GetInstanceByName("Contact");
            // Create an instance of the query class.
            var esq = new EntitySchemaQuery(entitySchema);
            // Adding all the columns of the schema to the query.
            esq.AddAllSchemaColumns();
            // Getting the collection of records in the [Contacts] section.
            var collection = esq.GetEntityCollection(userConnection);
```

```
foreach (var entity in collection)
{
    // The output in the http-response of the request from the Executor
    // utility of the necessary values.
    HttpContext.Current.Response.Write(entity.GetTypedColumnValue<string>
("Name"));
    HttpContext.Current.Response.Write(Environment.NewLine);
}
}
```

3. Executor utility configuration

You can use following [link](#) to download the utility configured for processing the example.

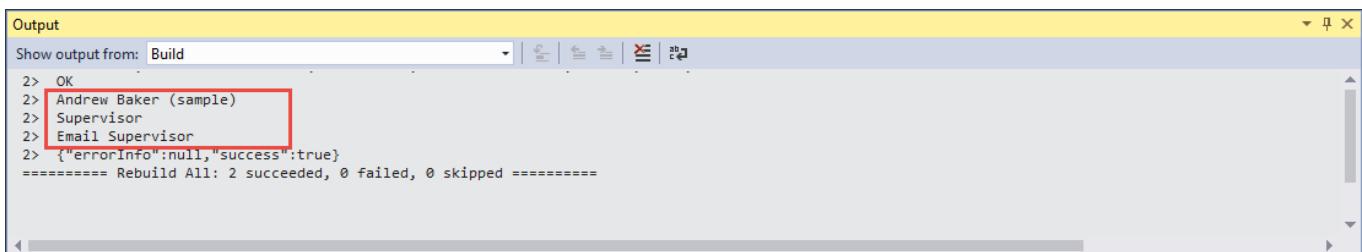
Open the Executor utility folder, for example, *C:\Executor*. Then, specify the values for the following configuration items in the configuration file:

- Loader – URL of the Creatio application loader. Usually this is the URL of the Creatio site, for example "<https://mycloudapp.creatio.com>".
- WebApp – URL of the Creatio application. Usually this is a path to default configuration of Creatio, for example "<https://mycloudapp.creatio.com/o>".
- Login – the name of the Creatio user, for example, "Supervisor".
- Password – the password of Creatio user.
- LibraryOriginalPath – the path to the initial copy of the class library. Usually, this is the path by which a class library is created after compilation in Visual Studio, for example, "C:\Projects\CreatioCustomServerLogic\CreatioCustomServerLogic.Cloud\bin\Debug\CreatioCustomServerLogic.Cloud.dll".
- LibraryCopyPath – the path by which a copy of the class library will be created for work with the remote server. This can be a temporary folder that contains the Executor utility, for example, "C:\Executor\CreatioCustomServerLogic.Cloud.dll".
- LibraryType – full name of the class in which the developed program logic is implemented, including the names of all namespaces. For example, "CreatioCustomServerLogic.Cloud.MyContactReader".
- LibraryName – name of the class library, for example, "CreatioCustomServerLogic.Cloud.dll".

4. Run the developed program code

The result of execution the development program code can be observed in the [Output] window of the Visual Studio after successful building of the class library (Fig. 8).

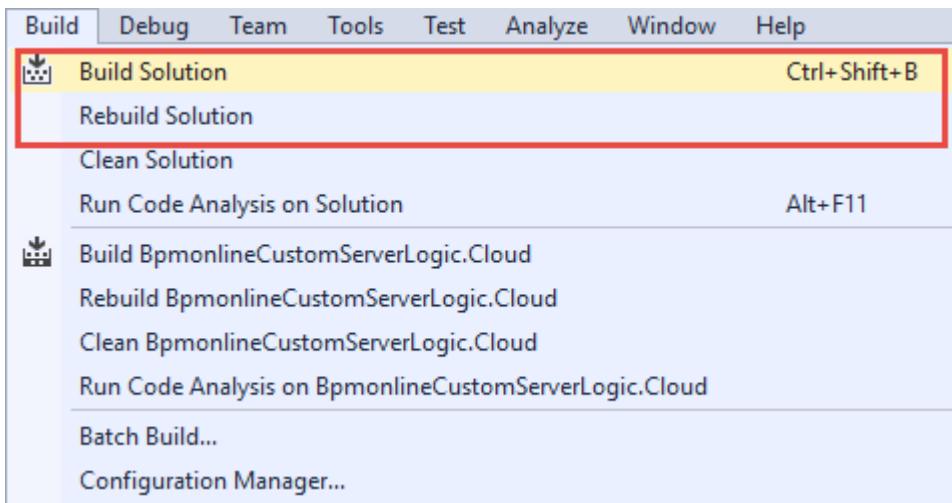
Fig. 8. Result of program code execution



The screenshot shows the Visual Studio Output window with the title bar 'Output'. The dropdown menu 'Show output from' is set to 'Build'. The window displays the following text:
2> OK
2> Andrew Baker (sample)
2> Supervisor
2> Email Supervisor
2> {"errorInfo":null,"success":true}
===== Rebuild All: 2 succeeded, 0 failed, 0 skipped =====
A red box highlights the line 'Andrew Baker (sample)'.

To launch the building process use the [Build Solution] and [Rebuild Solution] menu commands (Fig. 9).

Fig. 9. [Build] menu commands



Working with the client code

Contents

- **Introduction**
- **Automatic displaying of changes**

Working with the client code

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

Introduction

The updated method of working with the client code in the file system enables better development flexibility. Download the client schema source code from the database to *.js files and LESS module styles into *.less files for working with them in the Integrated Development Environment (IDE) (e.g. WebStorm, Visual Studio Code, Sublime Text, etc.).

General outline:

1. Pre-configure Creatio.
2. Create, obtain or update a package from the SVN repository.
3. Create a [Source code] schema.
4. Save the database content to the file system.
5. Carry out the development of the schema source code in IDE.
6. Save, compile and debug the source code.

General outline:

1. Pre-configure Creatio

Setting up Creatio for development in the file system is described in the “**Development in the file system**” article.

The *UseFileContent* attribute of the *clientUnits* element in the *Web.config* file (located in the application folder) was used to upload the client module source code to the file system up until Creatio version 7.10.0. Since version 7.10.0, the *UseFileContent* attribute has been removed from the *Web.Config* file. Use the method described in the “**Development in the file system**” to upload the client module source code from pre-installed packages (the *Autogenerated\Src* folder description).

2. Create, obtain or update a package from the SVN repository

Creating a custom package with or without SVN is described in the “**Creating a package for development**” and

“[Creating a package in the file system development mode](#)” articles. Installing and updating packages is described in the “[Installing packages from repository](#)” and “[Updating package from repository](#)” articles.

We recommend using [Tortoise SVN](#) or [Git](#) to work with version control repositories.

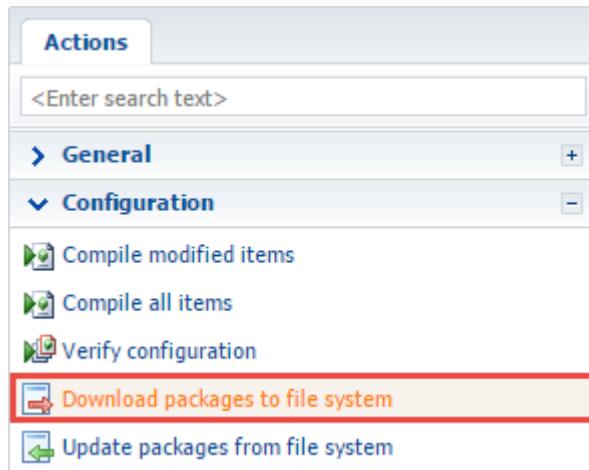
3. Create a custom schema for development

Learn more about custom schemas in the “[Creating a custom client module schema](#)” article.

4. Upload the schema from the database to the file system

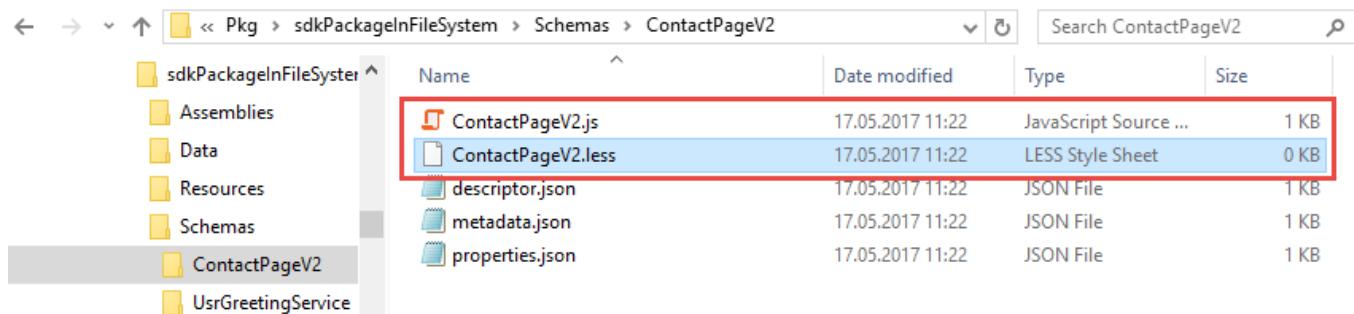
To do this, use the [Download packages to file system] command (Fig. 1) in the [Configuration] section.

Fig. 1. The [Download packages to file system] command



For example, if you created a replacing *ContactPageV2* schema ([Display schema - Contact card]) in a custom *sdkPackageInFileSystem* package, the files in the *Pkg|sdkPackageInFileSystem|Schemas|ContactPageV2* folder will contain the source code files of the *ContactPageV2.js* schema and *ContactPageV2.less* styles (Fig. 2).

Fig. 2. The source code schema file



5. Carry out the development of the schema source code in IDE

To perform the development, open the file with the schema source code in the preferred IDE (or any text editor) and add the necessary source code (Fig. 3).

Fig. 3. Editing a schema file in Visual Studio Code

The screenshot shows the Visual Studio Code interface with the following details:

- Title Bar:** ContactPageV2.js - Terrasoft.Configuration - Visual Studio Code
- File Menu:** File Edit Selection View Go Help
- Explorer:** Shows the project structure under TERRASOFT.CONFIGURATION:
 - OPEN EDITORS: ContactPageV2.js
 - TERRASOFT.CONFIGURATION:
 - Autogenerated
 - Lib
 - Pkg:
 - Custom
 - sdkPackageInFileSystem:
 - Assemblies
 - Data
 - Resources
 - Schemas:
 - ContactPageV2:
 - ContactPageV2.js (selected)
 - ContactPageV2.less
 - descriptor.json
 - metadata.json
 - properties.json
- Code Editor:** ContactPageV2.js


```

1 define("ContactPageV2", []);
2     function() {
3         return {
4             entitySchemaName: "Contact",
5             diff: /**SCHEMA_DIFF*/[
6                 {
7                     "operation": "remove",
8                     "name": "JobTitleProfile"
9                 }
10            ]/**SCHEMA_DIFF*/
11        );
12    };
13
      
```
- Status Bar:** Ln 1, Col 1 Tab Size: 4 UTF-8 CRLF JavaScript

For example, add the following source code to the *ContactPageV2.js* file to hide the [Full job title] field from the contact edit page:

```

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

6. Save, compile and debug the source code

The [Full job title] will be removed from the contact edit page upon saving the *ContactPageV2.js* file and refreshing the page (Fig. 4).

Fig. 4. Contact page without the [Full job title] field



100%

⌚ 5:26 AM,
Boston

Full name*

Andrew Baker

Mobile phone

+1 617 221 5187

Business phone

+1 617 440 2031

Email

a.baker@ac.com

Debug the code if you encounter any errors (see: “**Client code debugging**”).

To return to built-in Creatio development tools, do the following:

1. Update packages from file system.
2. Disable the file system development mode by setting the `enabled="false"` attribute of the `fileDesignMode` element of the `Web.config` configuration file (see “**Development in the file system**”).

Working with the client code. Automatic displaying of changes

Beginner Easy Medium **Advanced**

Introduction

When developing configuration server code in the file system, each time after making changes to the source code of the custom schema you need to refresh the browser page on which the application is opened. This reduces the development performance.

To avoid this, we developed the new functionality of automatic browser page reload after changes. This functionality works in a following way.

When the application starts, it creates an object that tracks the changes of the `.js` file with the source code of the developed module in the file system. If the changes have made, a message is sent to the client Creatio application. In the client application, a specific object which is signed to this message defines dependent objects of the changed module, destroys them, registers new paths to the modules and tries to load the modified module again. After that, all the pre-initialized modules will be requested by the browser via new paths and load changes from the file system. It does not take time to interpret and load other modules. Separate development page enables to avoid loading of additional modules (for example left or right panel, communication panel, etc.). This reduces the number of requests to the server.

This approach reveals the connectivity of the modules and detects unnecessary dependencies to eliminate them.

Known issues

1. If there is a syntax error in the source code of the module, the page will not automatically refresh. The page will need to be forcibly refreshed (for example, by pressing the F5 key). If the error is corrected, the page will return to the operable status.

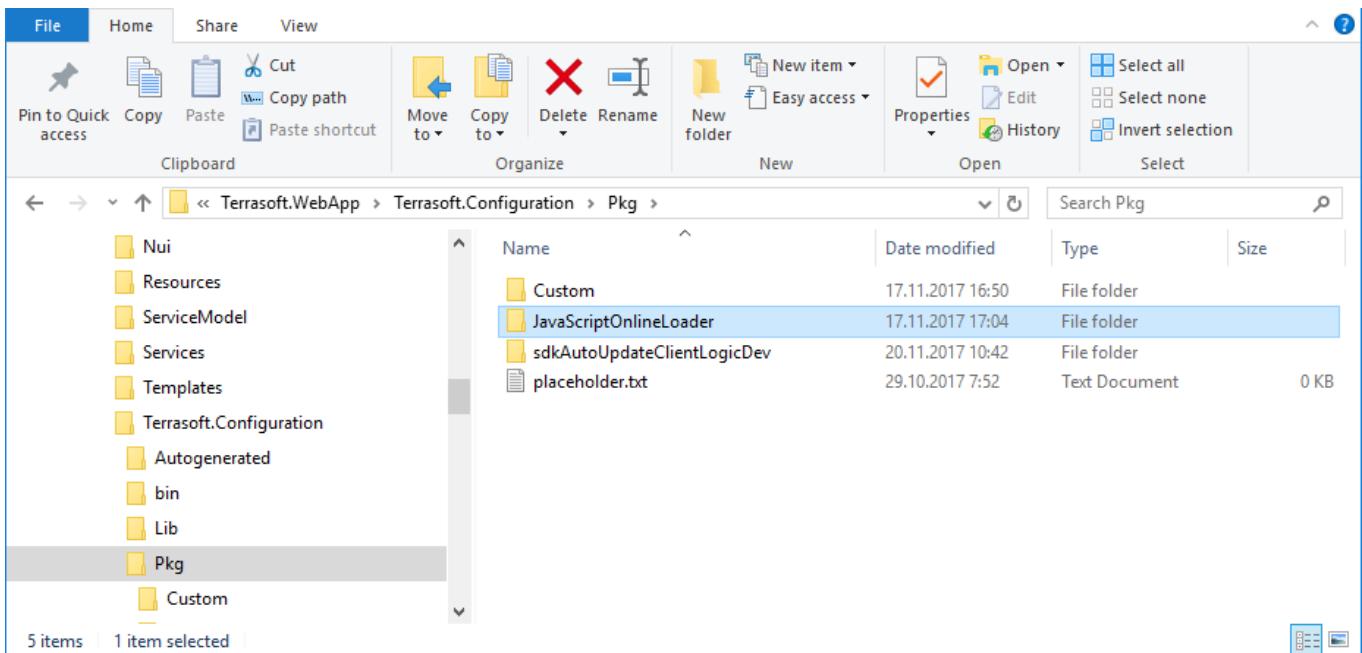
- Not all Creatio modules can be downloaded separately. The main reason is the effect of strong [coupling of modules](#).

Configuration steps

1. Install the JavaScriptOnlineLoader package

Enable the development mode in the file system and add the *JavaScriptOnlineLoader* folder with corresponding package to the *[Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg* folder (Fig. 1).

Fig. 1. The *JavaScriptOnlineLoader* package in the file system

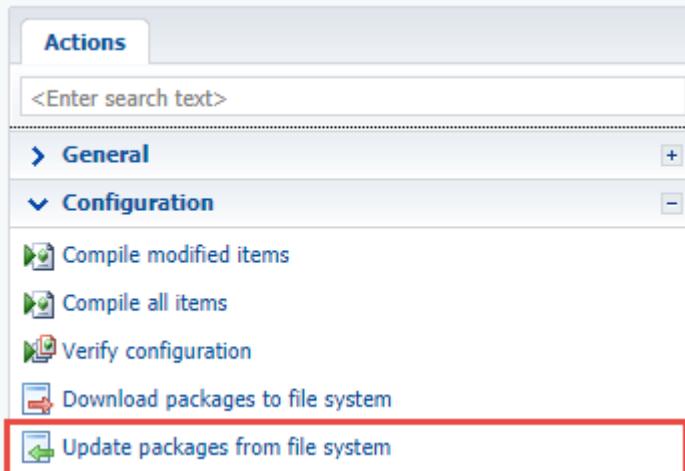


More information about the development mode in the file system can be found in the “[Development in the file system](#)” article.

The package is available on the GitHub (<https://github.com/vladimir-nikonov/pngstore/tree/master/JavaScriptOnlineLoader>). Also the archive with the package can be downloaded by the [link](#).

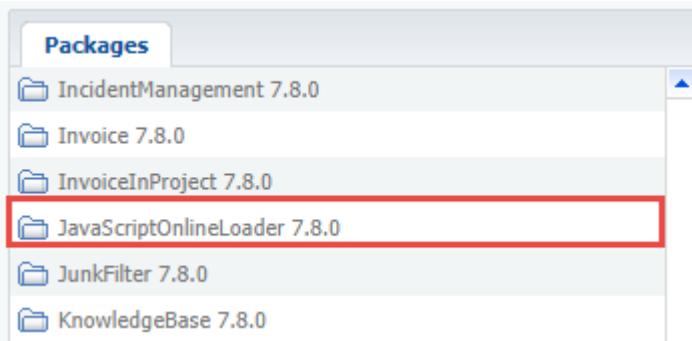
Load the package to the configuration with the [Update packages from file system] action (Fig. 2).

Fig. 2. The [Update packages from file system] action



As a result, the package will be displayed on the [Packages] tab (Fig. 3).

Fig. 3. The package in the [Configuration] section



2. Open the page of the developed module in the browser

To do this, open the *ViewModule.aspx* page with the added parameter with the following format:

```
?vm=DevViewModule#CardModuleV2/<Module name>
```

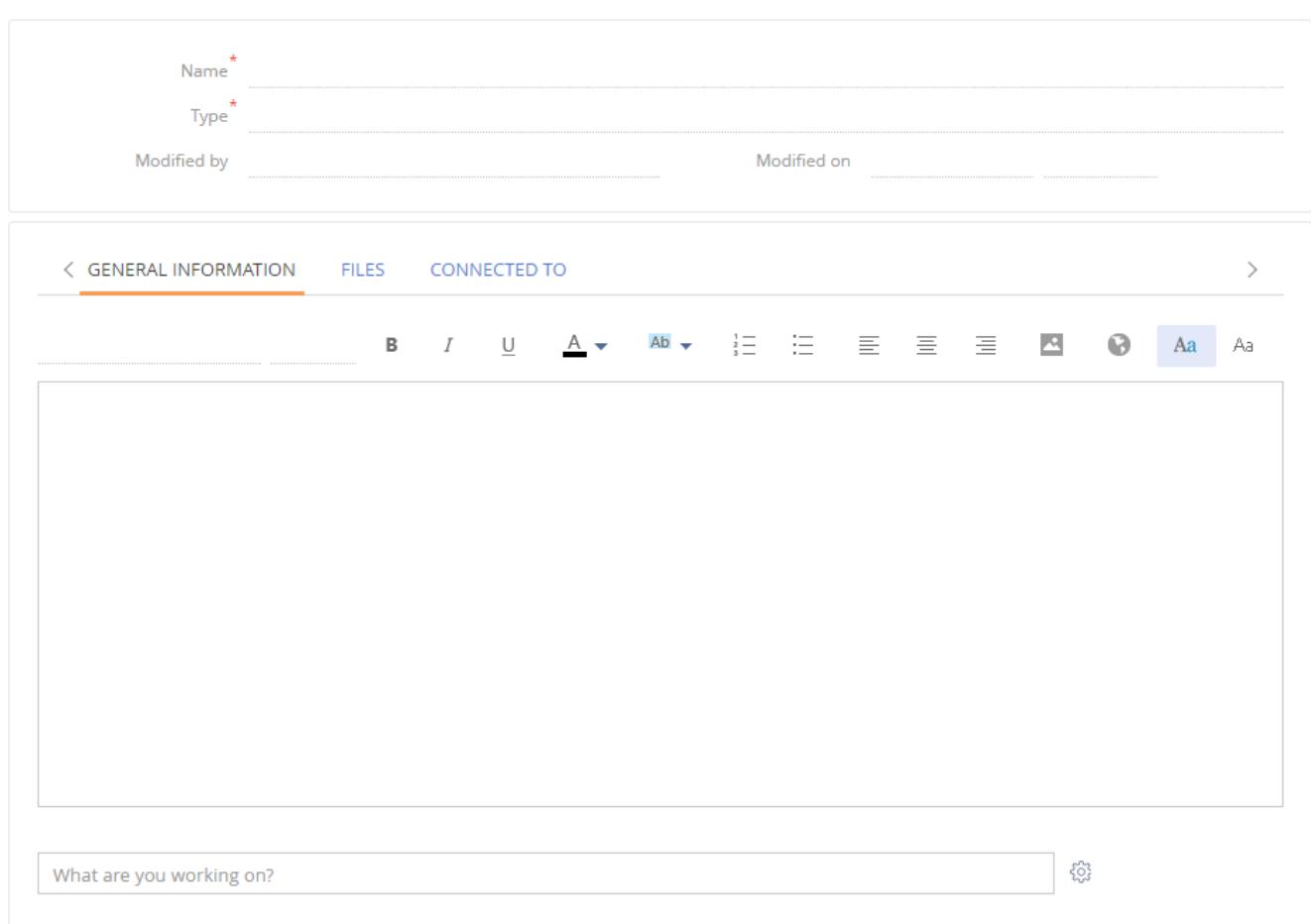
For example, the *KnowledgeBasePageV2* replacing schema (the schema of the [Knowledge base] section edit page) is added to the custom package. The page with the functions of automatic displaying of changes will be available at the following URL:

```
http://localhost/creatio/0/Nui/ViewModule.aspx?  
vm=DevViewModule#CardModuleV2/KnowledgeBasePageV2
```

The *http://localhost/creatio* is a URL of the Creatio application deployed on-site.

After clicking this URL, the *ViewModule.aspx* page will be displayed with the loaded module (Fig. 4).

Fig. 4. The *ViewModule.aspx* page with the loaded module



3. Change the source code of the developed schema

The source code of the developed schema can be changed in any text editor (for example, the Notepad). After saving the changes, the page opened in the browser will be automatically refreshed.

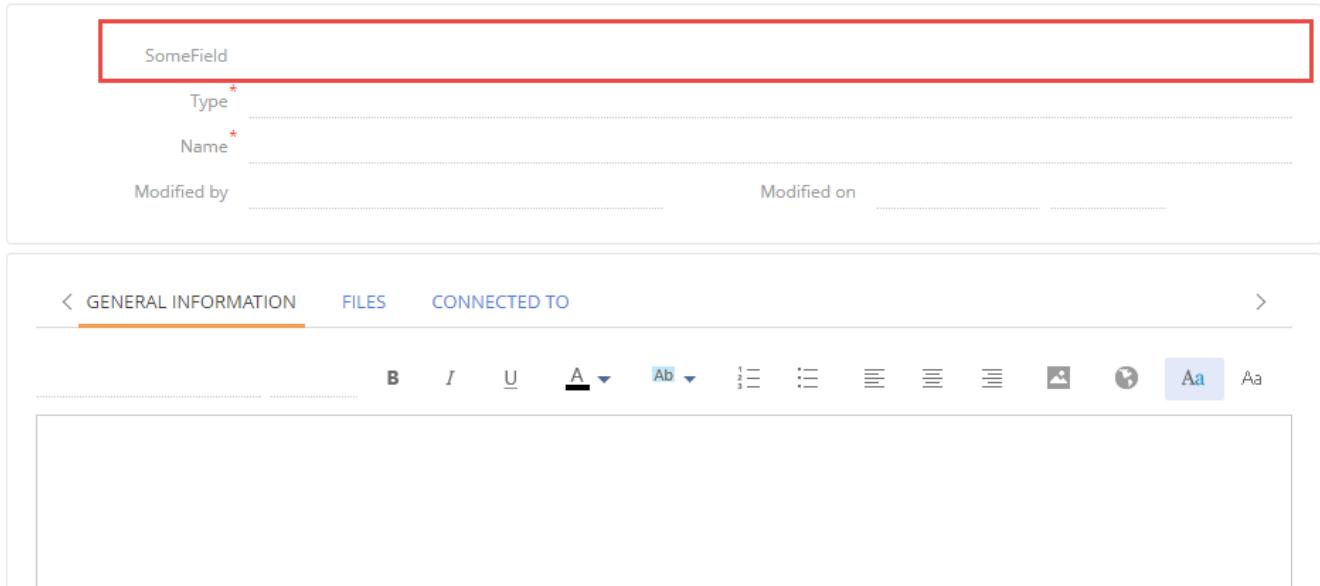
For example, the *KnowledgeBasePageV2* replacing schema (the schema of the [Knowledge base] section edit page) is added to the *sdkAutoUpdateClientLogicDev* custom package. After loading to the file system, the schema code will be available in the ..\Pkg\sdkAutoUpdateClientLogicDev\schemas\KnowledgeBasePageV2 folder.

If the following source code will be added to the *KnowledgeBasePageV2.js* file and save it, the browser page will be automatically refreshed. The changes will be displayed immediately (Fig. 5).

```
define("KnowledgeBasePageV2", [],  
    function() {  
        return {  
            entitySchemaName: "KnowledgeBase",  
            diff: /**SCHEMA_DIFF*/ [  
                {  
                    "operation": "insert",  
                    "parentName": "Header",  
                    "propertyName": "items",  
                    "name": "SomeField",  
                    "values": {  
                        "layout": {"column": 0, "row": 0, "colSpan": 24},  
                        "caption": "SomeField"  
                    }  
                }  
            ] /**SCHEMA_DIFF*/  
        }  
    }  
)
```

```
    };  
});
```

Fig. 5. Page with changes



File content

Contents

- **Packages file content**
- **Localization of the file content**
- **Client static content in the file system**
- **How to use TypeScript when developing custom functions**
- **Creating an Angular component to use in Creatio**

Packages file content

Beginner Easy Medium Advanced

Introduction

Starting with version 7.11.3 you can add file content (.js, .css files, images, etc.) to the custom packages.

File content of packages is a number of any files used by the application. File content is static and is not processed by the web server (see “**Client static content in the file system**”). This increases application performance.

File content is an integral part of the Creatio and is always stored in the
...|Terrasoft.WebApp|Terrasoft.Configuration|Pkg|<Package name>|Files folder.

Any files can be added to the package, but only the files needed for the client part of Creatio will be used.

You need to generate auxiliary files (see “Generation of auxiliary files” below) to use file content.

Recommended file storage structure

To use file content the *Files* folder was added to the package structure (see “**Package structure and contents**”). It is recommended to keep following structure of the *Files* folder:

```
-PackageName  
  ...
```

```
-Files
  -src
    -js
      bootstrap.js
      [other *.js files]
    -css
      [* .css files]
    -less
      [* .less files]
    -img
      [image files]
    -res
      [resource files]
    descriptor.json
  ...
descriptor.json
```

Here

js – folder with .js files of JavaScript source codes

css – folder with *.css style files

less – folder with *.less style files

img – folder with images

res – folder with resource files

descriptor.json – descriptor of the file content.

How to add a new file content to the package

Copy a file to the corresponding subfolder of the *Files* folder of specific package. The *Files* folder will be available by the ...|Terrasoft.WebApp|Terrasoft.Configuration|Pkg|<Package name>|Files path.

Descriptor of the file content

Information about bootstrap files of the package is stored in the descriptor.json file of the Files folder. The has following structure:

```
{
  "bootstraps": [
    ... // An array of strings containing relative paths to bootstrap files.
  ]
}
```

Example of descriptor.json:

```
{
  "bootstraps": [
    "src/js/bootstrap.js",
    "src/js/anotherBootstrap.js"
  ]
}
```

Bootstrap files of the package

The .js files that enable to manage loading of client configuration logic. The file does not have a clear structure.

```
(function() {
  require.config({
    paths: {
      "Module name ":" A link to the file content",
      ...
    }
  })
})()
```

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

Example of bootstrap.js:

```
(function() {
    require.config({
        paths: {
            "MyPackage1-ContactSectionV2": Terrasoft.getFileContentUrl("MyPackage1",
"src/js/ContactSectionV2.js"),
            "MyPackage1-Utilities": Terrasoft.getFileContentUrl("MyPackage1",
"src/js/Utilities.js")
        }
    });
})();
```

All bootstrap files are loaded asynchronously after the core is loaded, but before loading the configuration.

Loading of the bootstrap files

For correct loading of bootstrap files, the _FileContentBootstraps.js auxiliary file is generated in the static content folder (see “Generation of auxiliary files” below). This file contains information about bootstrap files of all packages.

Example of the _FileContentBootstraps.js:

```
var Terrasoft = Terrasoft || {};
Terrasoft.configuration = Terrasoft.configuration || {};
Terrasoft.configuration.FileContentBootstraps = {
    "MyPackage1": [
        "src/js/bootstrap.js"
    ]
};
```

File content versioning

For correct versioning of the file content, the _FileContentDescriptors.js auxiliary file is generated in the static content folder (see “Generation of auxiliary files” below). This file contains information about the files in the file content of all packages in the “key-value” collection view. Each key (file name) corresponds to a unique hash code. This guarantees downloading of the up to date version of the file to the browser.

After installing the file content, there is no need to clear the browser cache.

Example of the _FileContentDescriptors.js file:

```
var Terrasoft = Terrasoft || {};
Terrasoft.configuration = Terrasoft.configuration || {};
Terrasoft.configuration.FileContentDescriptors = {
    "MyPackage1/descriptor.json": {
        "Hash": "5d4e779e7ff24396a132a0e39cca25cc"
    },
    "MyPackage1/Files/src/js/Utilities.js": {
        "Hash": "6d5e776e7ff24596a135a0e39cc525gc"
    }
};
```

Generation of auxiliary files

Execute the BuildConfiguration operation in the WorkspaceConsole:

```
Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -
workspaceName=Default -destinationPath=Terrasoft.WebApp\ -
configurationPath=Terrasoft.WebApp\Terrasoft.Configuration\ -
useStaticFileContent=false -usePackageFileContent=true -autoExit=true
```

In this code:

- *operation* – operation name. *BuildConfiguration* – operation of configuration compilation.
- *useStaticFileContent* – a flag of using static content. Should be *false*.
- *usePackageFileContent* – a flag of using file content of the packages. Should be *true*.

Other WorkspaceConsole parameters are described in the "**WorkspaceConsole parameters**" article.

As a result the *_FileContentBootsraps.js* and *_FileContentDescriptors.js* auxiliary files will be generated in the folder with the static content ...|*Terrasoft.WebApp*|*conf*|*content*.

Also the generation of auxiliary files is performed at installation of packages from SVN and executing compilation action in the [Configuration] section.

Transition of modifications between environments

File content is an integral part of the package. The content is stored in the SVN store with all package content. The content can be transferred to another development environment via SVN (see "**Working with SVN in the file system**").

It is recommended to use Creatio built-in tools to transfer on test and production environments (see "**Transferring changes using packages export and import**" and "[Installing marketplace applications from a zip archive](#)").

Localization of the file content

Beginner

Easy

Medium

Advanced

Introduction

Starting with version 7.11.3 you can add file content (.js, .css files, images, etc.) to the custom packages.

File content of packages is a number of any files used by the application. File content is static and is not processed by the web server (see "**Client static content in the file system**"). This increases application performance.

More information about file content can be found in the "**Packages file content**".

Localization with configuration resources

To translate the resources it is recommended to use separate module with localizable resources created via internal Creatio tools in the **[Configuration]** section. The complete source code of this module is available below:

```
define("Module1", ["Module1Resources"], function(res) {
    return res;
});
```

To include localizable resources to the module that is defined in the file content of the package you need to define the module with resources. Example:

```
define("MyPackage-MyModule", ["Module1"], function(module1) {
    console.log(module1.localizableStrings.MyString);
});
```

Localization via i18n plugin

i18n is a plugin for AMD loader (for example, RequireJS) used for loading localizable string resources. The source code of the plugin can be found in the <https://github.com/requirejs/i18n> storage. Documentation is available by the <http://requirejs.org/docs/api.html#i18n> link.

To localize file content with RequireJS i18n plugin, perform the following steps:

1. Add the plugin to the folder with the source code .js files:

..|*Terrasoft.WebApp*|*Terrasoft.Configuration*|*Pkg*|*MyPackage1*|*content*|*js*|*i18n.js*.

MyPackage1 – working folder of the *MyPackage1* package (see "**Packages file content**").

2. Create the ..|*MyPackage1*|*content*|*nls* folder and put there one or several .js files with localizable resources. File

names can be arbitrary. File content – AMD modules with objects of the following structure:

- The “root” field contains the key-value collection where the “key” is the name of a localizable string and the “value” is localizable string of the default language. The value will be used if the requested language is not supported.
- Fields with the names of standard cultures (for example, “en-US”, “de-DE”) and the boolean value. The value is *true* if the supported culture is enabled and *false* if it is disabled.

For example the added ..\MyPackage1\content\js\nls>ContactSectionV2Resources.js file with the following content:

```
define({
  "root": {
    "FileContentActionDescr": "File content first action (Default)",
    "FileContentActionDescr2": "File content second action (Default)"
  },
  "en-US": true,
  "ru-RU": true
});
```

3. In the ..\MyPackage1\content\nls folder, create folders with the names corresponding to the cultures of the localization files that will be put in these folders (for example, “en-US”, “de-DE”). For example, if the German and English culture are supported the folder structure will be following:

```
content
  nls
    en-US
    ru-RU
```

4. In each created localization directory put the same number of .js files as in the ..\MyPackage1\content\nls root folder. File content is the AMD modules with objects of the key-value collections, where the “key” is the name of a localizable string and the “value” is a string of the language corresponding to the name of the folder (the code of the culture).

For example, if the German and English culture are supported you need to create two ContactSectionV2Resources.js files. The content of the ..\MyPackage1\content\js\nls\en-US>ContactSectionV2Resources.js, file corresponding to English culture:

```
define({
  "FileContentActionDescr": "File content first action",
  "FileContentActionDescr2": "File content second action"
});
```

The content of the ..\MyPackage1\content\js\nls\de-DE>ContactSectionV2Resources.js, file corresponding to German culture:

```
define({
  "FileContentActionDescr": "Die erste Aktion des Dateiinhalts"
});
```

As the translation of the “FileContentActionDescr2” string is not specified for the German culture the default value (“File content second action (Default)”) will be used.

5. Edit the bootstrap.js file.

- Connect the i18n plugin by specifying its name as the “i18n” alias in the RequireJS path configuration and specifying corresponding path to the plugin in the *paths* property.
- For the plugin specify a culture that is current for the user. Set the object with the *i18n* property to the *config* property of the configuration object of the RequireJS library. Set the object with the *locale* property and the value received from the *Terrasoft.currentUserCultureName* (the code of the current culture) to the object with the *i18n* property.
- For each file with localization resources set corresponding aliases and paths in the RequireJS path configuration. The alias must be a URL-path relative to the *nls* directory.

Example of the ..\MyPackage1\content\js\bootstrap.js file content:

```
(function() {
    require.config({
        paths: {
            "MyPackage1-Utilities": Terrasoft.getFileContentUrl("MyPackage1",
"content/js/Utilities.js"),
            "MyPackage1-ContactSectionV2": Terrasoft.getFileContentUrl("MyPackage1",
"content/js/ContactSectionV2.js"),
            "MyPackage1-CSS": Terrasoft.getFileContentUrl("MyPackage1",
"content/css/MyPackage.css"),
            "MyPackage1-LESS": Terrasoft.getFileContentUrl("MyPackage1",
"content/less/MyPackage.less"),
            "i18n": Terrasoft.getFileContentUrl("MyPackage1", "content/js/i18n.js"),
            "nls/ContactSectionV2Resources":
Terrasoft.getFileContentUrl("MyPackage1",
"content/js/nls/ContactSectionV2Resources.js"),
            "nls/ru-RU/ContactSectionV2Resources":
Terrasoft.getFileContentUrl("MyPackage1", "content/js/nls/ru-
RU/ContactSectionV2Resources.js"),
            "nls/en-US/ContactSectionV2Resources":
Terrasoft.getFileContentUrl("MyPackage1", "content/js/nls/en-
US/ContactSectionV2Resources.js")
        },
        config: {
            i18n: {
                locale: Terrasoft.currentUserCultureName
            }
        }
    });
})();
```

6. Use the resources by specifying the corresponding module of resources with the "i18n!" alias in the dependency array. For example, to use the *FileContentActionDescr* (see steps 2,4) string as a title for the new action in the [Contacts] section, add the following content to the ..\MyPackage1\content\js\ContactSectionV2.js file:

```
define("MyPackage1-ContactSectionV2", ["i18n!nls/ContactSectionV2Resources",
"css!MyPackage1-CSS", "less!MyPackage1-LESS"], function(resources) {
    return {
        methods: {
            getSectionActions: function() {
                var actionMenuItems = this.callParent(arguments);
                actionMenuItems.addItem(this.getButtonMenuItem({ "Type": "Terrasoft.MenuSeparator" }));
                actionMenuItems.addItem(this.getButtonMenuItem({
                    "Click": { "bindTo": "onFileContentActionClick" },
                    "Caption": resources.FileContentActionDescr
                }));
                return actionMenuItems;
            },
            onFileContentActionClick: function() {
                console.log("File content clicked!")
            }
        },
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    }
});
```

Client static content in the file system

Beginner

Easy

Medium

Advanced

Introduction

Before the version 7.11, at the request of client content (.js, .css files), the application server generated the content dynamically, based on the current structure of package connections and schema dependencies. Generated data were cached and sent to client application.

Starting with version 7.11 all client content is preliminary generated in special application folder i. e. it becomes static. When requesting client content, the IIS searches for requested content in this folder and sends it to the client application. Thus, the overall performance of the application is increased and the server load is reduced.

Advantages and disadvantages

Advantages and disadvantages of using the client static content are given in the Table 1.

Table 1. Advantages and disadvantages of using the client static content

Advantages	Disadvantages
<i>Dynamic generation of client content</i>	
No need to pre-generate client content	Processor overload when computing the hierarchy of packages, schemas, and content generation
	Database overload when for getting the hierarchy of packages, schemas, and content generation
	Memory consumption for caching client content
<i>Usage of preliminary generated client content</i>	
Minimum CPU load (CPU)	Need to pre-generate client content
Missing database queries	
Client content is cached by IIS	

Generating static client content

Client content is generated in the specific folder (.\Terrasoft.WebApp\conf). It contains .js files with schema source code, .css files of styles and .js files of resources of all cultures of the application.

Starting with version 7.11.1 the .\Terrasoft.WebApp\conf folder also contains images.

The application's IIS pool user requires modify permission (reading and writing of files and subfolders and deletion of the folder) to the .\Terrasoft.WebApp\conf directory. Without the write permission Creatio application will not be able to generate static content.

The IIS pool user name is set in the [Identity] property. You can access this property through the [Advanced Settings] menu command on the [Application Pools] tab of the IIS Manager.

The actions to start generation of client content

Primary or secondary generation of static client content starts when the following actions are performed:

- Saving a schema through client schema designer and client objects designer.
- Saving through section wizard and detail wizard.
- Installing and deleting applications from Marketplace and zip archive.
- Applying translation.
- The [Compile all items] and [Compile modified items] actions in the [Configuration] section.

When deleting schemas and packages from the [Configuration] section you need to perform [Compile all items] and [Compile modified items] actions.

When installing and updating schemas and packages from the SVN you need to perform [Compile all items] action.

Only the [Compile all items] action performs full regeneration of client static content. Other actions lead only to regeneration of modified schemas.

Generation of client content with the WorkspaceConsole utility

The *BuildConfiguration* operation was added to the *WorkspaceConsole* utility and this operation performs generation of client content. Operation parameters are listed in table 2.

Table 2. Parameters of the BuildConfiguration operation

Parameter	Details
workspaceName	Workspace name by default (<i>Default</i>).
destinationPath	Folder to which the static content will be generated
webApplicationPath	Path to the web application from which the information about connection to database will be read. This parameter is optional. If this value has not been indicated, the connection will be established to the database specified in the connection string of the <i>Terrasoft.Tools.WorkspaceConsole.config</i> file. If the value is specified, the connection will be established with the database from the <i>ConnectionStrings.config</i> file of the web application.
force	If the value is set to <i>true</i> , the generation of the content will be performed for all schemas. If the value is set to <i>false</i> , the generation will be performed only for modified schemas. This parameter is optional. The value is <i>false</i> by default.

Use cases:

```
Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -  
workspaceName=Default -destinationPath="C:\WebApplication\Creatio\Terrasoft.WebApp" -  
force=true -logPath=C:\wc\log
```

```
Terrasoft.Tools.WorkspaceConsole.exe -operation=BuildConfiguration -  
workspaceName=Default -webApplicationPath="C:\WebApplication\Creatio" -  
destinationPath="C:\WebApplication\Creatio\Terrasoft.WebApp" -force=true -  
logPath=C:\wc\log
```

Compatibility with the development in the file system mode

Currently, the development in the file system is no compatible with getting client content from preliminary generated files. For the correct work of the development in the file system you need to disable getting static client content from the file system. Set the “false” for the *UseStaticFileContent* flag in the Web.config file to disable this functions.

```
<fileDesignMode enabled="true" />  
...  
<add key="UseStaticFileContent" value="false" />
```

Generation of client content when adding a new culture

Execute the [Compile all items] action in the [Configuration] section after adding new cultures.

If a user cannot log in to the system after adding new culture, you need to access the [Configuration] section by the `http://[path to application]/o/dev` path and execute the [Compile all items] action.

Changes in the parameter object that generates an image URL (version 7.11.1)

Images in the client part of Creatio are always being requested by a browser with a specific URL specified in the *src* attribute of the *img* html-element. The *Terrasoft.ImageUrlBuilder* (*imagurlbuilder.js*) module with the *getUrl(config)* public method that gets the image URL is used in the URL generation. This method receives the *config* configuration JavaScript object that contains an object of parameters in the *params* property. The image URL is being generated on the basis of this object.

Till the 7.11.0 version the structure of the *params* object had the following view:

```
config: {  
    params: {  
        schemaName: "",  
        resourceItemName: "",  
        hash: ""  
    }  
}
```

In this code:

- *schemaName* – schema name (string)
- *resourceItemName* – image name in the Creatio (string)
- *Hash* – image hash (string).

Starting with version 7.11.1 the *resourceItemExtension* string property that contains file extension (for example, .png) was added to the parameters list. A new structure of the *params* object:

```
config: {  
    params: {  
        schemaName: "",  
        resourceItemName: "",  
        hash: "",  
        resourceItemExtension: ""  
    }  
}
```

Starting with version 7.11.1 if the *params* object is generated in the custom program code (not obtained from the resources), the *resourceItemExtension* property should be added to the object. In the opposite case, the image will be retrieved from the database, not from the static content. In the next versions, the ability to retrieve an image from the database will be disabled. Therefore, the absence of the *resourceItemExtension* property will cause errors when loading images on a page.

An example of correct generation of configuration object of parameters for getting the URL of a static image:

```
var localizableImages = {  
    AddButtonImage: {  
        source: 3,  
        params: {  
            schemaName: "ActivityMiniPage",  
            resourceItemName: "AddButtonImage",  
            hash: "c15d635407f524f3bbe4f1810b82d315",  
            resourceItemExtension: ".png"  
        }  
    }  
}
```

How to use TypeScript when developing custom functions

Beginner

Easy

Medium

Advanced

Introduction

Starting with version 7.11.3 you can add file content (.js, .css files, images, etc.) to the custom packages.

File content of packages is a number of any files used by the application. File content is static and is not processed by the web server (see “**Client static content in the file system**”). This increases application performance.

More information about file content can be found in the “**Packages file content**”.

File content enables to use languages which can be compiled to JavaScript (for example TypeScript) in custom functions development. More information about TypeScript can be found at <https://www.typescriptlang.org>.

TypeScript installation

One way to install the TypeScript tools is to use the [NPM package manager](#) for the Node.js. For this, run the following command in the Windows console:

```
npm install -g typescript
```

Check the Node.js execution environment in your system, before installing TypeScript via the NMP. Download the installer by the <https://nodejs.org> link.

Case description

When saving an account record, display the message about the correctness of filling the [Also known as] field for the user. The field should contain only letters. Implement the validation logics in the TypeScript language.

Source code

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

Case implementation algorithm

1. Switch to the file system development mode

For more information about entering the file system development mode, see the “**Development in the file system**” article.

2. Create the structure of the file content storage

Recommended structure of the file content storage is described in the “**Packages file content**” article. For this:

1. Create the *Files* folder in the custom package loaded to the file system.
2. Add the *src* folder with the *js* subfolder to the *Files* folder.
3. Add the *descriptor.json* file with following content to the *Files* folder:

```
{  
    "bootstraps": [  
        "src/js/bootstrap.js"  
    ]  
}
```

4. Add the *bootstrap.js* file with the following content to the *Files|src|js* folder:

```
(function () {  
    require.config({  
        paths: {  
            "LettersOnlyValidator": Terrasoft.getFileContentUrl("sdkTypeScript",  
"src/js/LettersOnlyValidator.js")  
        }  
    });  
})();
```

The LettersOnlyValidator.js file specified in the bootstrap.js will be compiled at the step 4.

3. Implement the validation class in the TypeScript language

Create the *Validation.ts* file in the *Files|src|js* folder and declare the *StringValidator* interface in this file:

```
interface StringValidator {  
    isAcceptable(s: string): boolean;  
}  
export = StringValidator;
```

Create the *LattersOnlyValidator.ts* file in this folder. Declare the *LattersOnlyValidator* class in this file. The class will implement the *StringValidator* interface:

```
// Import the module in which the StringValidator interface is implemented.
import StringValidator = require("Validation");

// The created class must belong to the Terrasoft (module) namespace.
module Terrasoft {
    // Declaring the class of value validation.
    export class LettersOnlyValidator implements StringValidator {
        // A regular expression that allows the use of only letter characters.
        lettersRegexp: any = /^[A-Za-z]+$/;
        // Validating method.
        isAcceptable(s: string) {
            return !Ext.isEmpty(s) && this.lettersRegexp.test(s);
        }
    }
}
// Creating and exporting an instance of a class for require.
export = new Terrasoft.LettersOnlyValidator();
```

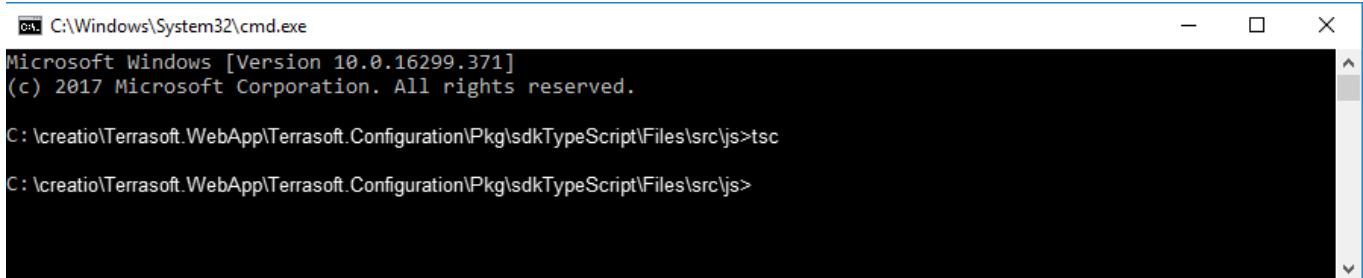
4. Compile the TypeScript source codes to the JavaScript source codes.

Add the `tsconfig.json` configuration file to the `Files\src\js` folder to set up the compilation:

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "amd",
        "sourceMap": true
    }
}
```

Go to the `Files\src\js` folder via the Windows console and execute the **tsc** command (Fig. 1).

Fig. 1. Execution of the tsc command



As a result of compilation the JavaScript version of the `Validation.ts` and `LetttersOnlyValidator.ts` files and the `.map` files facilitating debugging in the browser will be created in the `Files\src\js` folder (Fig. 2).

Fig. 2. Result of the tsc command execution

Pkg > sdkTypeScript > Files > src > js		
Name	Date modified	Type
bootstrap.js	09.05.2018 11:26	JavaScript File
LettersOnlyValidator.js	09.05.2018 14:12	JavaScript File
LettersOnlyValidator.js.map	09.05.2018 14:12	Linker Address Map
LettersOnlyValidator.ts	09.05.2018 13:58	TS File
tsconfig.json	08.05.2018 16:31	JSON File
Validation.js	09.05.2018 14:12	JavaScript File
Validation.js.map	09.05.2018 14:12	Linker Address Map
Validation.ts	08.05.2018 16:27	TS File

The content of the *LettersOnlyValidator.js* file that will be used in the Creatio (automatically generated):

```
define(["require", "exports"], function (require, exports) {
    "use strict";
    var Terrasoft;
    (function (Terrasoft) {
        var LettersOnlyValidator = /** @class */ (function () {
            function LettersOnlyValidator() {
                this.lettersRegexp = /^[A-Za-z]+$/;
            }
            LettersOnlyValidator.prototype.isAcceptable = function (s) {
                return !Ext.isEmpty(s) && this.lettersRegexp.test(s);
            };
            return LettersOnlyValidator;
        })();
        Terrasoft.LettersOnlyValidator = LettersOnlyValidator;
    })(Terrasoft || (Terrasoft = {}));
    return new Terrasoft.LettersOnlyValidator();
});
//# sourceMappingURL=LettersOnlyValidator.js.map
```

5. Perform the generation of auxiliary files

To generate the *_FileContentBootstraps.js* and *FileContentDescriptors.js* auxiliary files (see “**Packages file content**”):

1. Enter the [Configuration] section.
2. Load the package to the configuration with the [Update packages from file system] action.
3. Click the [Compile all items].

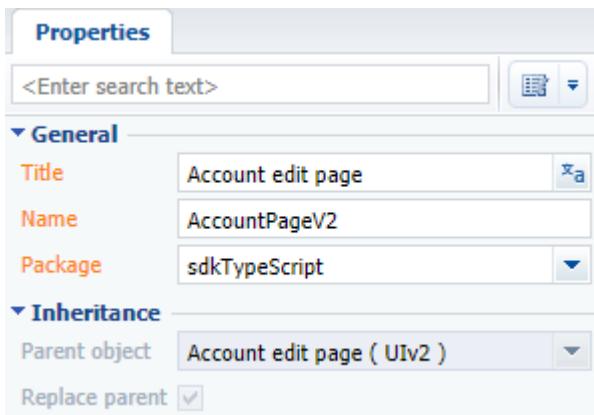
Perform this step to apply changes in the *bootstrap.js* file. You can also use the WorkspaceConsole utility (“**Packages file content**”).

6. Use validator in the Creatio schema

In the [Configuration] section:

1. Load the package to the configuration with the [Update packages from file system] action.
2. **Create replacing schema** of the edit page of the account record (Fig. 3).

Fig. 3. Properties of the replacing schema



3. Export the package to the file system using the [Download packages to file system] action.

4. Modify the ..\sdkTypeScript\Schemas\AccountPageV2\AccountPageV2.js file in the following way:

```
// Declaration of the module and its dependencies.
define("AccountPageV2", ["LettersOnlyValidator"], function(LettersOnlyValidator) {
    return {
        entitySchemaName: "Account",
        methods: {
            // Validation method.
            validateMethod: function() {
                // Determining the correctness of filling the AlternativeName column.
                var res =
LettersOnlyValidator.isAcceptable(this.get("AlternativeName"));
                // Output of the result to the user.
                Terrasoft.showInformation("Is 'Also known as' field valid: " + res);
            },
            // Overriding the method of the parent schema that is called when the
            record is saved.
            save: function() {
                // Calling the validation method.
                this.validateMethod();
                // Calling the basic functions.
                this.callParent(arguments);
            }
        },
        diff: /**SCHEMA_DIFF*/ [] /**SCHEMA_DIFF*/
    };
});
```

When the file with the schema source code is saved and the system web-page is updated, the warning message will be displayed on the account edit page when the page is saved (Fig.4, Fig. 5).

Fig. 4. Incorrectly populated field

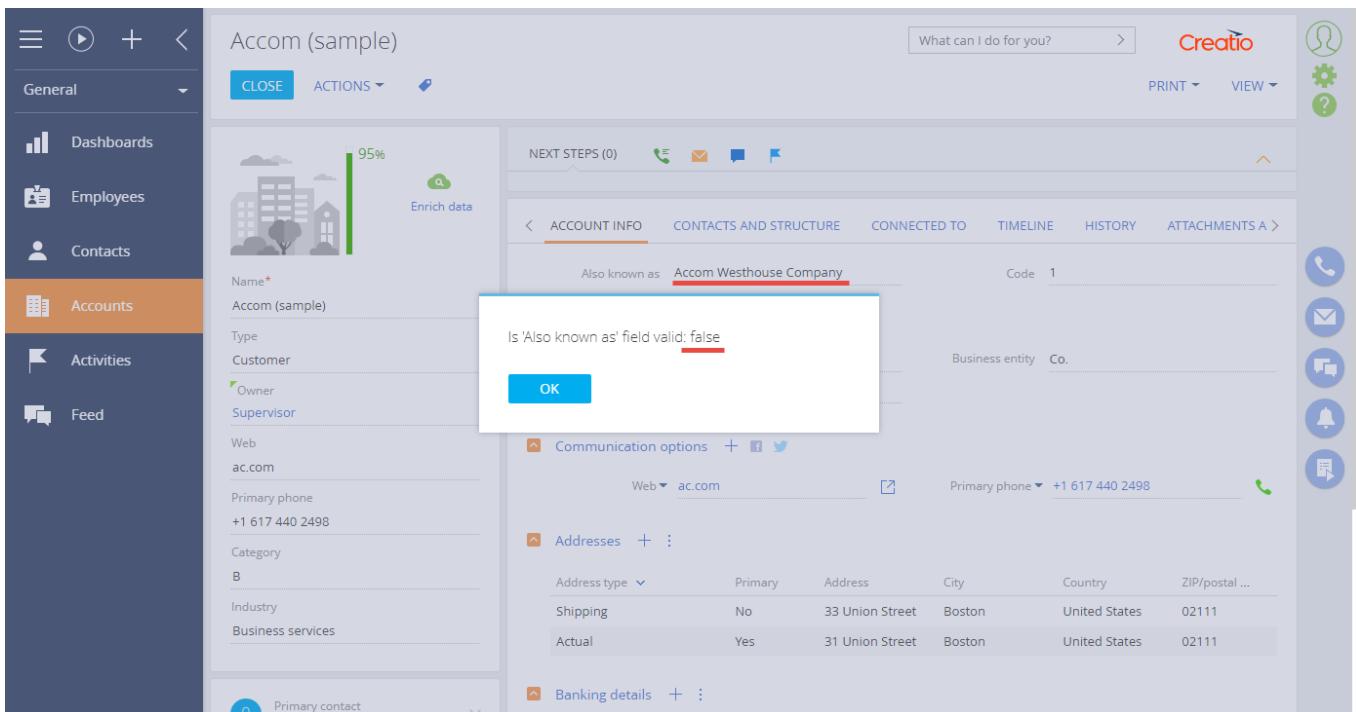
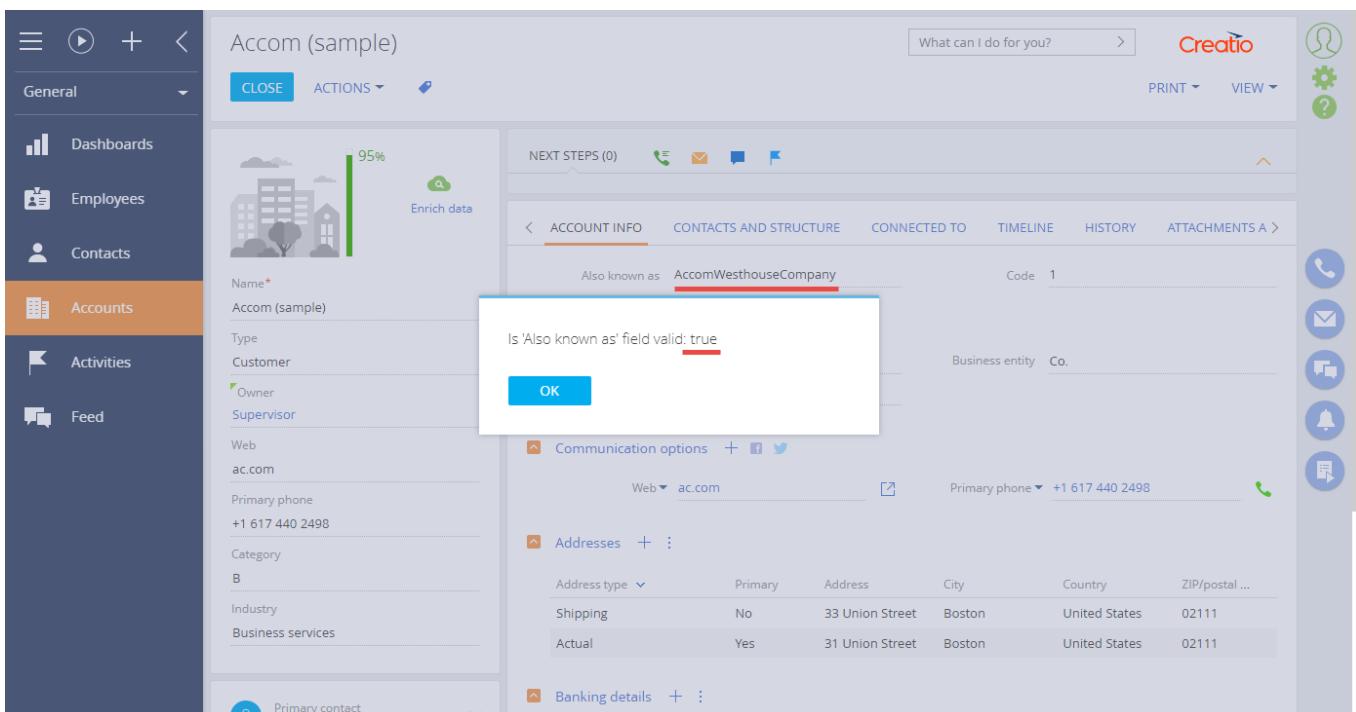


Fig. 5. Correctly populated field



Field validation is described in the “[How to add the field validation](#)” article.

Creating an Angular component to use in Creatio

[Beginner](#)

[Easy](#)

[Medium](#)

[Advanced](#)

Introduction

To build in Angular components into Creatio, use the Angular Elements functionality. Angular Elements – an npm package for packing Angular components into Custom Elements and defining HTML elements with standard behavior (Custom Elements is part of the Web-Components standard).

Creating a custom Angular component

Step 1. Configure the development environment using Angular CLI

To do this, install:

1. [Node.js® and npm package manager](#)
2. Angular CLI.

Before proceeding to the actual integration, verify the Angular version used by Creatio. To do this, take the following steps;

1. Enable the **isDebug mode** by running the following code in the browser console.

```
Terrasoft.SysSettings.postPersonalSysSettingsValue("IsDebug", true)
```

To apply the changes, log out of Creatio and log back in.

2. In the browser console, run the JavaScript code to print the Angular version to install.

```
window.ng.core.VERSION.major
```

3. To install Angular CLI, run the following command in the command prompt:

```
npm install -g @angular/cli
```

For example, to install Angular CLI version 8, run the following command in the command prompt:

```
npm install -g @angular/cli@8
```

Step 2. Create an Angular application

Run the *ng new* command in the command prompt and specify the name of the application, for example, *my-app*:

```
ng new my-app
```

Step 3. Install the Angular Components package

Navigate to the folder of your application, created on the previous step, and run

```
ng add @angular/elements
```

Step 4. Register Angular Component as a Custom Element

To transform the component into a custom HTML element, update the *app.module.ts* file:

1. Add module import of the *createCustomElement*. module.
2. Specify the component name in the *entryComponents* section of the module.
3. In the *ngDoBootstrap* method, register the component under the HTML tag.

```
import { BrowserModule } from "@angular/platform-browser";
import { NgModule, DoBootstrap, Injector, ApplicationRef } from "@angular/core";
import { createCustomElement } from "@angular/elements";
import { AppComponent } from "./app.component";
@NgModule({
    declarations: [AppComponent],
    imports: [BrowserModule],
    entryComponents: [AppComponent]
})
export class AppModule implements DoBootstrap {
    constructor(private injector: Injector) {
    }
    ngDoBootstrap(appRef: ApplicationRef): void {
        const component: any = createCustomElement(AppComponent, { injector:
this.injector });
    }
}
```

```
        customElements.define("my-component", component);
    }
}
```

Specify the corresponding HTML tag in the *app.component.ts* file.

```
@Component({
  selector: "my-component",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"]
})
```

4. Delete the *bootstrap* section from the module or leave the array empty.

Step 5. Build an application

Creatio implements a single Angular core to reduce the amount of memory consumed by custom elements and their connection to the application. Change the building procedure of the newly created Angular application to save dependencies separately from the business logic of the element. Use the *ngx-build-plus* tool for that. The business logic will be saved to the *main-es5.js* file that you can connect to a Creatio package layer. The Angular core dependencies are already included in the Creatio core.

To use the mechanism, connect the *ngx-build-plus* tool by running:

```
ng add ngx-build-plus
```

After that, run *ng g ngx-build-plus:externals*. As a result, you will update the *angular.json* file and create the *webpack.externals.js* file. The file contains the list of dependencies not to include in the build. Additionally, the *build:my-app:externals* command will be generated in the *package.json* file.

Extend the command with the *--outputHashing none* flag to make further working with files more convenient.

The single-core mechanism has the following dependencies:

```
module.exports = {
  "externals": {
    "rxjs": "rxjs",
    "@angular/core": "ng.core",
    "@angular/common": "ng.common",
    "@angular/common/http": "ng.common.http",
    "@angular/platform-browser": "ng.platformBrowser",
    "@angular/platform-browser-dynamic": "ng.platformBrowserDynamic",
    "@angular/compiler": "ng.compiler",
    "@angular/elements": "ng.elements"
  }
}
```

Build the app excluding dependencies by running:

```
build:my-app:externals
```

Connecting Custom Element to Creatio

Add the resulting *main-es5.js* file to a Creatio package as file content. See “**Packages file content**” article for details.

Step 1. Move the file to the static content of the package

To do this, copy the file to the *[Custom package name] | Files | src | js* folder, for example, *MyPackage | Files | src | js*.

Step 2. Set a dependency for the single Angular core

In the *files | src | js | bootstrap.js* file, declare a dependency of the created component on the *ng-core* (common module for using Angular elements).

When using `ng-core`, the production mode is enabled in the single-core mechanism. Remove a call to the `enableProdMode` function in the `[Custom package name]\src\main-es5.js` file. Otherwise, the browser console will display the "Cannot enable prod mode after platform setup" error during runtime.

To do this, set `ng-core` (common module for using Angular elements) as a dependency for `bootstrap`.

```
(function() {
    require.config({
        paths: {
            "MyComponent": Terrasoft.getFileContentUrl("MyPackage", "src/js/main-es5.js"),
        },
        shim: {
            "MyComponent": {
                deps: ["ng-core"]
            }
        }
    });
})();
```

To load `bootstrap`, specify the path to the file, by creating the `descriptor.json` file in `[Custom package name]\Files`:

```
{
    "bootstraps": [
        "src/js/bootstrap.js"
    ]
}
```

Load from the file system and compile.

Step 3. Load the component in the required schema/module

Create a schema or module in the package to use the created custom element. Load the schema or module in the dependency load block of the module.

```
define("CustomModule", ["MyComponent"], function() {
```

Step 4. Create an HTML-module and add it to the DOM.

Here is an example code for a module that adds a custom `my-component` element to the DOM of the Creatio page:

```
/** 
 * @inheritDoc Terrasoft.BaseModule#render
 * @override
 */
render: function(renderTo) {
    this.callParent(arguments);
    const component = document.createElement("my-component");
    component.setAttribute("id", this.id);
    renderTo.appendChild(component);
}
```

Working with data

The Angular component receives data using the public properties/fields marked with the `@Input` decorator

Properties with camelCase names not defined in the decorator explicitly will be transformed into kebab-case HTML attributes.

Example of creating a component property (`app.component.ts`):

```
@Input('value')
public set value(value: string) {
    this._value = value;
```

}

An example of passing data to the component (*CustomModule.js*):

```
/**
 * @inheritDoc Terrasoft.BaseModule#render
 * @override
 */
render: function(renderTo) {
    this.callParent(arguments);
    const component = document.createElement("my-component");
    component.setAttribute("value", 'Hello');
    renderTo.appendChild(component);
}
```

Data retrieval is implemented via the event functionality. Mark the public field (of the *EventEmitter* type) using the *@Output* decorator. To initialize an event, call the *emit(T)* method of the field and pass the required data.

An example of event implementation in the component (*app.component.ts*):

```
/**
 * Emits item click.
 */
@Output() itemClick = new EventEmitter<any>();

/**
 * Handles item click.
 * @param eventData - Event data.
 */
handleItemClick(eventData: any) {
    this.itemClick.emit(eventData);
}
```

An example of event processing in Creatio (*CustomModule.js*):

```
/**
 * @inheritDoc Terrasoft.Component#initDomEvents
 * @override
 */
initDomEvents: function() {
    this.callParent(arguments);
    const el = this.component;
    if (el) {
        el.on("itemClick", this.onItemClickListener, this);
    }
}
```

Description of the solutions for typical errors

Table 1 lists solutions for typical errors that can emerge when creating an Angular component in Creatio.

Table 1. List of the solutions for typical errors

Error	Solution
Cannot enable prod mode after platform setup	When using <i>ng-core</i> , the production mode is enabled in the single-core mechanism. Remove <i>enableProdMode</i> from the <i>my-app src main.ts</i> file.
The selector "my-component" did not match any elements	Remove the <i>bootstrap</i> property from the <i>@NgModule</i> decorator in the module with the description of the <i>ngDoBootstrap</i> method. Alternatively, make the property an empty array. Otherwise, Creatio will load the modules described in the array, not the module that contains <i>ngDoBootstrap</i> .
You provided an invalid object where a stream was expected.	When providing a polyfill web component, make sure that the polyfill does not come with a <i>zone.js</i> file since <i>zone.js</i> is supplied with <i>ng-core</i> .

You can provide an Observable, Promise, Array, or Iterable

Type AppModule does not have 'ngModuleDef' property Check the Angular version in the custom component and the *ng-core* version.
To check the *ng-core* version, run `window.ng.core.Version`.

Development tools. SQL

Contents

- **How to work with PostgreSQL**

How to work with PostgreSQL

Beginner

Easy

Medium

Advanced

General recommendations

1. Avoid using the *CREATE OR REPLACE* command for creating triggers, views, and functions. Instead, use the *DROP ... IF EXISTS* construction first (you may also use the *CASCADE* command if needed), then use *CREATE OR REPLACE*.
2. Use the “*public*” scheme instead of the “*dbo*” scheme.
3. Note that system names are case-sensitive. Remember to wrap the names of tables, columns and other elements in quotes (“*“”*”).
4. Instead of the *MS SQL*’s *BIT* type, use the *Postgres*’s *BOOL* type. The *WHERE "boolColumn" = true* construction is redundant for matching the value of a *BOOL* type field. Using *WHERE "boolColumn"* or *WHERE NOT "boolColumn"* will suffice.
5. *Postgres* allows using a shortened form of explicit conversion, `::TEXT`.
6. String matching is case-sensitive in *Postgres*. To perform case-insensitive matching, use the *iLIKE* keyword. Note that this type of matching is significantly slower than the *UPPER+LIKE* combination. Additionally, the *UPPER+LIKE* combination has less strict index applicability rules compared to *iLIKE*.
7. If there is no implicit type casting, you can create one using the *CREATE CAST* command. More information is available in the [PostgreSQL documentation](#).
8. *Postgres* lacks a built-in *NESTLEVEL* function for recursive procedures. Instead, you should use a special parameter in the procedure for storing the current recursion depth.
9. Use the *NAME* type instead of the *SYSNAME* type.
10. Avoid using empty *INSTEAD* triggers and create rules instead, for instance:

```
CREATE RULE US_VwAdministrativeObjects AS
ON UPDATE TO "VwAdministrativeObjects"
DO INSTEAD NOTHING;
```

11. Implicit type casting from the *INT* type to the *BOOL* type does not work for the *UPDATE* command in *Postgres* even with the corresponding *CAST* operator. You must cast an *INT* value to the *BOOL* type explicitly.

12. The methods for formatting string literals are described in detail in the PostgreSQL documentation:

- [quote_ident](#),
- [quote_literal](#),
- [format](#).

13. Use the following construction instead of `@@ROWCOUNT`:

```
DECLARE rowsCount BIGINT = 0;
GET DIAGNOSTICS rowsCount = row_count;
```

14. Instead of using the following MS SQL construction:

```
(CASE WHEN EXISTS (
```

```

SELECT 1
FROM [SysSSPEntitySchemaAccessList]
WHERE [SysSSPEntitySchemaAccessList].[EntitySchemaUID] = [BaseSchemas].[UID]
)
THEN 1 ELSE 0 END) AS [IsInSSPEntitySchemaAccessList]

```

use the following PostgreSQL construction

```

EXISTS (
  SELECT 1
  FROM "SysSSPEntitySchemaAccessList"
  WHERE "EntitySchemaUID" = BaseSchema."UID"
) "IsInSSPEntitySchemaAccessList"

```

The response field will have a *BOOL* value.

Data type matching.

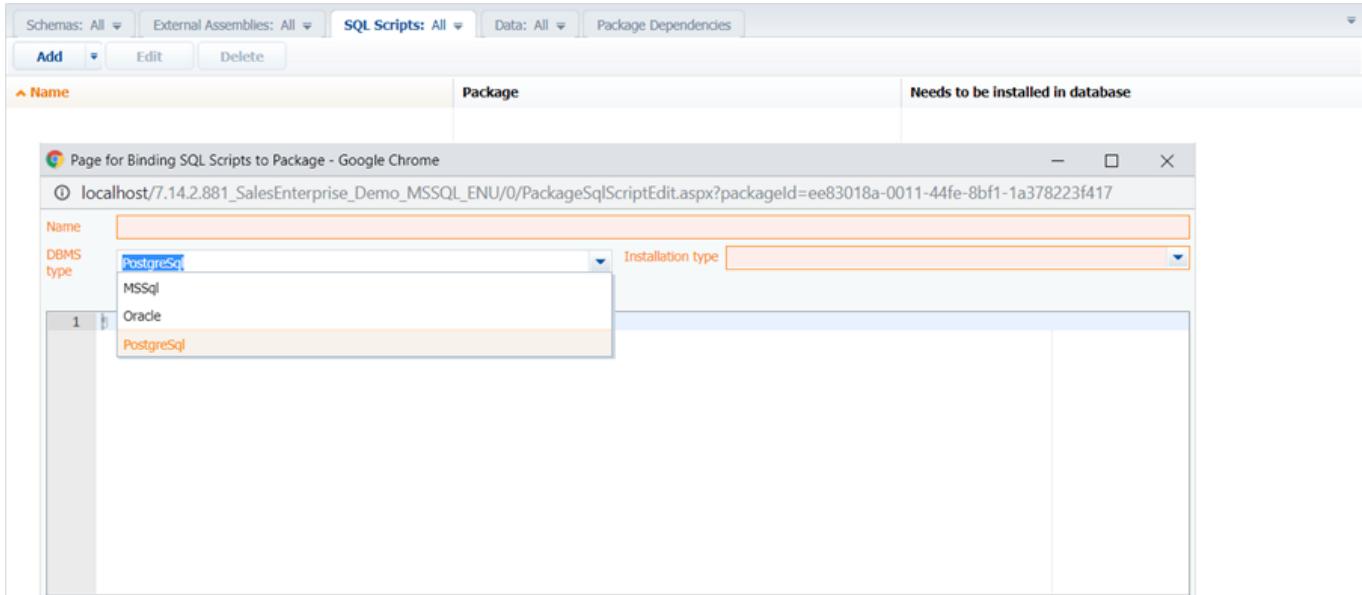
Table 1. – Table for Creatio, MS SQL and PostgreSQL data type matching

Data type in Creatio object designer	Data type in MS SQL	Data type in PostgreSQL
BLOB	VARBINARY	BYTEA
Boolean	BIT	BOOLEAN
Color	NVARCHAR	CHARACTER VARYING
CRC	NVARCHAR	CHARACTER VARYING
Currency	DECIMAL	NUMERIC
Date	DATE	DATE
Date/Time	DATETIME2	TIMESTAMP WITHOUT TIME ZONE
Decimal (0.ooooooooo1)	DECIMAL	NUMERIC
Decimal (0.0001)	DECIMAL	NUMERIC
Decimal (0.001)	DECIMAL	NUMERIC
Decimal (0.01)	DECIMAL	NUMERIC
Decimal (0.1)	DECIMAL	NUMERIC
Encrypted string	NVARCHAR	CHARACTER VARYING
File	VARBINARY	BYTEA
Image	VARBINARY	BYTEA
Image Link	UNIQUEIDENTIFIER	UUID
Integer	INTEGER	INTEGER
Lookup	UNIQUEIDENTIFIER	UUID
Text (250 characters)	NVARCHAR(250)	CHARACTER VARYING
Text (50 characters)	NVARCHAR(50)	CHARACTER VARYING
Text (500 characters)	NVARCHAR(500)	CHARACTER VARYING
Time	TIME	TIME WITHOUT TIME ZONE
Unique identifier	UNIQUEIDENTIFIER	UUID
Unlimited length text	NVARCHAR(MAX)	TEXT

Binding an SQL scenario to a package

If you have SQL scripts (including MS SQL-specific ones) bound to a package, create a new script for PostgreSQL that would implement the same features in the PostgreSQL syntax. To do this, add a new PostgreSQL script on the [SQL scenarios] tab (fig. 1).

Fig. 1. – Binding a PostgreSQL scenario to a package



Compare *MS SQL* and *PostgreSQL* scripting examples

To view the examples, download the files by clicking the links below.

Views

1. [An example SQL script for creating a view and triggers for adding, modifying, and deleting records in the target table.](#)
2. [An example SQL script for illustrating the usage of a rule instead of a trigger in PostgreSQL.](#)

Stored procedures and functions

1. [An example SQL script for creating a stored procedure that uses cycles, cursors, and temporary tables.](#)
2. [An example recursive stored procedure that returns a table and uses PERFORM.](#)
3. [An example stored procedure that uses exception handling and custom script execution.](#)
4. [An example function.](#)

PostgreSQL script development documentation:

1. [Official documentation by the vendor.](#)

Version control systems

Contents

- **Subversion**
- **Git**

Subversion

Contents

- **Introduction**
- **Creating a package in the file system development mode**
- **Installing an SVN package in the file system development mode**
- **Binding an existing package to SVN**
- **Updating and committing changes to the SVN from the file system**
- **Creation of the package and switching to the file system development mode**

Working with SVN in the file system

Beginner

Easy

Medium

Advanced

Introduction

Subversion (SVN) is a centralized system designed for collaborative work. It is based on a repository that contains data in form of a “tree” hierarchy of files and folders. Users can connect to the repository to browse, view or modify files. Their modifications are available to all other users, and vice versa.

All modifications are documented in SVN, including the information about added, deleted, and moved files and folders. Users can access SVN files and folders at any given moment. Additionally, they can view all other versions of all files and folders.

Learn more about configuring and using Subversion in this [article](#).

General outline:

Repository – a central database, usually located on a file server that contains versioned files with their full history. A repository can be accessed through various network protocols or from a local disk.

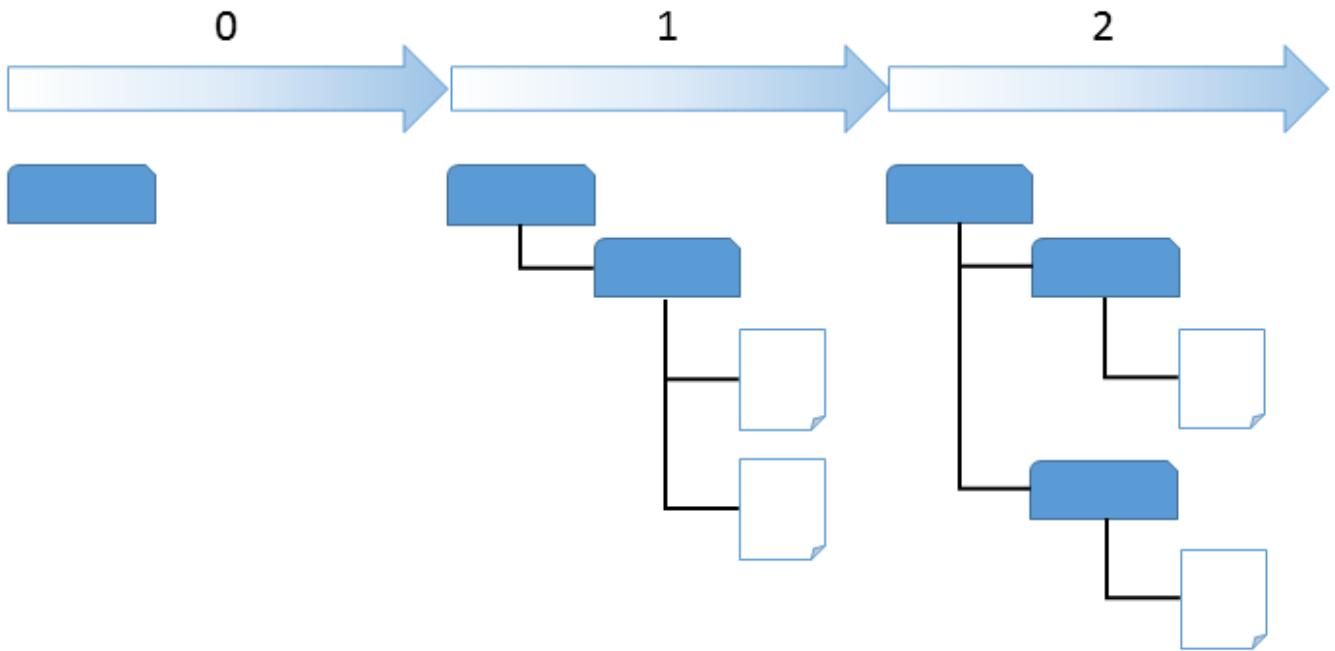
Working copy – a folder located on the developer’s computer. A developer can get the latest version of the files from the repository, work with them locally, and commit these files back to the repository when they are done. A working copy does not include the project’s history, but does contain the copies of files that were located in the repository prior to any changes made by the developer. This enables complete visibility over any changes made to the files.

Detailed changes are only documented for text files. SVN documents only the general information about changes made to binary files.

Revision – a documented state of the file hierarchy. In the repository, each commit is treated as an atomic transaction. Developers can modify several files, create, delete, rename and copy files and folders, and commit the entire set of changes as a single “revision”.

In SVN, all revisions are stored in form of file system “trees” - an array of revision numbers starting with 0 and “growing” from left to right (Fig. 1). Each number corresponds to a file system tree. And each tree is a “snapshot” of the storage state after each commit.

Fig. 1. Revisions in the repository



Unlike other version control systems, revision numbers in Subversion refer to whole trees instead of individual files.

Versioning models

File-sharing problems

While working in SVN, there may be a problem where two (or more) developers are using the same file to implement different functionality. In this case, if one of the developers commits their changes a few seconds before the other, their changes may be overwritten. And while the system documents all changes, their work may still be lost in the latest version of the file, if the other developer accidentally overwrites the same file.

The following versioning models are used to avoid such problems:

- The “Lock-Modify-Unlock” solution
- The “Copy-Modify-Merge” solution

The “Lock-Modify-Unlock” solution

This versioning model only enables a single user to modify a specified file. A user may “block” the file for all other users, and they will not be able to commit their changes until the lock is released.

Disadvantages:

- Locking may cause administrative problems, e.g. when the first developer locks a file and forgets to release the lock.
- Locking may cause unnecessary serialization. If the developers are working on two separate parts of the same document which do not overlap (e.g., the beginning and the end), the changes can be properly merged together if the file is not locked.
- Locking may create a false sense of security. Two separate files that depend on each other may be locked by two different developers, which means the changes made to each locked file are semantically incompatible, but the two developers may think they are beginning a safe, insulated task. Thus, this model inhibits the two developers from discussing their incompatible changes early on.

This model is more appropriate when the two separate files can not be merged. For example, if two users are editing an image at the same time, they will not be able to merge their changes.

The “Copy-Modify-Merge” solution

In this model, each user's client reads the repository and creates a personal working copy of the file or project. Users then work in parallel, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but the user is responsible for making it happen.

correctly.

A *conflict* will happen if the changes overlap when two users work on the same file simultaneously. In this case, the user who commits the changes should select the necessary revision from the list of files in a conflict state. Upon resolving the conflict, the merged file can be committed to the repository.

The chance of semantic and syntactic conflicts increases if there is no communication between users.

Determining the state of the working copy file

Subversion records information about the following properties in the .svn service folder of a working copy for each file:

- the revision, which the file in the working copy is based on (working revision of the file)
- date and time of the latest file update from the repository

Based on this information, Subversion can determine the state of the working copy file:

1. Not modified and not outdated. Not modified in the working copy. The repository did not document any changes to this file since its working revision. The update or the commit procedures will not be executed.
2. Modified locally and not outdated. Modified in the working copy. The repository did not document any changes to this file since its base revision. The update will not be executed. The system will commit the changes successfully.
3. Not modified, outdated. Not modified in the working folder, modified in the repository. The file must be updated to match the current public revision. The update will not be executed. The system will commit the changes successfully.
4. Modified locally and outdated. Modified in the working folder and in the repository. The commit procedure will fail. The file must first be updated by attempting to merge the changes published by another developer with the local changes. The user must resolve the conflict if Subversion can not merge the files.

Working copy used in Creatio

If the **file system development mode** is enabled, Creatio will use a custom working copy of each user package with connected versioning. These working copies are located in the folder specified in the `defPackagesWorkingCopyPath` element of the `ConnectionString.config` configuration file (see “[Deploying the Creatio on-site application](#)”).

If the file system development mode is enabled, the working copy can be created manually in the `[Installed application path]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\[Package name]` folder (see: “[Creating a package in the file system development mode](#)”).

Using an application to work in SVN

We recommend using [TortoiseSVN](#) (version 1.9 and higher) to work with Subversion (SVN) in the file system. Once installed, the application will be built in in the Windows UI. Learn more about TortoiseSVN in this [article](#).

See also

- [Creating a package in the file system development mode](#)
- [Installing an SVN package in the file system development mode](#)
- [Binding an existing package to SVN](#)
- [Updating and committing changes to the SVN from the file system](#)
- [Creation of the package and switching to the file system development mode](#)

Creating a package in the file system development mode

Beginner

Easy

Medium

Advanced

Introduction

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

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

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

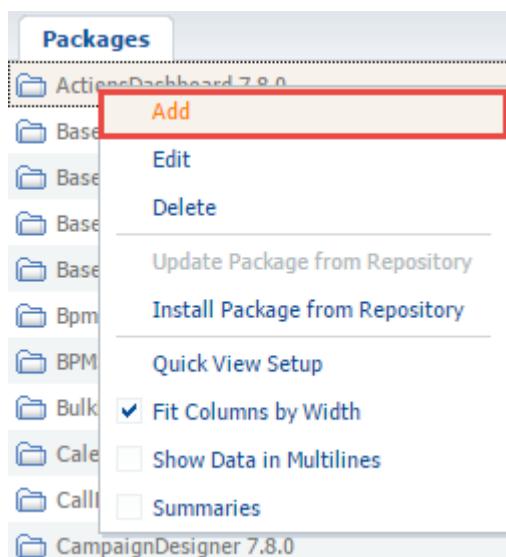
Ensure that **application is configured to access the SVN repository** before binding the package to the SVN repository when development mode in the file system is enabled.

Package creation process

1. Create a package in the application

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

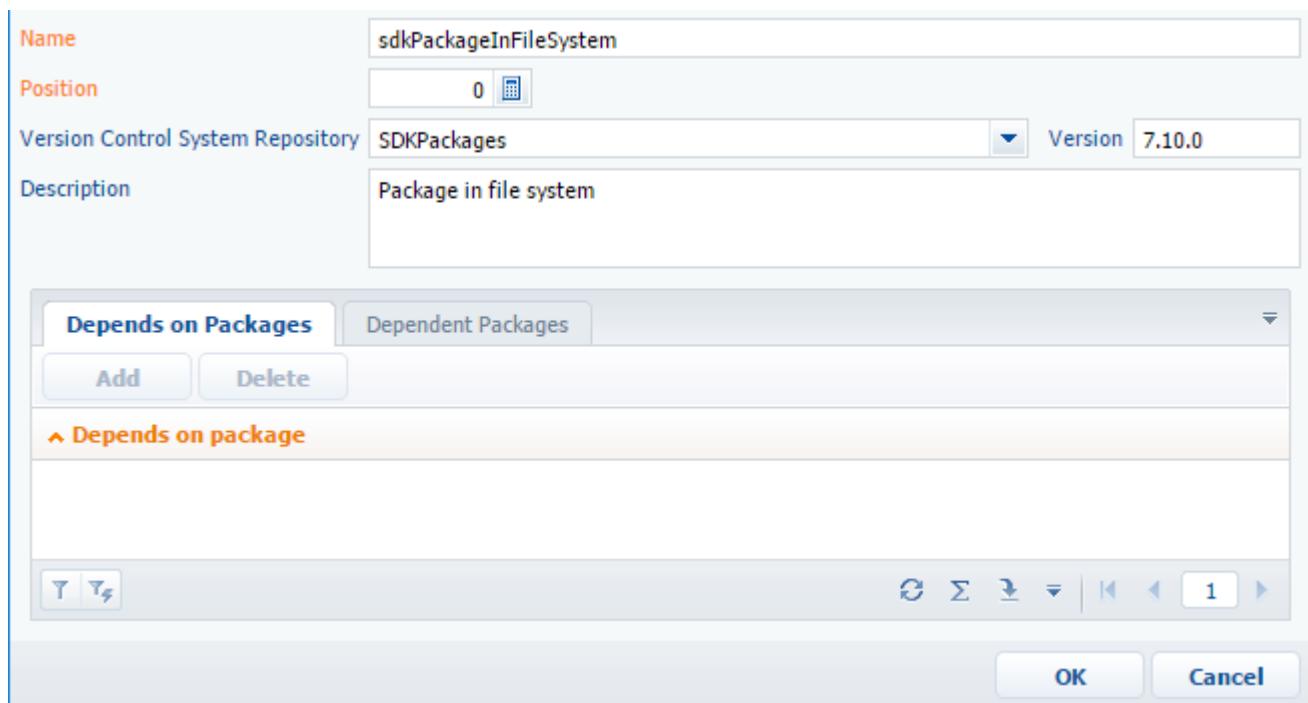
Fig. 1 Adding a package in the [Configuration] section



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

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

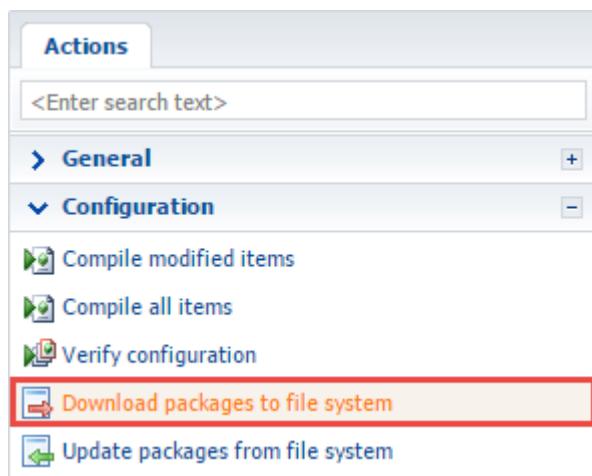
Fig. 2 Package summary



2. Download created package to file system

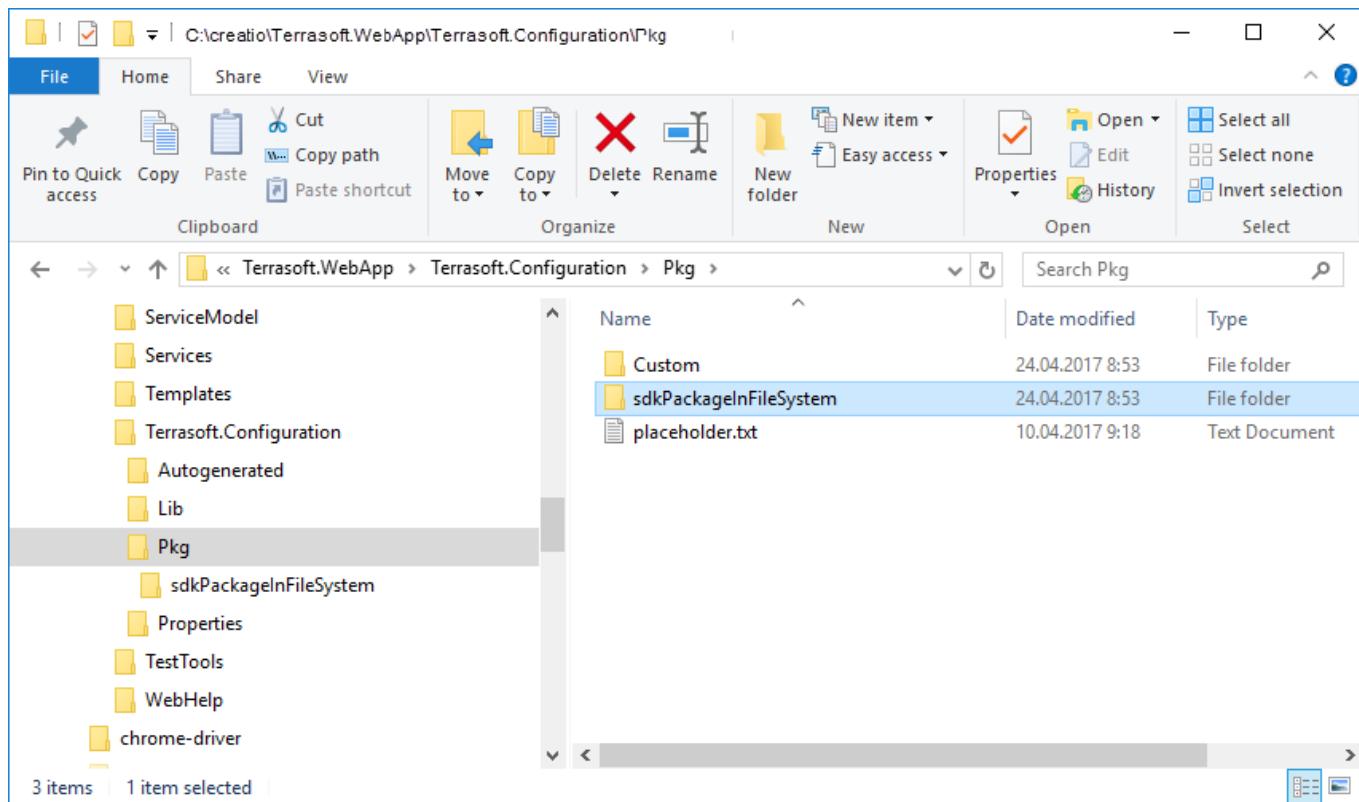
Click [Download packages to file system] (Fig. 3).

Fig. 3 [Download packages to file system] action



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

Fig. 4 Package in the file system



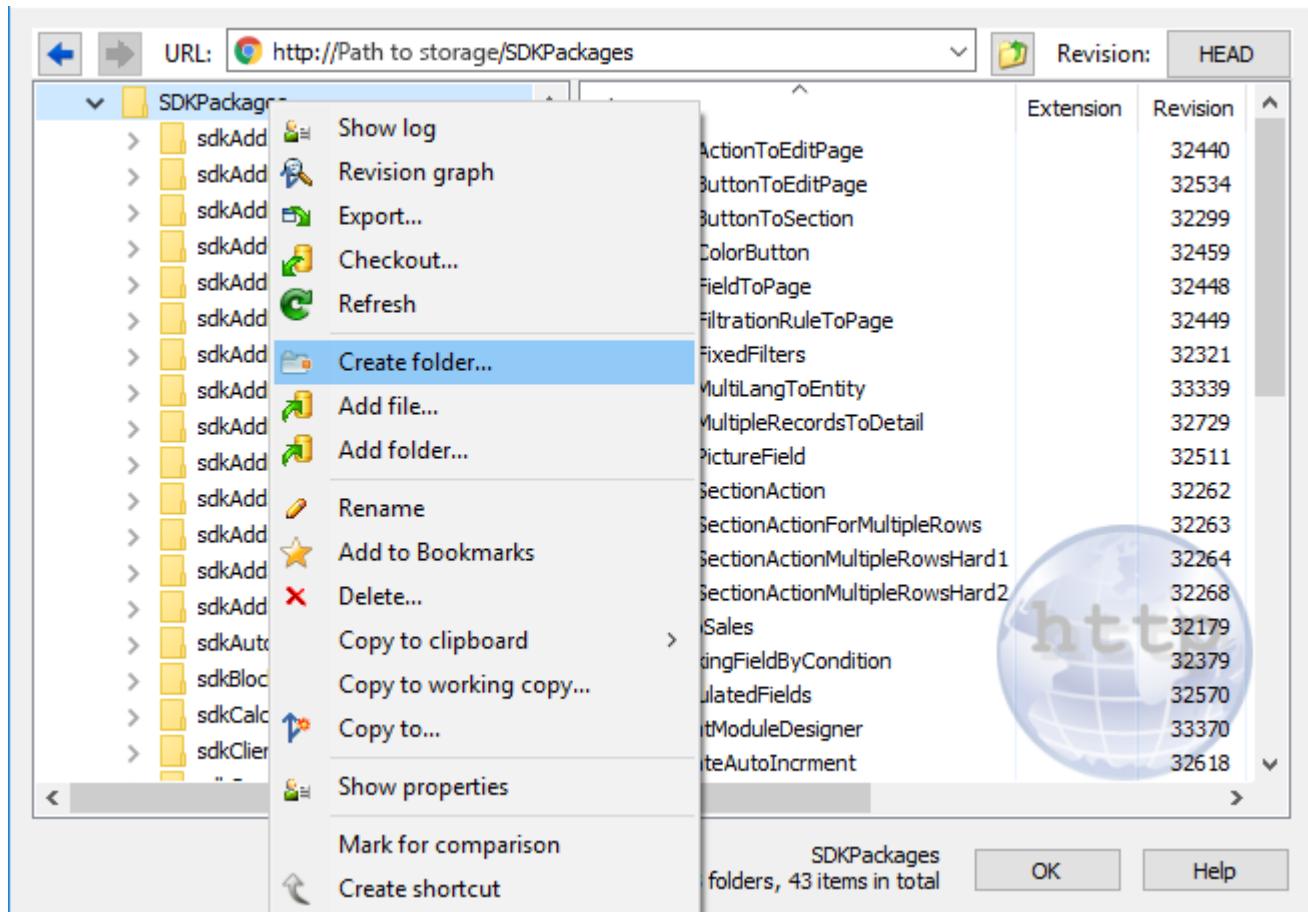
When the file system development mode is enabled, the package must be manually added to the repository.

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

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

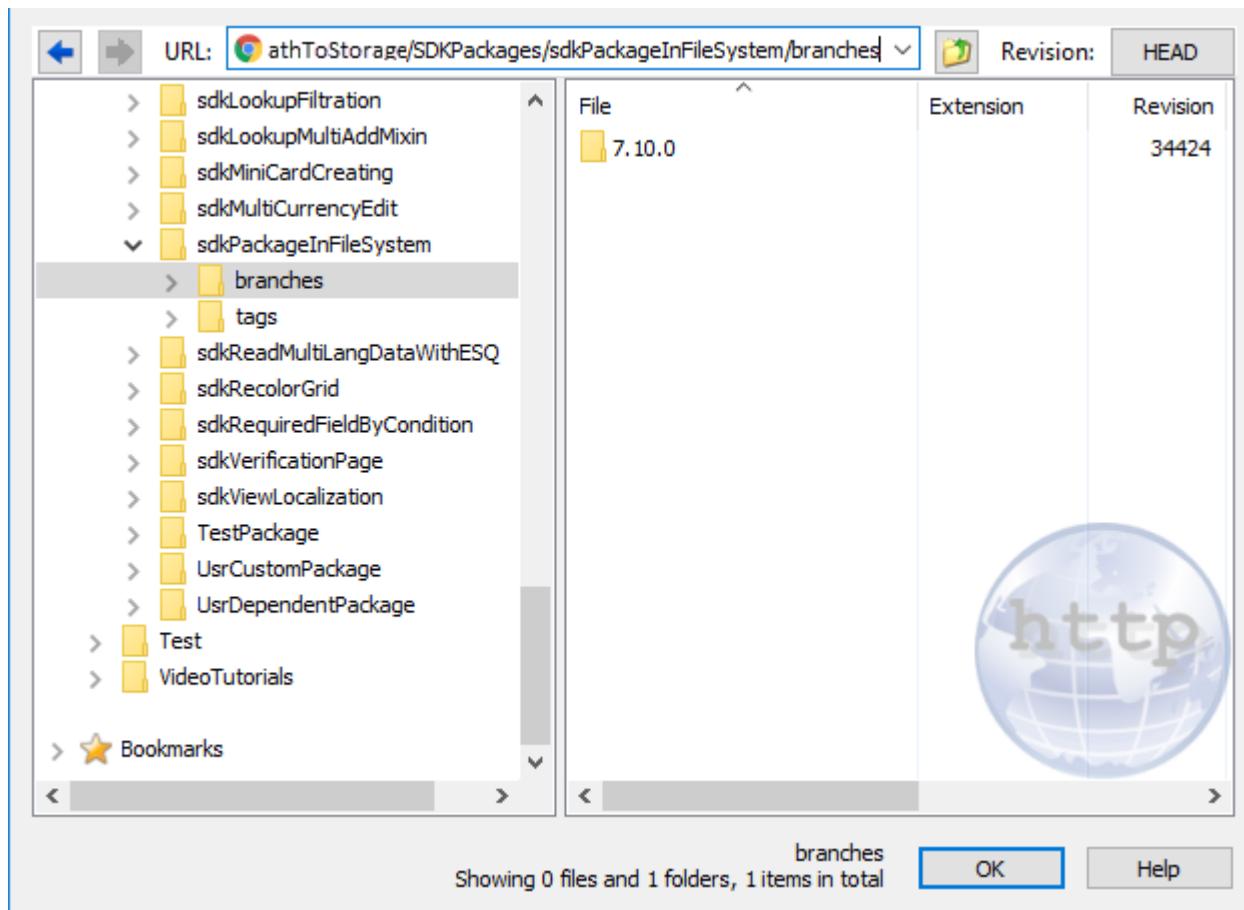
This is a brief example of working with SVN via TortoiseSvn. More information about working with SVN repository via TortoiseSvn can be found in the [documentation](#).

Fig. 5 Creating a folder in the SVN repository



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

Fig. 6 Flat package structure in the repository



4. Create a working copy of the package version branch

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

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

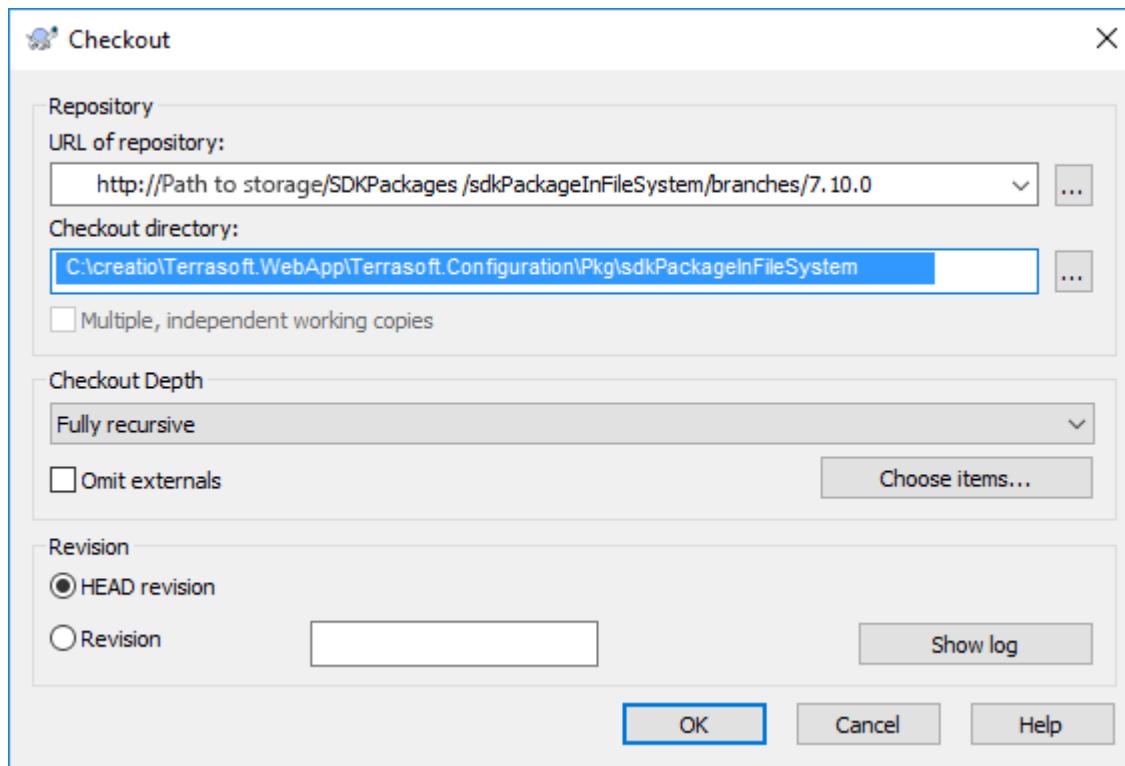
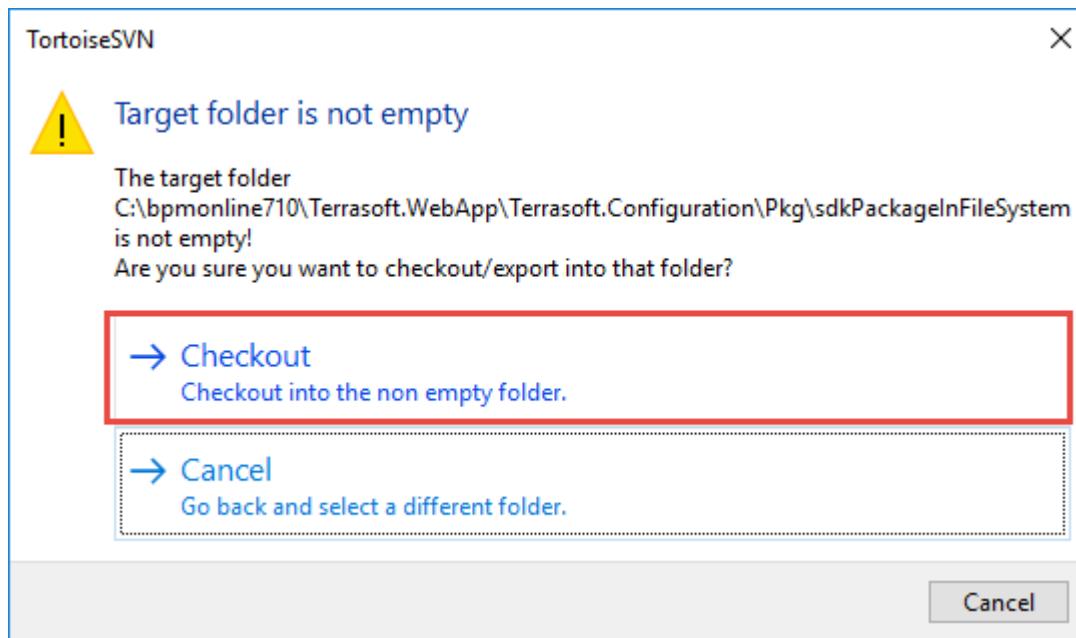


Fig. 8 Confirmation of the SVN checkout operation to the existing folder.



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

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

Name	Date modified	Type
Custom	24.04.2017 8:53	File folder
sdkPackageInFileSystem	24.04.2017 10:31	File folder
placeholder.txt	10.04.2017 9:18	Text Document

5. Commit the package folder in the repository

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

Fig. 10 Adding a folder to the repository

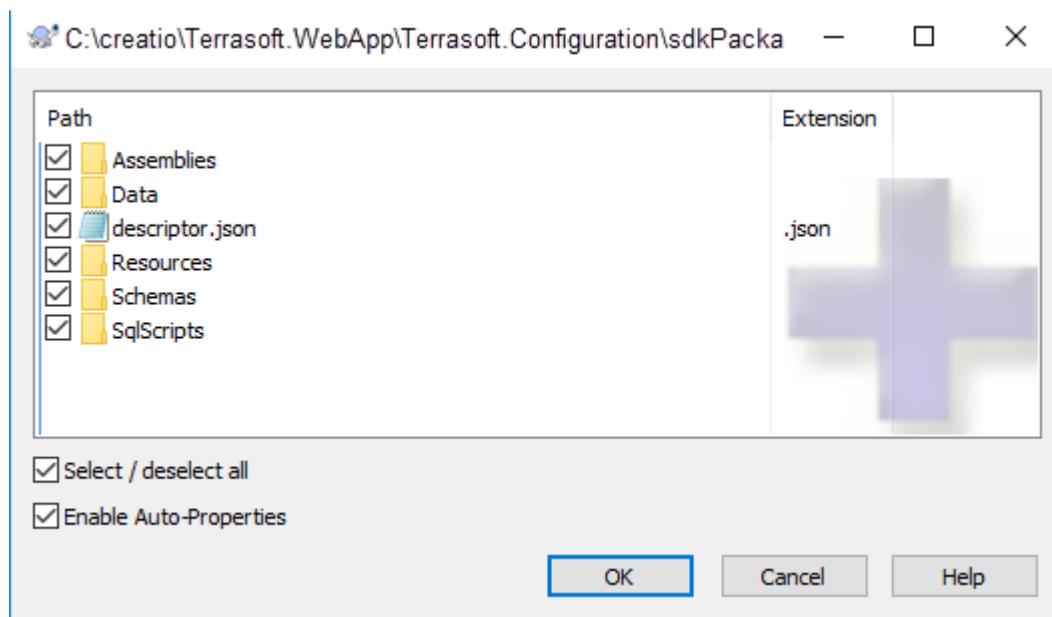
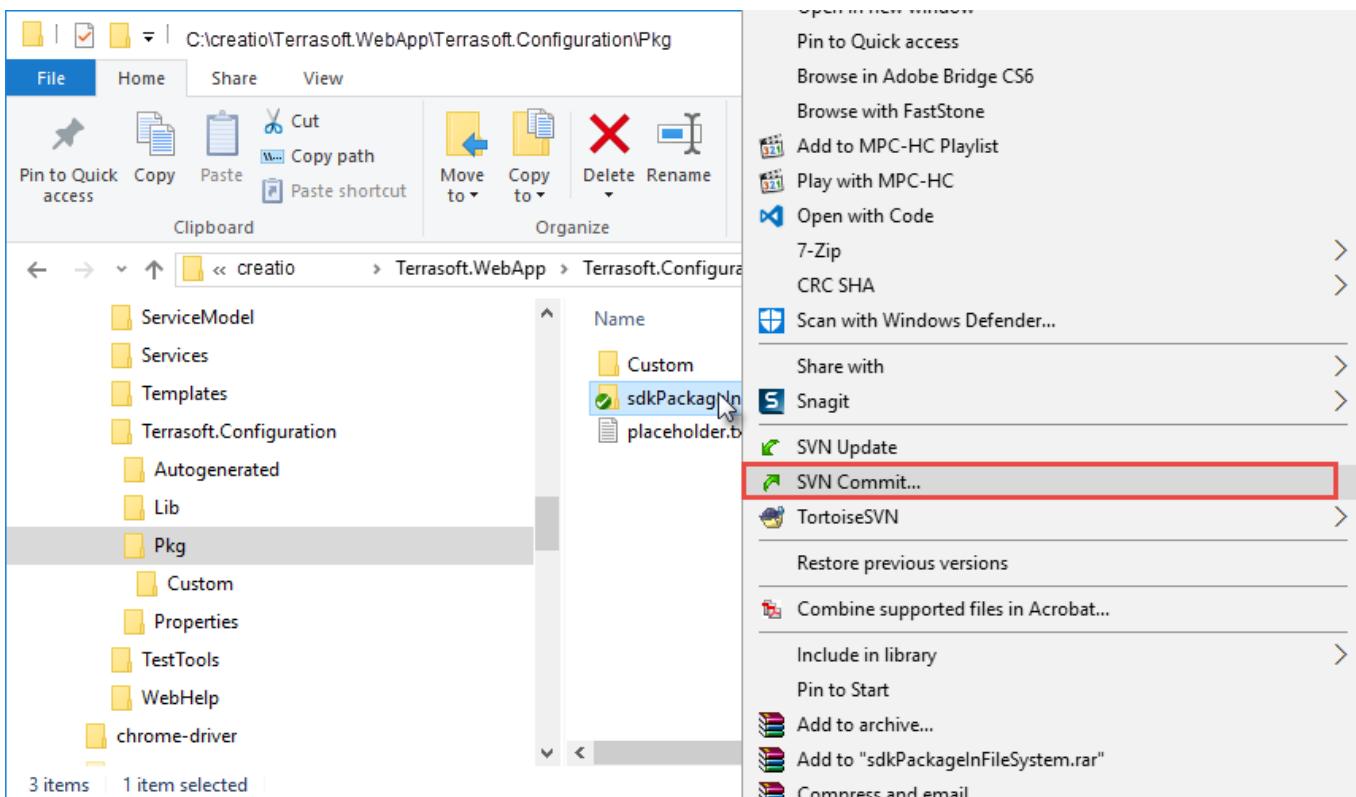


Fig. 11 Committing changes to the repository



See also

- [Working with SVN in the file system](#)
- [Installing an SVN package in the file system development mode](#)
- [Binding an existing package to SVN](#)
- [Updating and committing changes to the SVN from the file system](#)
- [Creation of the package and switching to the file system development mode](#)

Installing an SVN package in the file system development mode

Beginner Easy **Medium** Advanced

Introduction

Package installation from the SVN repository in the **file system development mode** is considerably different from package setup in the [Configuration] section (see “[Installing packages from repository](#)”). The main difference is that the interaction with the SVN repository (installing, committing, updating packages) is performed only from the file system.

The procedure for installing an SVN package in the file system development mode is as follows:

1. Install the package in the file system.
2. Install the package in the application.
3. Generate source codes.
4. Compile the changes.
5. Update the database structure.
6. Install SQL scripts and bound data, if necessary.

To apply all necessary changes automatically, after the package is installed, enable the automatic mechanisms that apply changes. To do this, edit the ..\Terrasoft.WebApp\Web.config file and set the following *appSettings* element keys to *true*:

```

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

```

The *AutoUpdateOnCommit* key is responsible for automatic updating of packages from the SVN before committing. If this key is set to *false*, then the application will notify the user that an update is required if the package schemas have been modified. The *AutoUpdateDBStructure*, *AutoInstallSqlScript*, *AutoInstallPackageData* keys are responsible for automatic database structure update, SQL script installation and installation of bound data respectively.

If the mechanism for auto-applying the changes is enabled, then steps 3–6 can be skipped. The mode must be enabled before the package is installed.

If you need to interact with the SVN repository both from the [Configuration] section and the file system, then:

1. Install the package from the repository in the [Configuration] section (see “[Installing packages from repository](#)”).
2. Export the package to the file system using the [Download packages to file system] action.

Repeat steps 3–6 from the package installation sequence.

Case description

In the file system development mode, install the package from the SVN repository under the following URL:

<http://svn-server:8050/SDKPackages/sdkCreateDetailWithEditableGrid/branches/7.8.0>

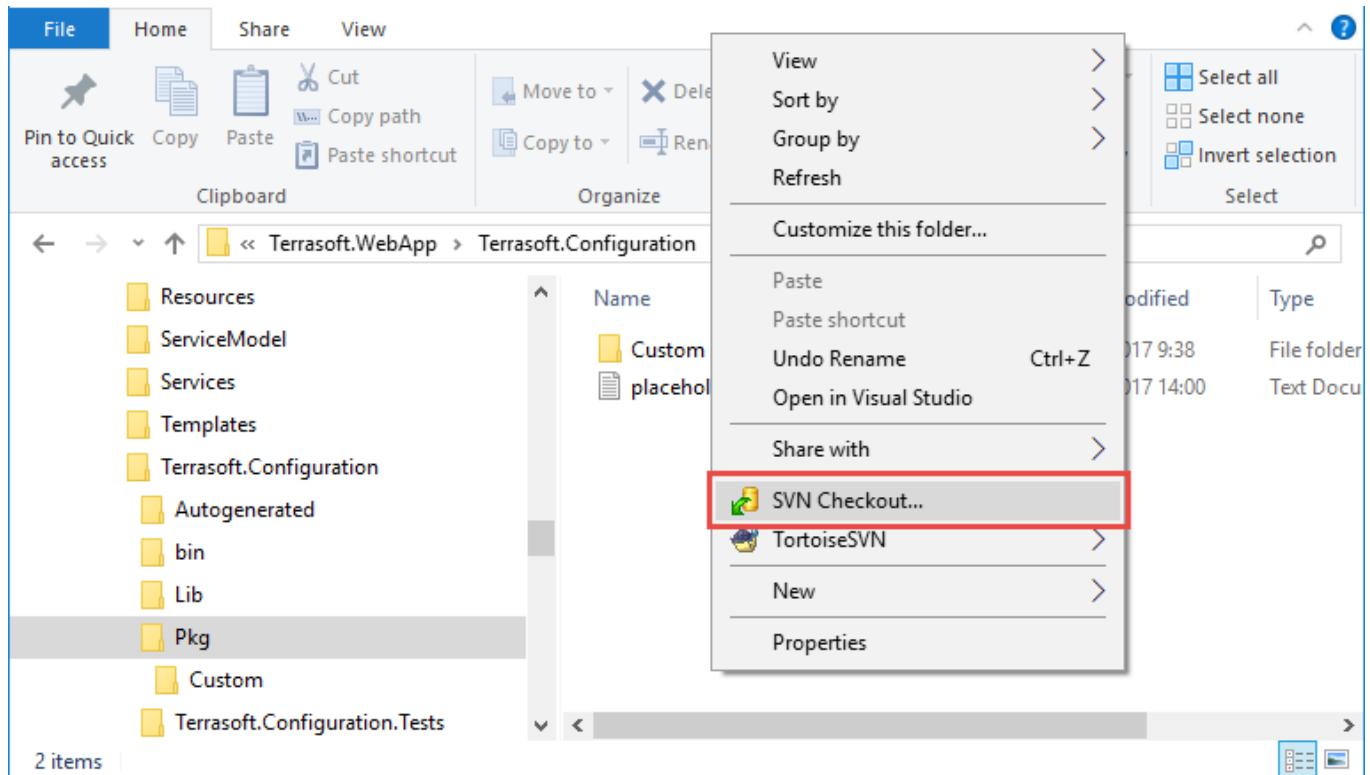
The package in this particular example contains a detail with editable list, which was created according to this article: “[Adding a detail with an editable list](#)”.

Case implementation algorithm

1. Installing the package in the file system

Open the application catalog ...\\Terrasoft.WebApp\\Terrasoft.Configuration\\Pkg in Windows Explorer and execute [SCN Checkout] (Fig. 1).

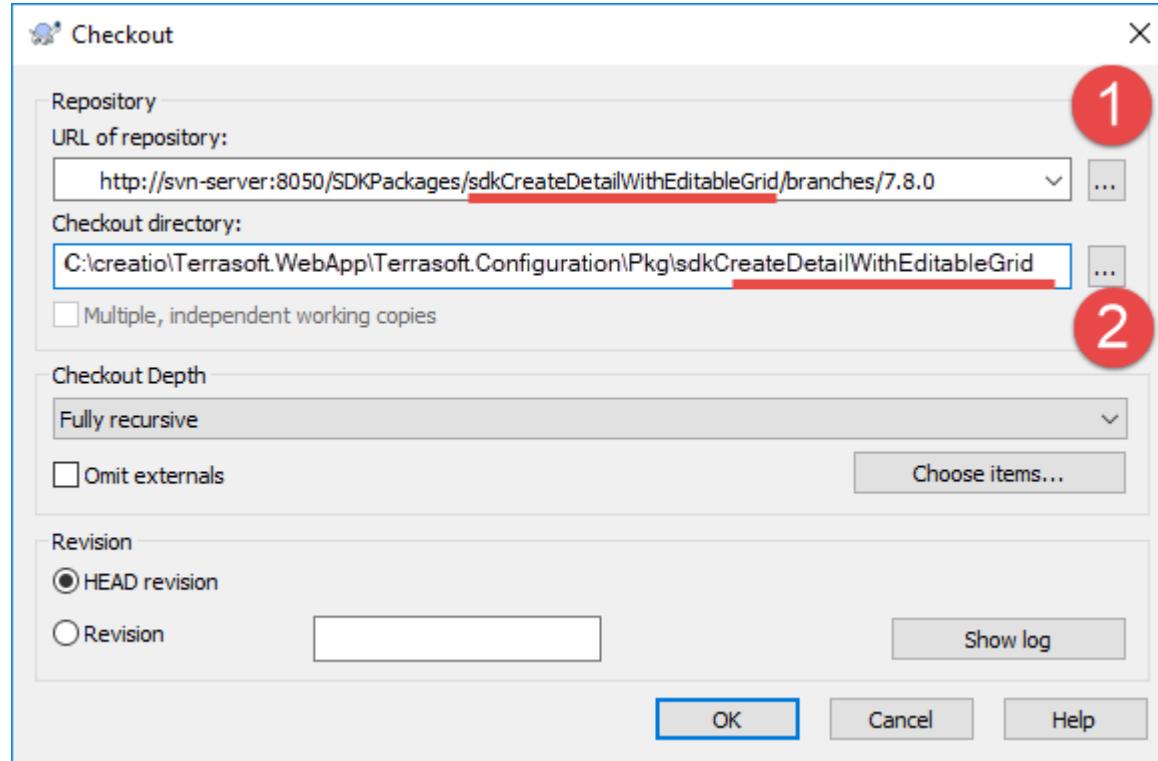
Fig. 1. Running the [SVN Checkout] action



We recommend using the [TortoiseSVN](#) client (v.1.8 and up) for working with Subversion. This client is a Windows shell extension and is built into the Explorer context menu. For more information see [TortoiseSVN documentation](#).

In the checkout window (Fig. 2), specify the package repository address (1) and the export catalog for the package contents (2).

Fig. 2. TortoiseSVN Checkout window



The checkout catalog name must be the same as the package name (Fig. 2).

After the checkout is complete (Fig. 3), the ..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg catalog will contain a local working copy of the package (Fig. 4).

Fig. 3. Checkout operation log

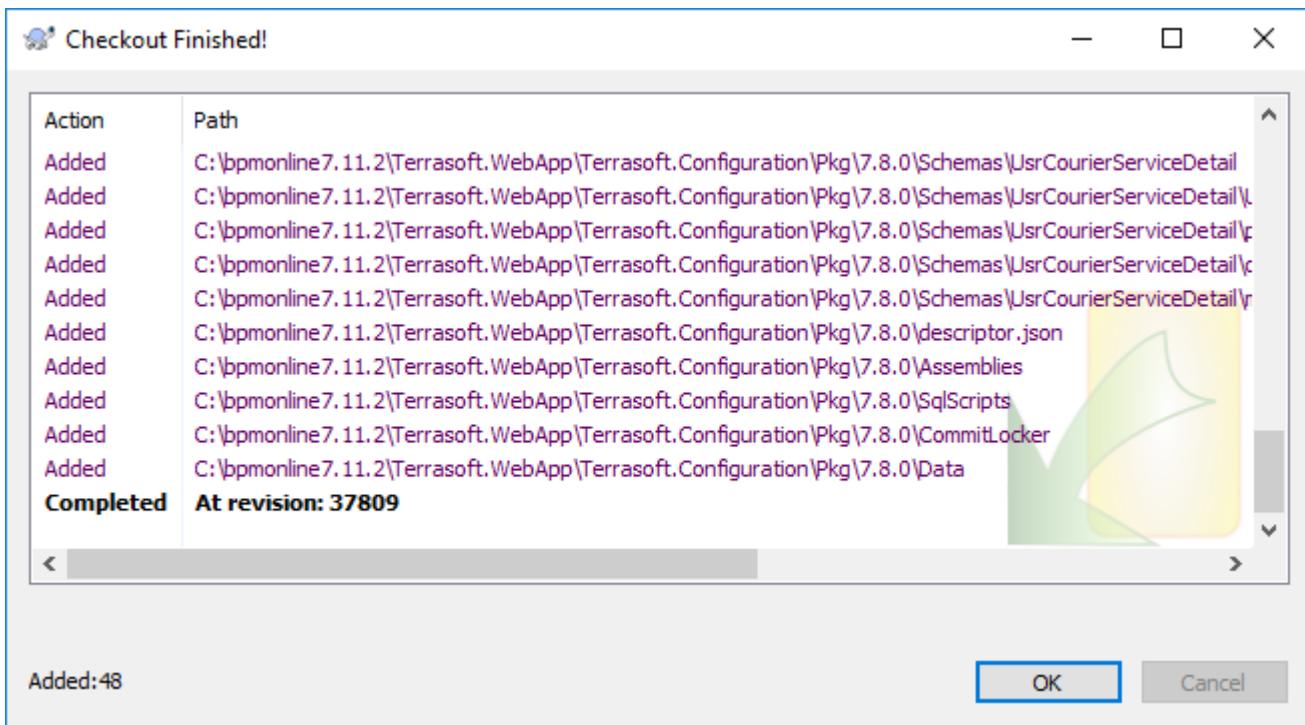
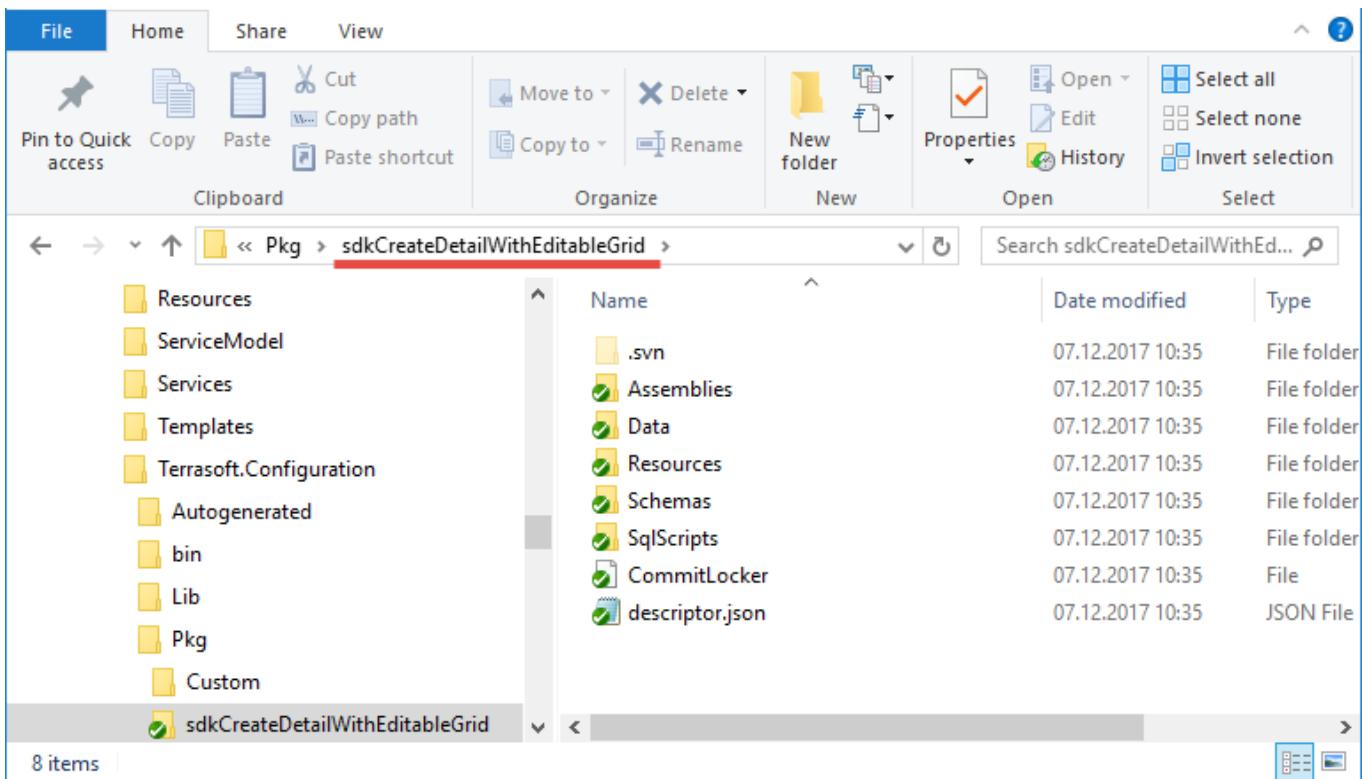


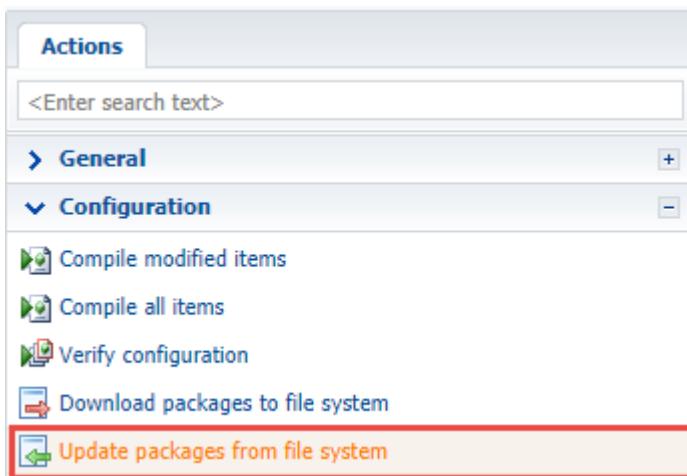
Fig. 4. Package working copy



2. Installing the package in the application

To install a package from the file system, go to the [Configuration] section and execute the [Update packages from file system] action (Fig. 5).

Fig. 5. The [Update packages from file system] action

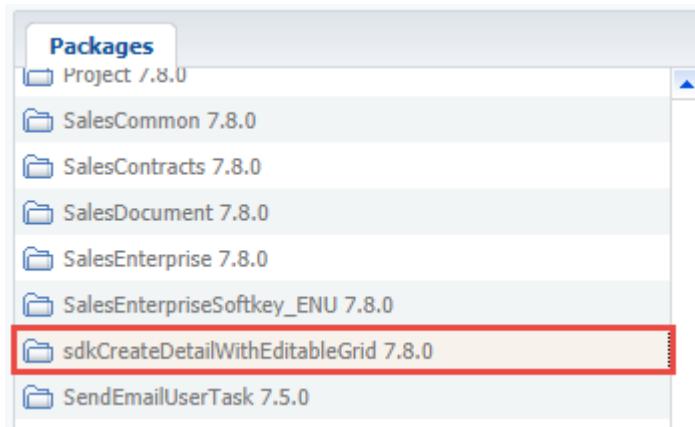


As a result, the package will be added to the application (Fig. 6, Fig. 7).

Fig. 6. Message with the package importing status

Name	Type	Status
sdkCreateDetailWithEditableGrid	Package	Added
OrderPageV2	Schema	Added
UsrCourierService	Schema	Added
UsrCourierServiceDetail	Schema	Added

Fig. 7. The package on the [Packages] tab



If the repository name is missing from the package name, then all changes can be committed to the repository from the file system only.

3. Generating source codes

Execute the [Generate where it is needed] action in the [Configuration] section to generate source core. For more on source code actions, see “**Built-in IDE. The [Configuration] section**”.

4. Compiling the changes

To compile the changes, execute the [Compile modified items] action. For more on configuration actions, see “**Built-in IDE. The [Configuration] section**”.

The requirement for updating the database structure and installing SQL scripts is indicated by the [Database update required] and [Needs to be installed in the database] columns. Please refer to the [Last error message text] column

in case of errors during the database structure update and SQL script installation.

Not all these columns display in the list of the [Schemas], [SQL scripts] and [Data] tabs of **the [Configuration] section** by default. Right-click the list and select the [Set up columns] command to add these columns to view.

5. Updating the database structure

After the compilation is complete, run the [Update where it is needed] action. For more on database structure actions, see “**Built-in IDE. The [Configuration] section**”.

6. Installing SQL scripts and bound data

If the package contains bound SQL scripts and data, you will need to run additional actions. For more on the SQL script and data actions, see “**Built-in IDE. The [Configuration] section**”.

After all the setup actions have been completed, the package functions will become available. In this case, it is a custom detail with editable list (Fig. 8).

Fig. 8. Custom detail with editable list

The screenshot shows a Creatio application window titled "ORD-1 (sample)". The top navigation bar includes "CLOSE", "ACTIONS", a search bar ("What can I do for you?"), the "Creatio" logo, and a "VIEW" dropdown. To the right is a vertical toolbar with icons for user profile, settings, help, phone, email, messaging, notifications, and file operations. The main content area displays an order detail page. At the top, it shows "Status: 1. Draft" and "Payment amount, \$: 0.00". Below this, under the "DELIVERY" tab, it shows "Delivery type: Courier" and "Payment type: Non-cash payment". There are expandable sections for "Delivery address" and "Recipient information". A red box highlights a section for "Courier Service" which contains fields for "Order" (set to "ORD-1 (sample)") and "Account" (set to "Accom (sample)"). An input field "Enter a value" is present with a search icon and a "Enter" button. The bottom of the window has a footer with standard browser controls (refresh, back, forward, etc.).

Users may need to update the page and clear browser cache to access the new functions.

See also

- **Working with SVN in the file system**
- **Creating a package in the file system development mode**
- **Binding an existing package to SVN**
- **Updating and committing changes to the SVN from the file system**

- **Creation of the package and switching to the file system development mode**

Binding an existing package to SVN

Beginner

Easy

Medium

Advanced

You can bind an existing package to SVN only in an on-site application. Working with such a package is possible only in the file system development mode.

After binding a package to SVN in the file system, it can be installed in a different application using Creatio built-in tools (“[Installing packages from repository](#)”).

Introduction

Simple custom functions can be developed by a single developer. The package in which the development is performed often is not connected to the version control repository (SVN).

As the complexity grows, more developers are required to work with the package, which makes it necessary to bind the package to SVN.

The general sequence for binding a package to the repository is as follows:

1. Switch to the file system development mode
2. Export the package to the file system.
3. Create the catalogs for the package in the SVN repository.
4. Create a working copy of the package branch.
5. Commit the package catalog in the repository.

As an alternative, you can use direct SQL queries to the database for binding packages to the repository. To do this:

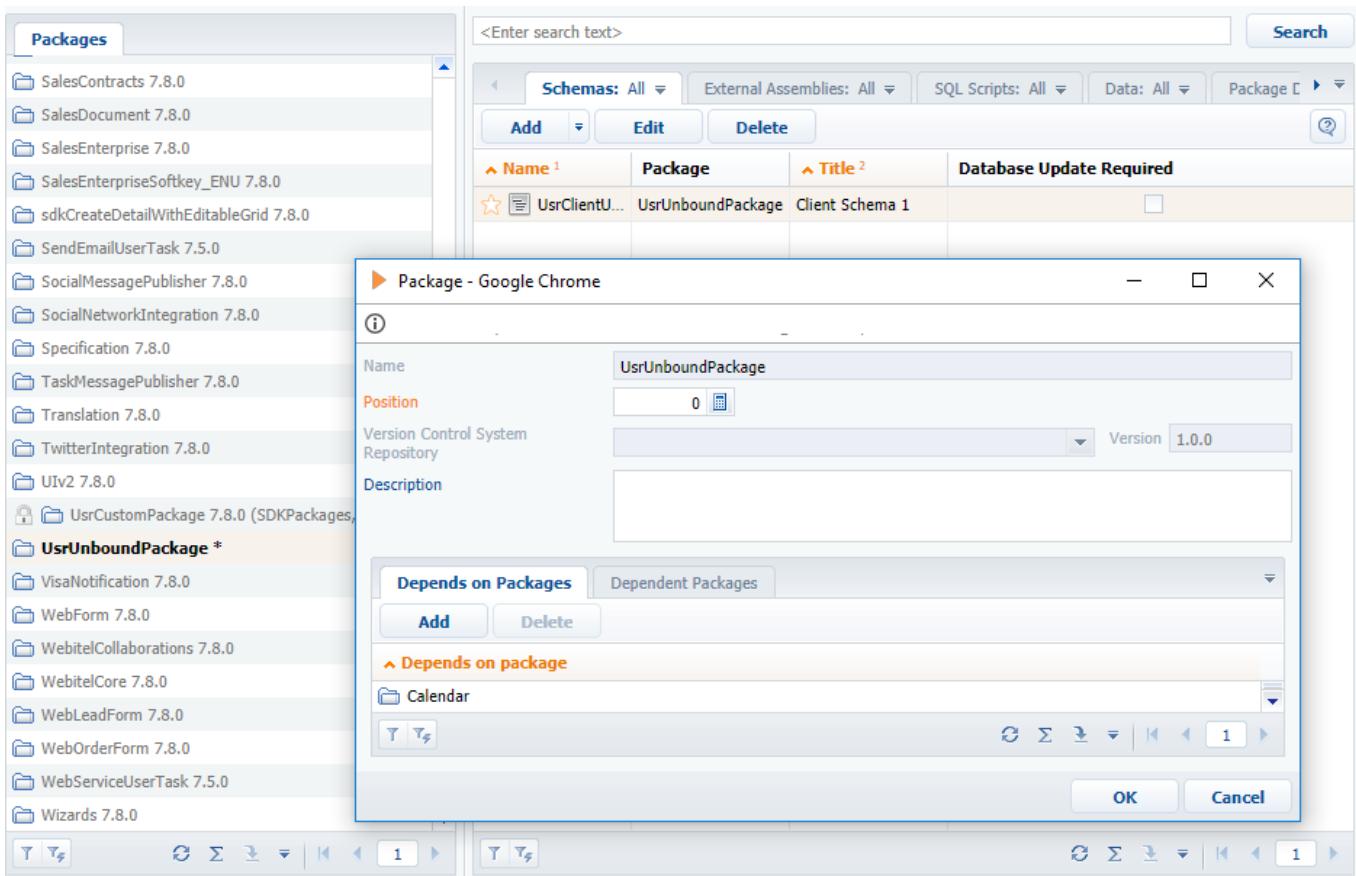
1. Add information about the SVN repository in the **[Configuration]** section.
2. Bind the repository to the package. To do this:

- In the SysRepository table, read the record ID, which contains the SVN repository address.
- In the SysPackage table, add the obtained Id to the SysRepositoryId for the unbound package.

Case description

Bind the custom *UsrUnboundPackage* package (Fig. 1) to the repository.

Fig. 1. Custom package properties



You can obtain SVN access via the following URL:

<http://svn-server:8050/SDKPackages>

Case implementation algorithm

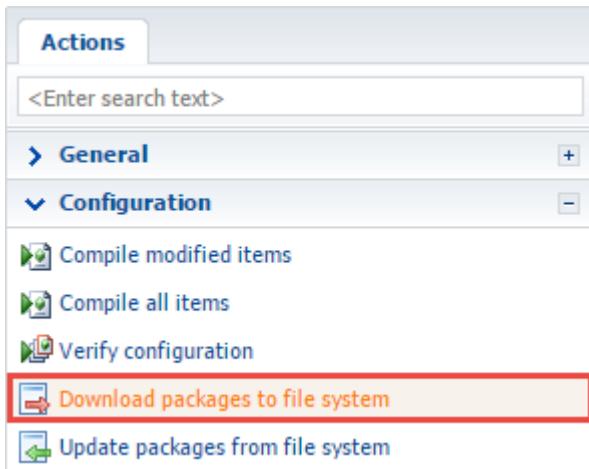
1. Switch to the file system development mode.

For more information about the file system development mode, see the “**Development in the file system**” article.

2. Export the package to the file system.

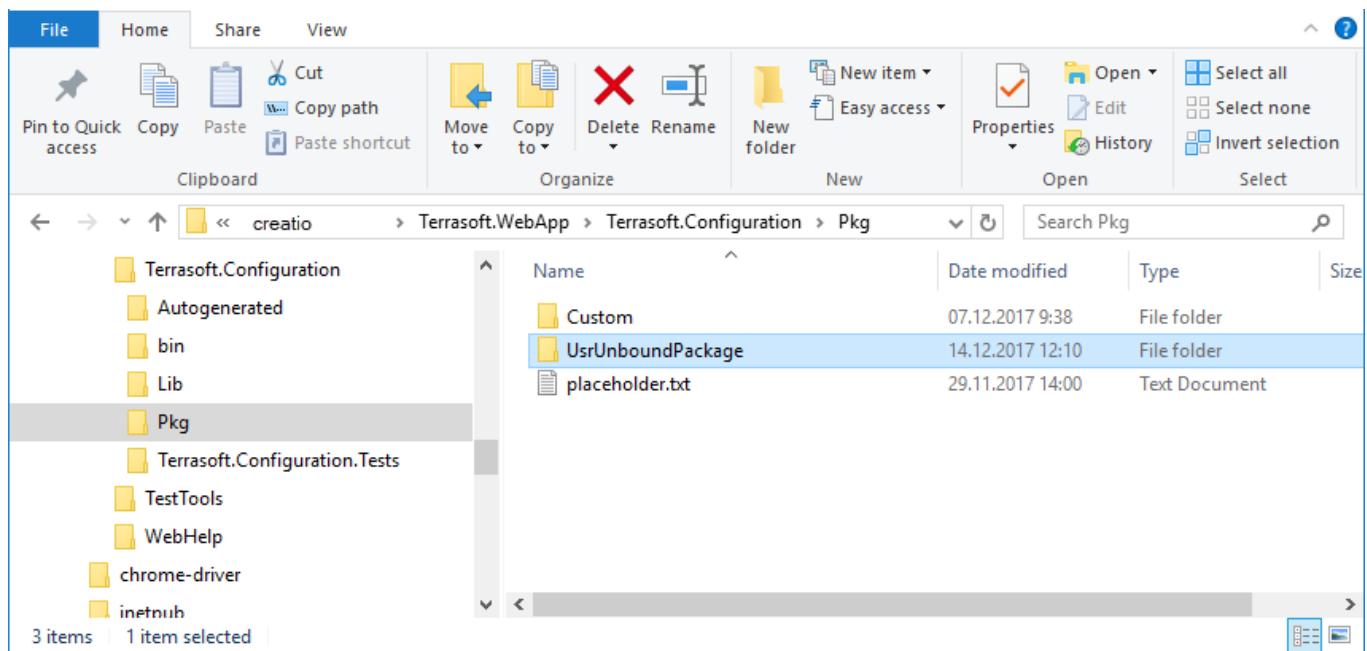
In the [Configuration] section, run the [Download packages to file system] command (Fig. 2).

Fig. 2. The [Download packages to file system] command



As a result, the package will be exported to the following catalog:
...|Terrasoft.WebApp|Terrasoft.Configuration|Pkg|UsrUnboundPackage (Fig. 3).

Fig. 3. The package in the file system

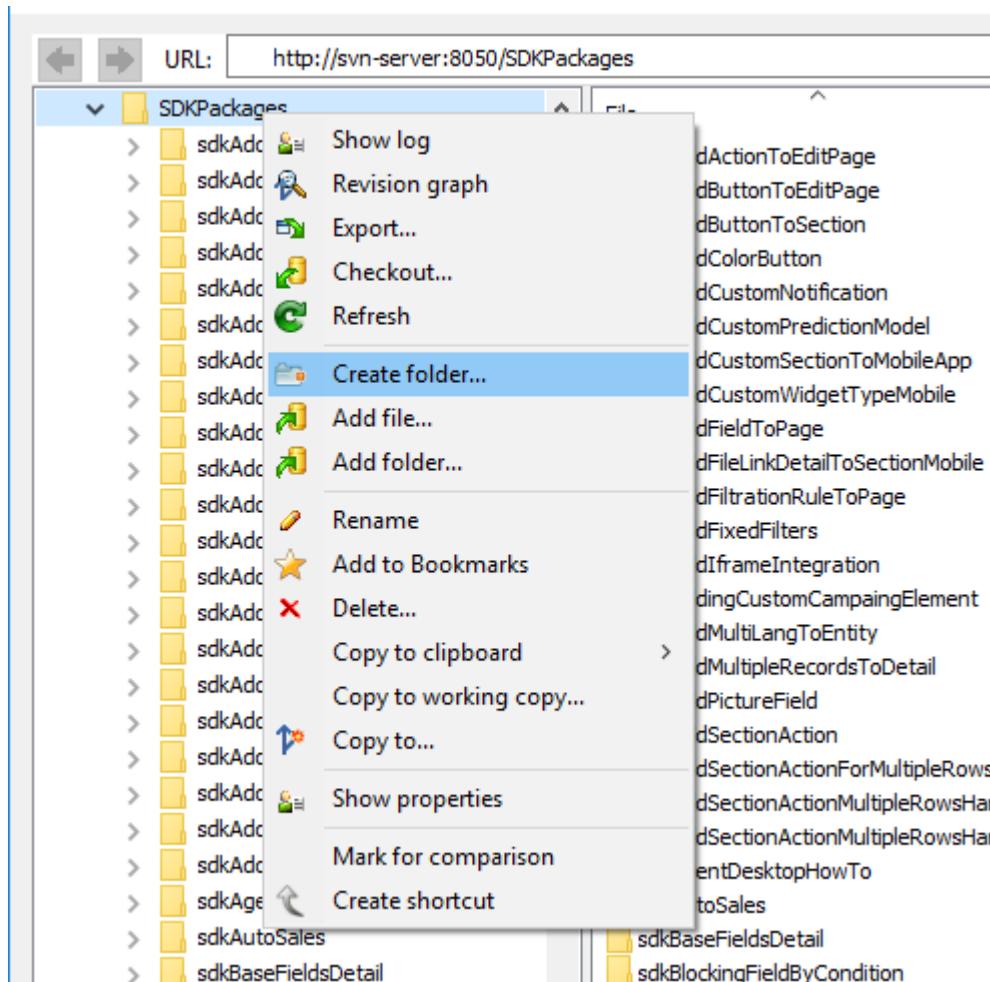


3. Create the catalogs for the package in the SVN repository.

To create the package catalogs using an SVN client (such as [TortoiseSvn](#)), go to the repository and add a catalog (Fig. 4) with the same name as the package.

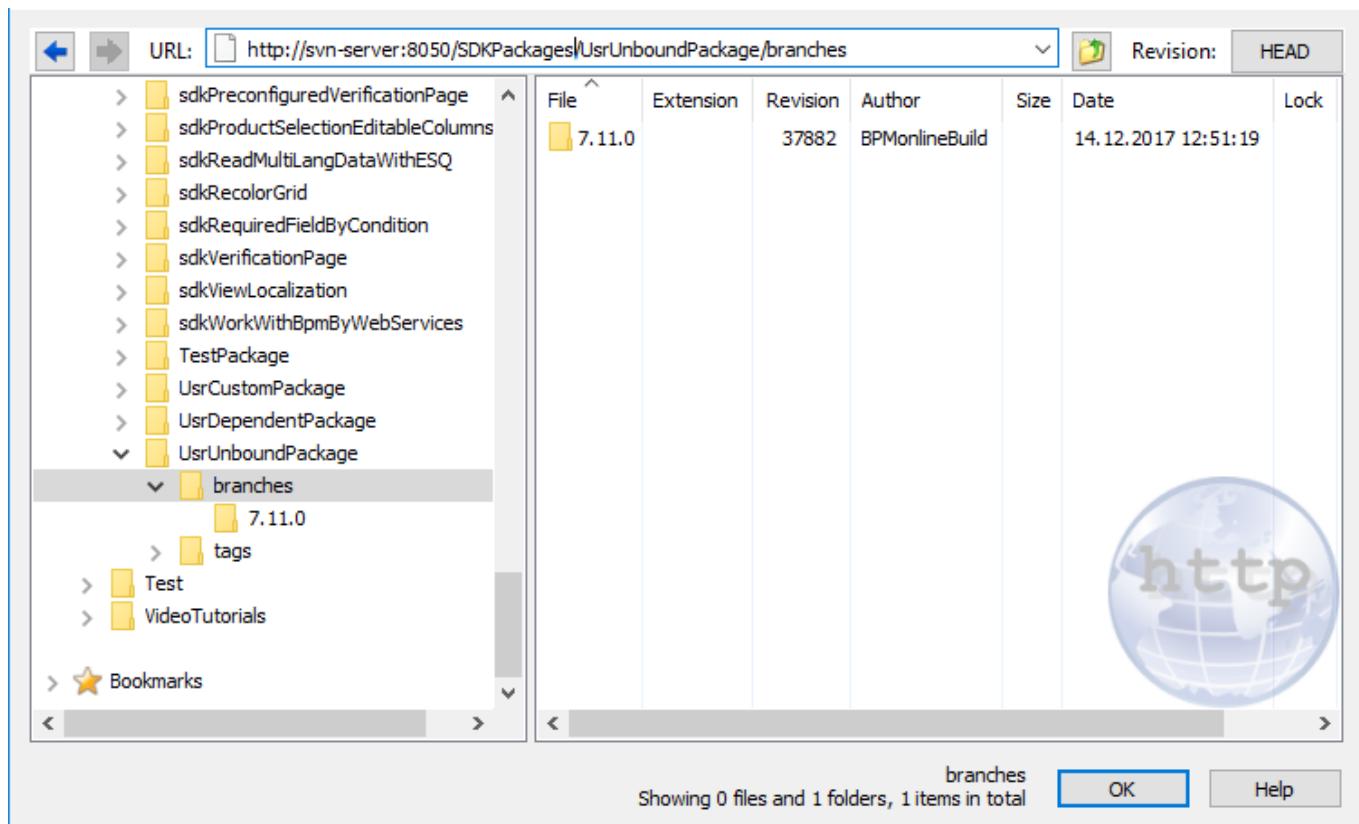
This article contains only general instructions for working with the SVN client. For in-depth instructions on working with the repository via TortoiseSvn are available in the [official TortoiseSvn documentation](#).

Fig. 4. Creating a catalog in the SVN repository



Create the *branches* and *tags* sub-folders in the package catalog, i.e., reproduce Creatio **flat package structure**. In the *branches* catalog, create a package version catalog, i.e., *7.11.0* (Fig. 5).

Fig. 5. Flat package structure in the repository



Reproducing the flat package structure is required only if you plan on using Creatio built-in tools for working with SVN.

4. Create a working copy of the package branch.

To create a working copy of the version-controlled package branch, export the catalog whose name matches the package version number (the [SVN Checkout...] command, Fig. 6) to the package's catalog in the file system.

Fig. 6. Running the [SVN Checkout] action

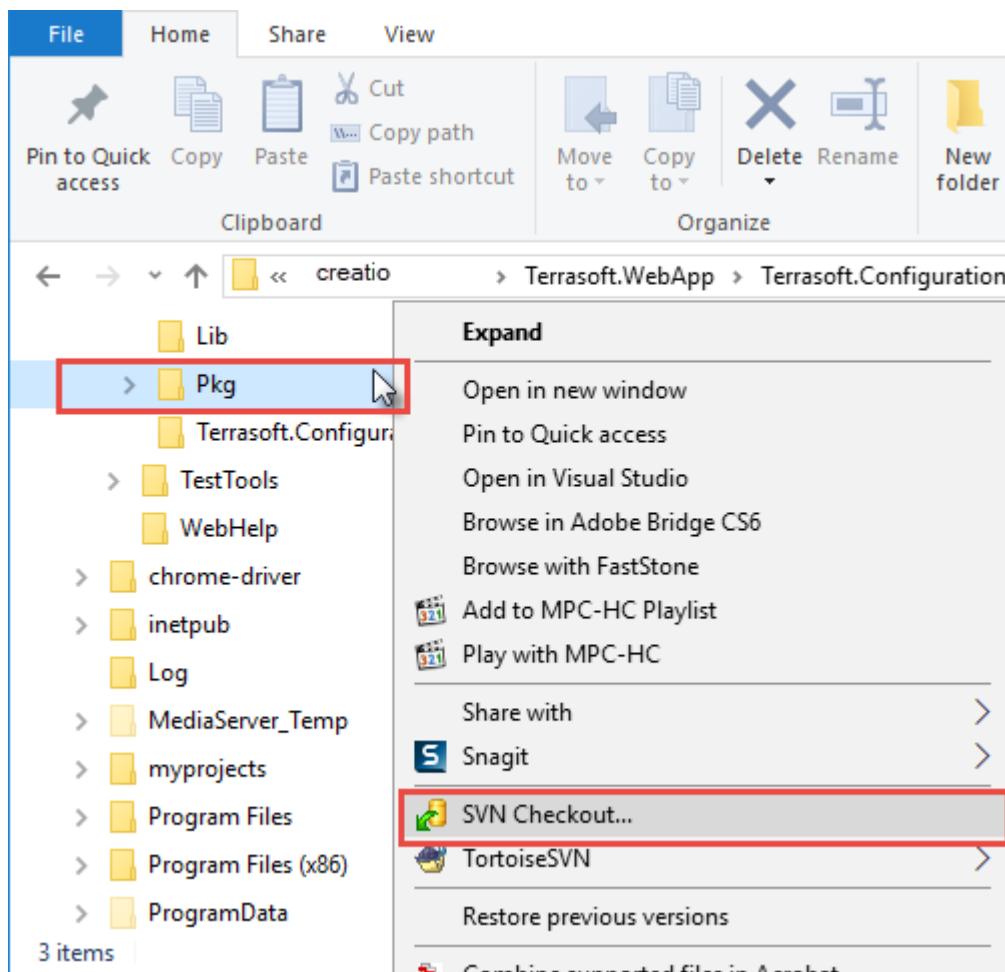


Fig. 7. Exporting working copy of a version-controlled package branch

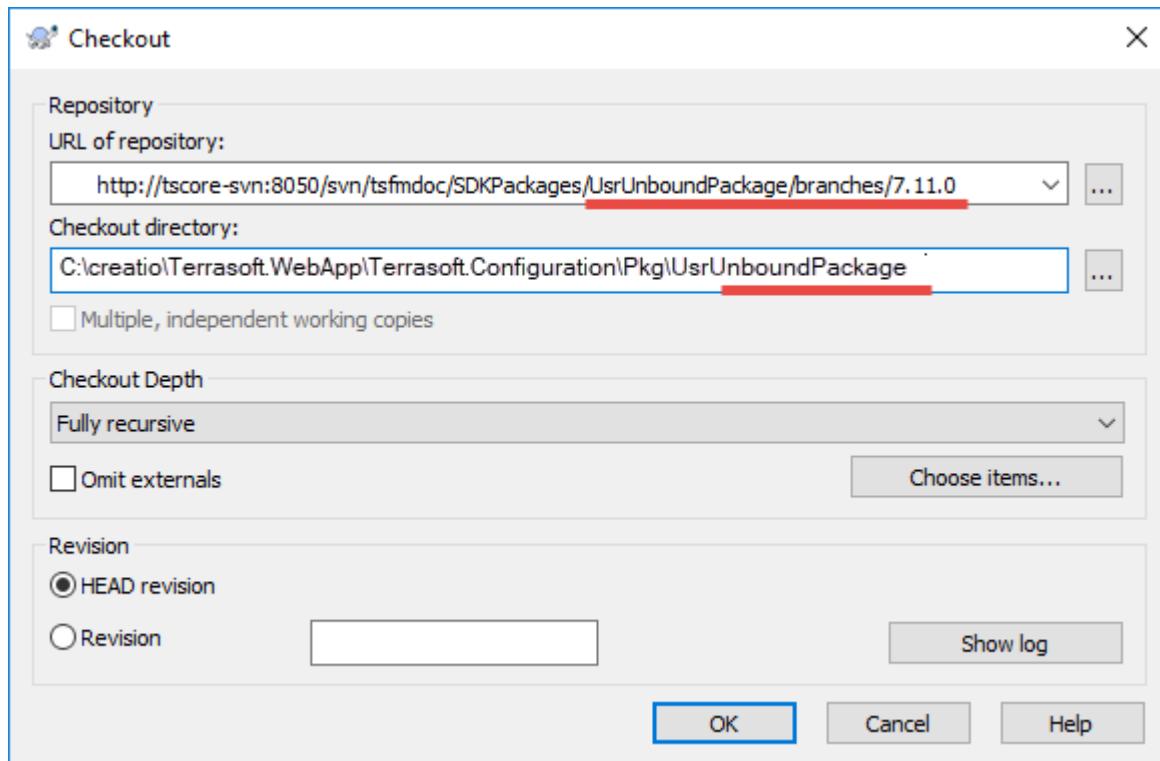
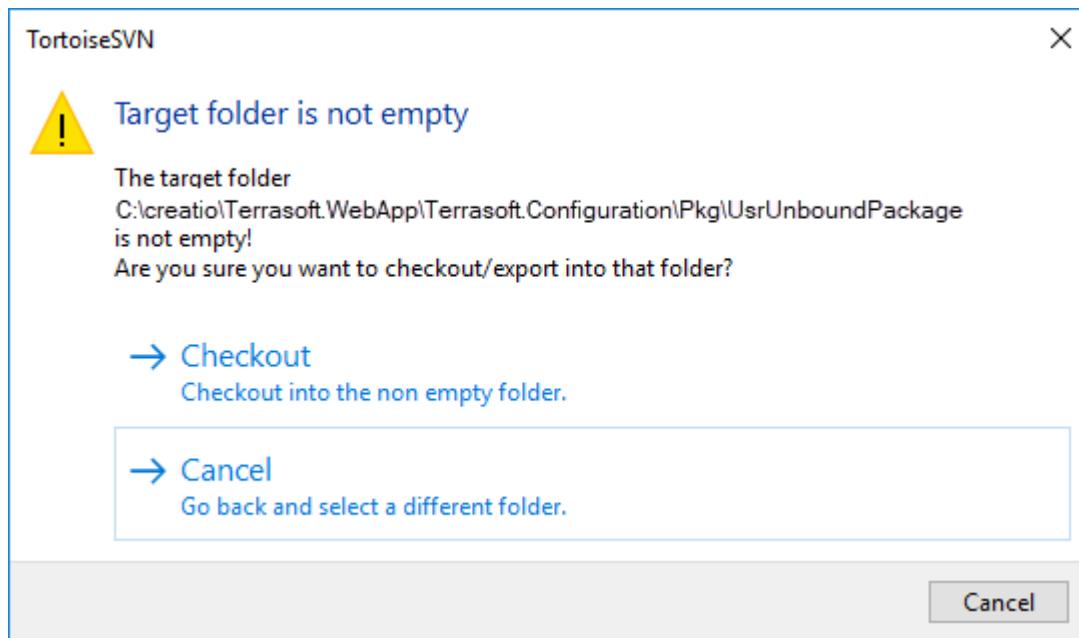


Fig. 8. Confirming the export to an existing catalog



As a result, the package catalog in the file system (...|Terrasoft.WebApp|Terrasoft.Configuration|Pkg|UsrUnboundPackage) will become a working copy of a 7.11.0 package in the repository (Fig. 9).

Fig. 9. A catalog, connected to the SVN repository

Custom	07.12.2017 9:38	File folder
UsrUnboundPackage	14.12.2017 12:55	File folder
placeholder.txt	29.11.2017 14:00	Text Document

5. Commit the package catalog in the repository.

To commit the contents of a package catalog to the repository, run the TortoiseSVN [Add...] command (Fig. 10 and 11), then run the [SVN Commit...] command (Fig. 12).

Fig. 10. The [Add] command

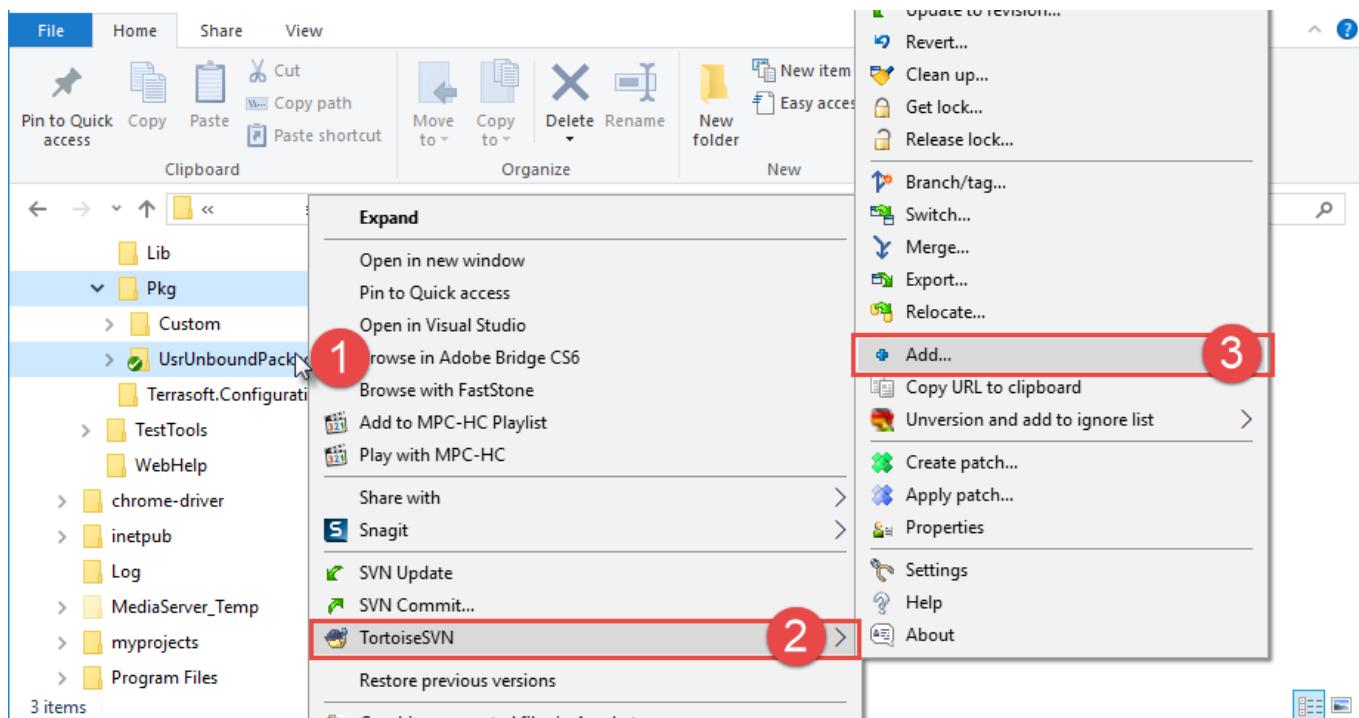


Fig. 11. Dialog for selecting items to add to the repository

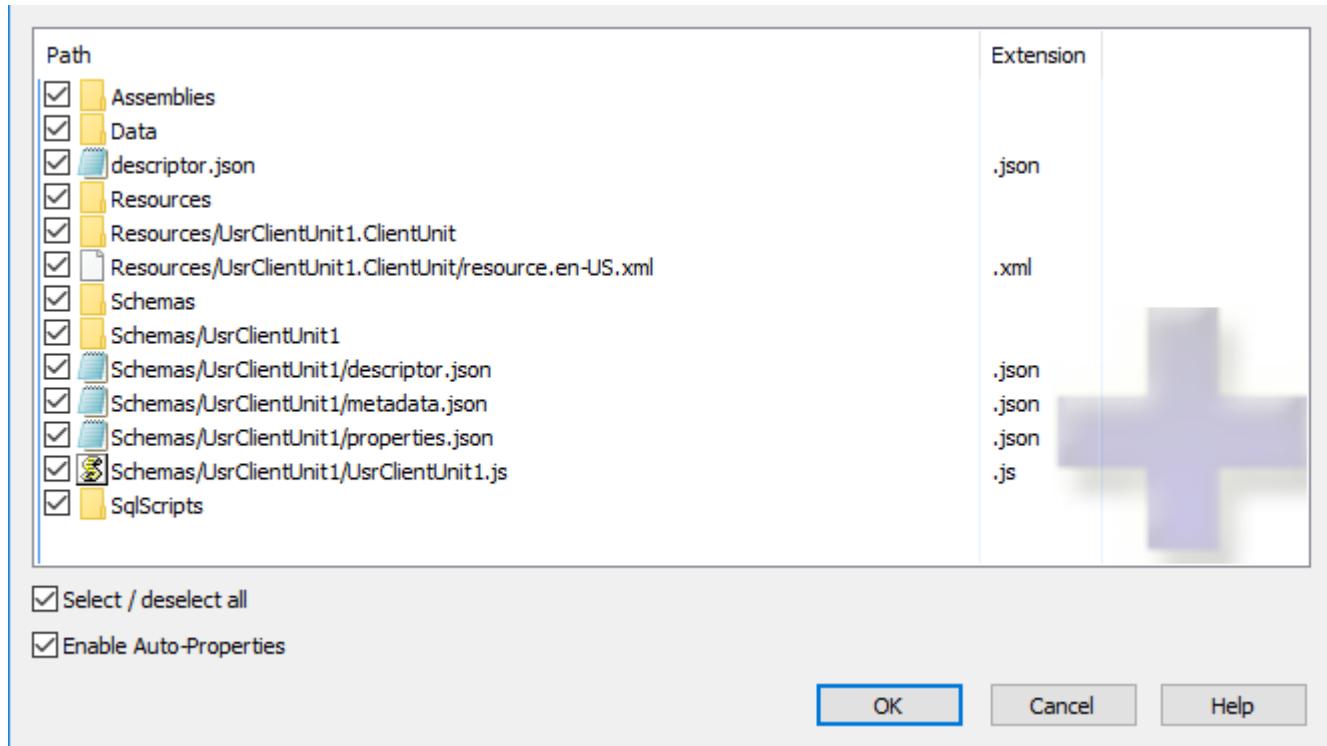
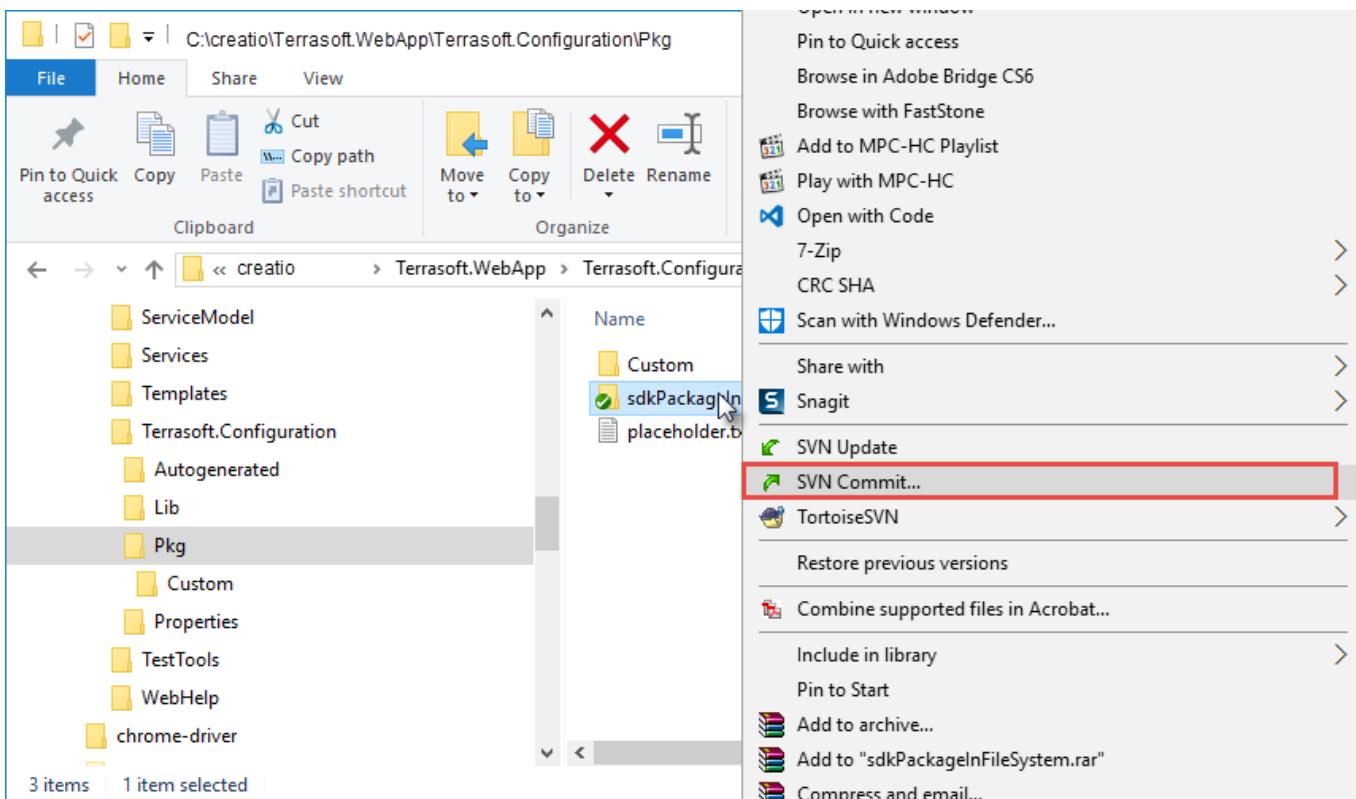


Fig. 12. Committing to the repository



As a result, all package contents will be bound to the SVN repository (Fig. 13).

Fig. 13. A package in SVN

The screenshot shows a TortoiseSVN commit dialog with the following details:

File	Extension	Revision	Author	Size	Date
descriptor.json	.json	37883	BPMonlineBuild	257 bytes	14.12.2017 13:19:24
metadata.json	.json	37883	BPMonlineBuild	516 bytes	14.12.2017 13:19:24
properties.json	.json	37883	BPMonlineBuild	94 bytes	14.12.2017 13:19:24
UsrClientUnit1.js	.js	37883	BPMonlineBuild	62 bytes	14.12.2017 13:19:24

Commit message: Add new package schemas

Repeat step 5 to commit new package schemas in the SVN repository.

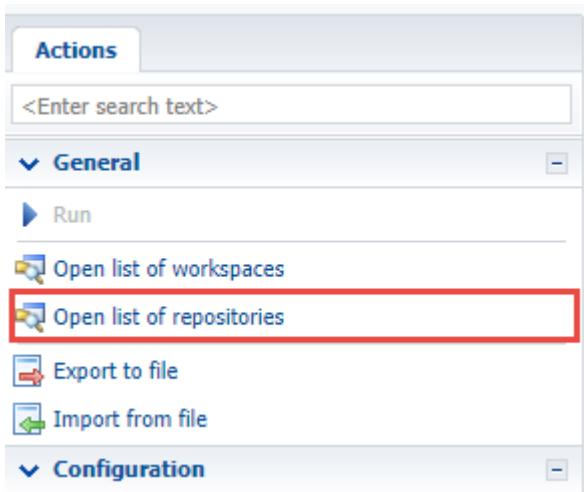
Alternative implementation

1. Add information about the SVN repository in the [Configuration] section.

If the information about the needed repository has not been added in the [Configuration] section:

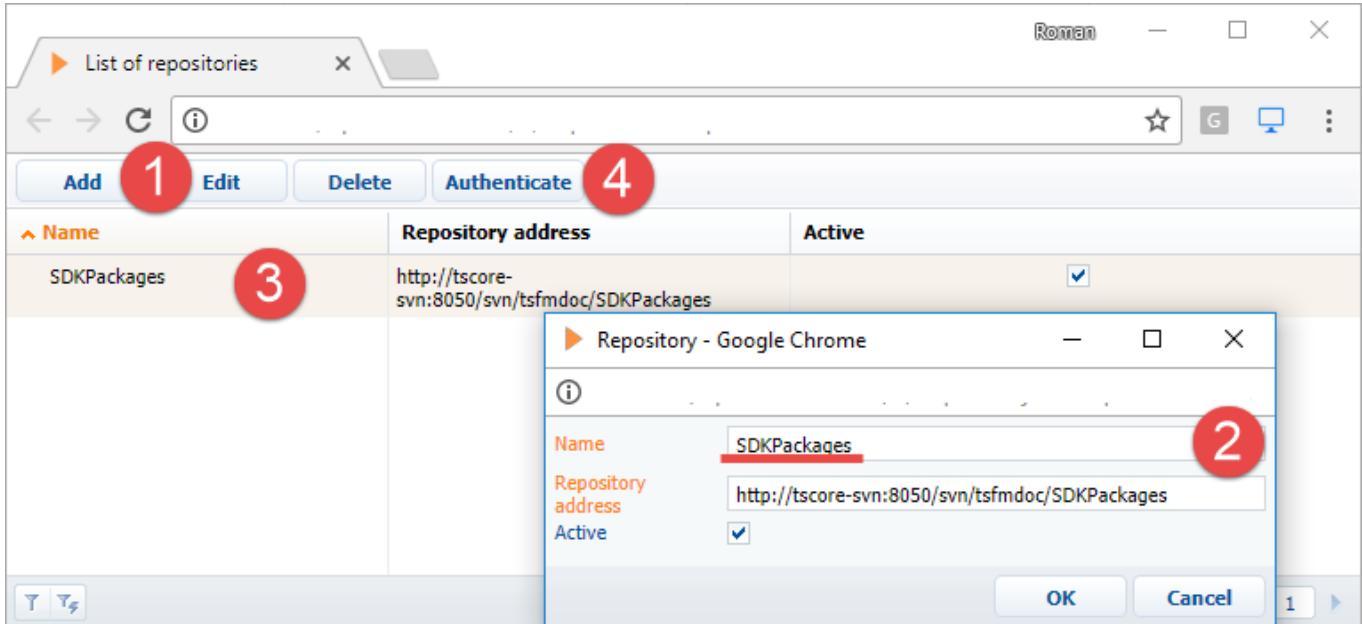
- Run the [Open list of repositories] command (Fig. 14).

Fig. 14. The [Open list of repositories] command



2. In the opened [List of repositories], use the [Add] command (Fig. 15, 1) to add necessary repository (Fig. 15, 2). After this, repository information will display in the [List of repositories] window (Fig. 15, 3). Select the string with information on the repository and perform authentication (Fig. 15, 4).

Fig. 15. The actions of the [Open list of repositories] window



2. Bind the repository to the package.

Execute repository binding SQL query. Example of the SQL query to binding repository to a package is as follows:

```
UPDATE SysPackage
SET
    [SysRepositoryId] =
    (
        select top 1 Id from SysRepository
        where Name = 'SDKPackages' -- Repository name.
    )
where [Name] = 'UsrUnboundPackage' -- Name of the custom package.
```

In this query, "SDKPackages" is the repository name (Fig. 15, 2), and "UsrUnboundPackage" is the name of the custom package.

To apply the changes in the repository, log out from the application and then log back in.

3. Export the package to the file system.

In the [Configuration] section, run the [Download packages to file system] command (Fig. 2). As a result, the bound package will be exported to the following catalog:

...|Terrasoft.WebApp|Terrasoft.Configuration|Pkg|UsrUnboundPackage (Fig. 9).

See also

- **Working with SVN in the file system**
- **Creating a package in the file system development mode**
- **Installing an SVN package in the file system development mode**
- **Updating and committing changes to the SVN from the file system**
- **Creation of the package and switching to the file system development mode**

Updating and committing changes to the SVN from the file system

Beginner

Easy

Medium

Advanced

Introduction

Different developers can use the SVN to develop the functionality in the same file. The possible situation when one of the developers modifies the file first and the next developer could overwrite file with a new version and the modifications of the first developer will be lost. The modifications of the first developer are saved by the system, but these modifications will be lost in the last revision of the file. To avoid this, use one of the following versioning models: "Lock-Modify-Unlock" or "Copy-Modify-Merge" (see "**Working with SVN in the file system**").

Regardless of the chosen solution, the actions for updating and committing changes to the SVN are almost the same.

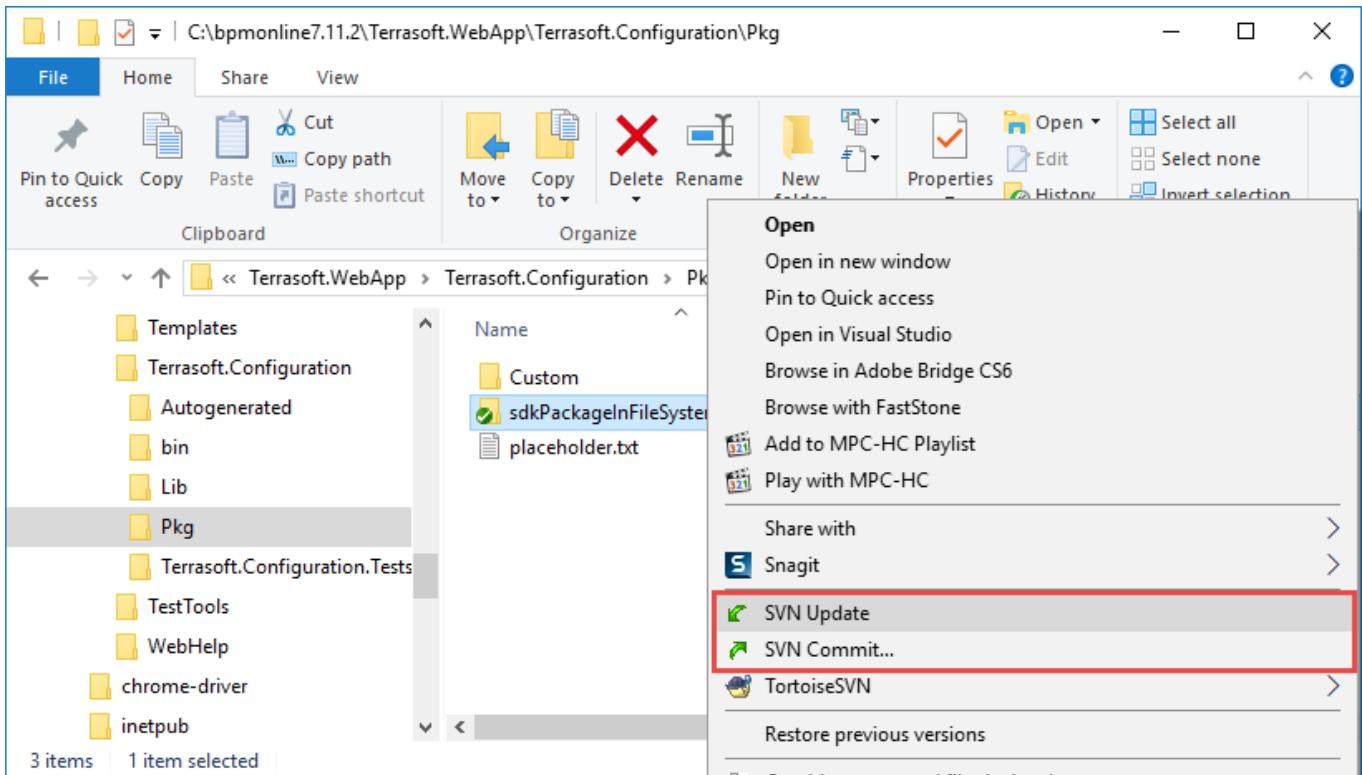
Case description

The *sdkPackageInFileSystem* custom package is implemented in the Creatio configuration (see "**Creating a package in the file system development mode**"). Update the package in the file system development mode and after that commit it to the SVN.

Updating the package

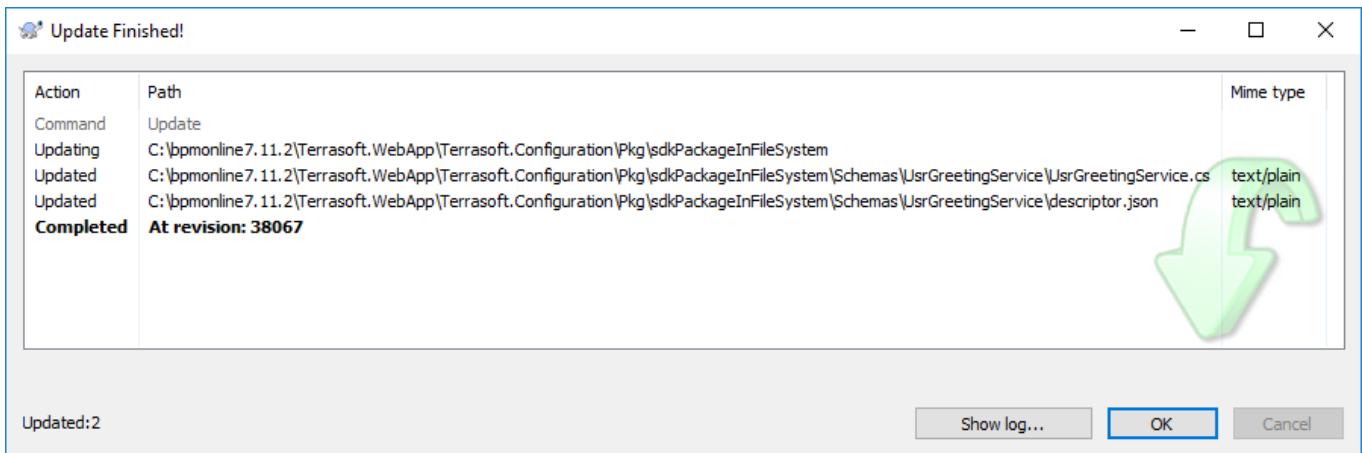
To get the latest package revision select the corresponding package in the ...|Terrasoft.WebApp|Terrasoft.Configuration|Pkg folder and perform the [SVN Update] action (Fig.1).

Fig. 1. Running the [SVN Update] action



After the action is executed, the window with update results will be displayed (Fig. 2).

Fig. 2. Update results



As a result the descriptor.json file (the package modification date has changed) and the UsrGreetingService schema were modified. The source code of the schema is available below:

```
namespace Terrasoft.Configuration
{
    using System.ServiceModel;
    using System.ServiceModel.Activation;
    using System.ServiceModel.Web;
    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Required)]
    public class UsrGreetingService : System.Web.SessionState.IReadOnlySessionState
    {
        [OperationContract]
        [WebInvoke(Method = "GET", UriTemplate = "Hello")]
        public string TestHello()
        {
```

```
        return "Hello!";
    }
}
}
```

To implement changes to the application database, execute the [Update packages from file system] action in the **[Configuration] section**. If the schemas of an object or a source code were modified, execute steps 3 – 6 of the “[Installing an SVN package in the file system development mode](#)” article to apply changes.

Package contents modifications

After updating the package you can modify its contents. For example, add the TestHelloWorld() method to the source code of the UsrGreetingService.cs schema.

```
namespace Terrasoft.Configuration
{
    using System.ServiceModel;
    using System.ServiceModel.Activation;
    using System.ServiceModel.Web;
    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class UsrGreetingService : System.Web.SessionState.IReadOnlySessionState
    {
        [OperationContract]
        [WebInvoke(Method = "GET", UriTemplate = "Hello")]
        public string TestHello()
        {
            return "Hello!";
        }

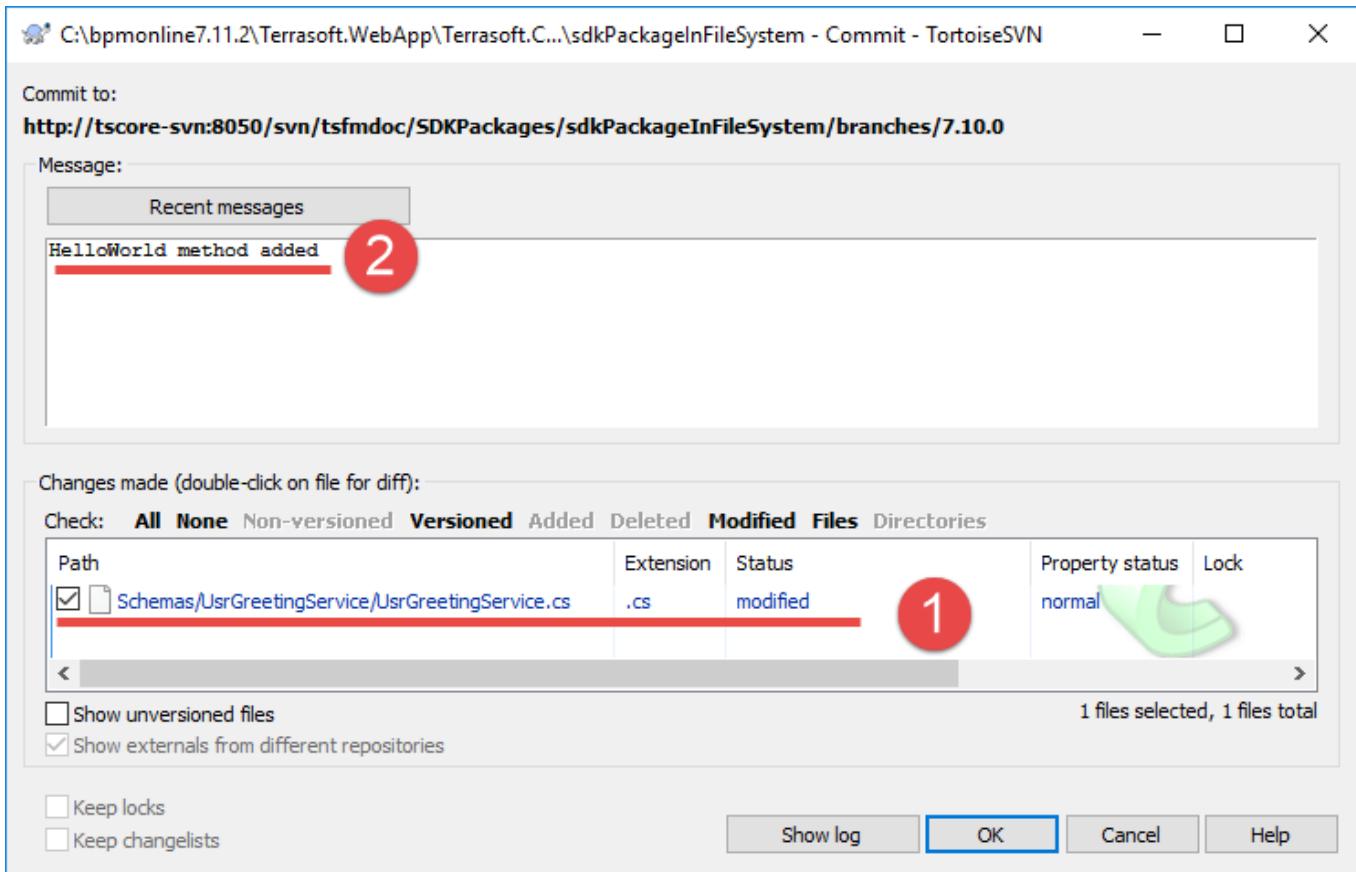
        [OperationContract]
        [WebInvoke(Method = "GET", UriTemplate = "HelloWorld")]
        public string TestHelloWorld()
        {
            return "Hello world!";
        }
    }
}
```

Committing a package to storage

To commit the modifications in the SVN select the corresponding package in the ..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg folder and perform the [SVN Commit...] action. (Fig. 1).

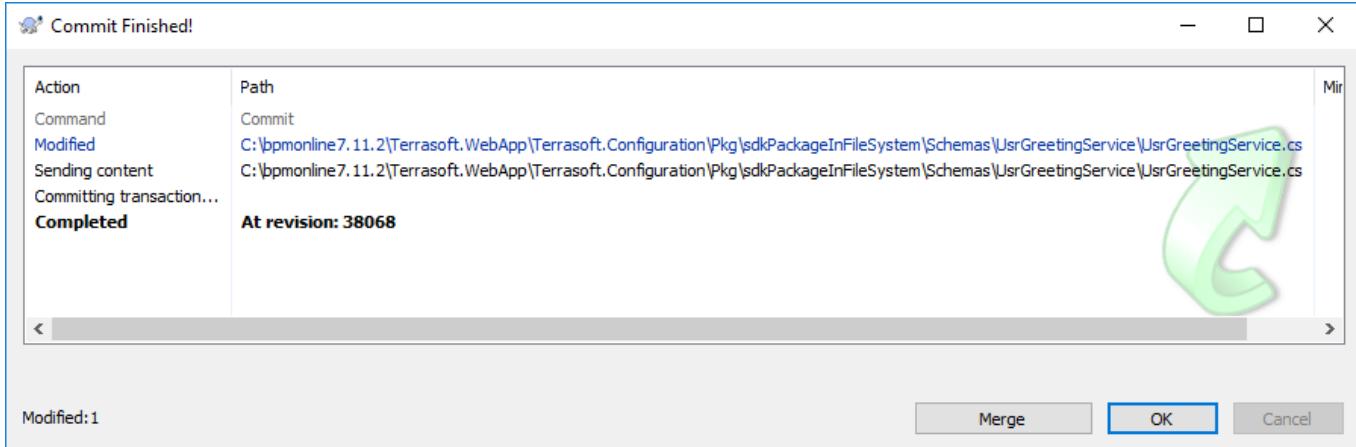
After that the revision properties window with modified files (1) will be displayed (Fig. 3). In this window you can add the log message with description of changes for the current revision (2). Click the [OK] button to start committing.

Fig. 3. Revision properties window



Committing process is displayed in the information window (Fig. 4).

Fig. 4. Result of committing



See also

- **Working with SVN in the file system**
- **Creating a package in the file system development mode**
- **Installing an SVN package in the file system development mode**
- **Binding an existing package to SVN**
- **Creation of the package and switching to the file system development mode**

Creation of the package and switching to the file system development mode

Beginner

Easy

Medium

Advanced

Introduction

After performing the [Download packages to file system] action in the development in the file system mode all custom packages will be uploaded to the ..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg folder. The content of the custom package uploaded to the file system will not be bound to the SVN storage even if the package is bound to the storage in the [Configuration] section.

Fill out the [Revision control system storage] field to bind a package to the SVN storage (see “[Creating a package for development](#)”). A working copy of the package will be created in the file system. A path to the folder where the work copies of the packages are being created is specified in the *defPackagesWorkingCopyPath* setting in the *ConnectionStrings.config* file (see “[What parameters are used in ConnectionStrings.config](#)”).

This feature can be used to create a package bound to SVN and used for development in the file system. If you specify a path to the ..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg folder in the *defPackagesWorkingCopyPath* setting, the package will be automatically bound to the SVN storage after it is uploaded to the file system.

To do this:

1. Specify a path to the ..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg. folder in the *defPackagesWorkingCopyPath* setting.
2. In the development mode, use the built-in tools to create a package in the [Configuration] section bound to the SVN storage.
3. Commit the package in the storage in the [Configuration] section.
4. Switch to the file system development mode
5. Export the package to the file system.
6. Add new elements of the package to the SVN storage.

Case description

In the development mode, use the built-in tools to create a custom package in the [Configuration] section bound to the SVN storage. Configure the Creatio so that the content of the package in the development mode was bound to the SVN storage after the package upload.

The case requires understanding of the difference between development modes. In general: in the development mode in the file system, it is necessary to work with the SVN storage only from the file system, and in the development mode using the built-in tools it is necessary to work with SVN only via the built-in tools of the [Configuration] section.

Case implementation algorithm

1. Modify the *defPackagesWorkingCopyPath* setting

Specify a path to the ..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg. folder in the *defPackagesWorkingCopyPath* setting of the *ConnectionStrings.config* file. Example:

```
<?xml version="1.0" encoding="utf-8"?>
<connectionStrings>
  ...
    <add name="defPackagesWorkingCopyPath"
        connectionString="C:\creatio\Terrasoft.WebApp\Terrasoft.Configuration\Pkg" />
  ...
</connectionStrings>
```

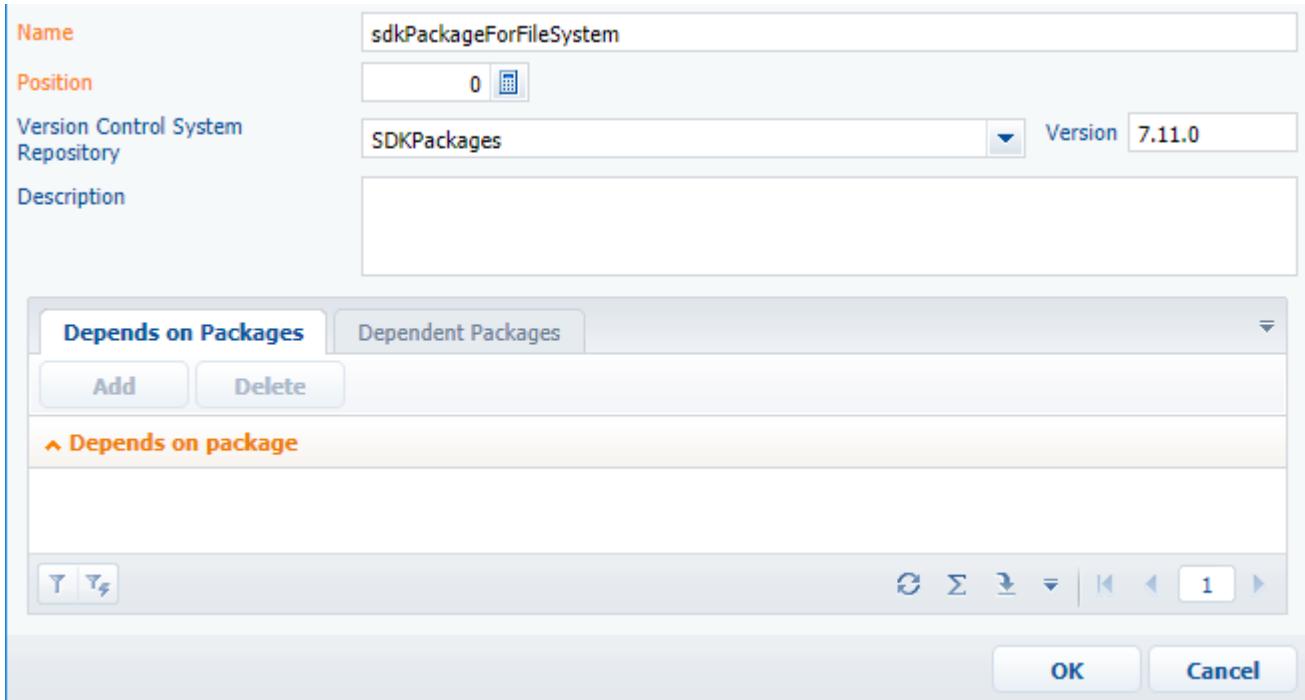
This modification enables to combine the folder with working copies of custom packages with the folder in which the packages will be uploaded in the development in the file system mode.

2. Create a custom package

In the development mode, create a custom package in the [Configuration] section bound to the SVN storage via the

built-in tools. Please refer to “**Creating a package for development**” for any details. Specify the name, storage and version of the created package (Fig. 1).

Fig. 1. Package properties



After creation of the package add necessary dependencies from the base packages (see “**Package dependencies**”).

3. Commit a package to storage

To commit a package to the storage perform the [Commit package to repository] action (Fig. 2). In the dialog box (Fig. 3) add the description of changes (1) and press the [OK] button. After the commit is complete, the corresponding message will appear (3).

Fig. 2. The [Commit package to storage] action

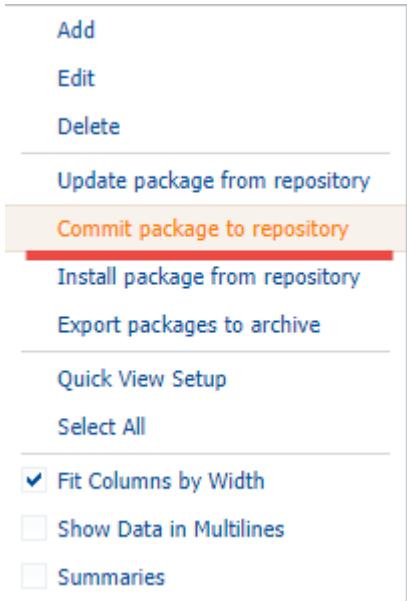
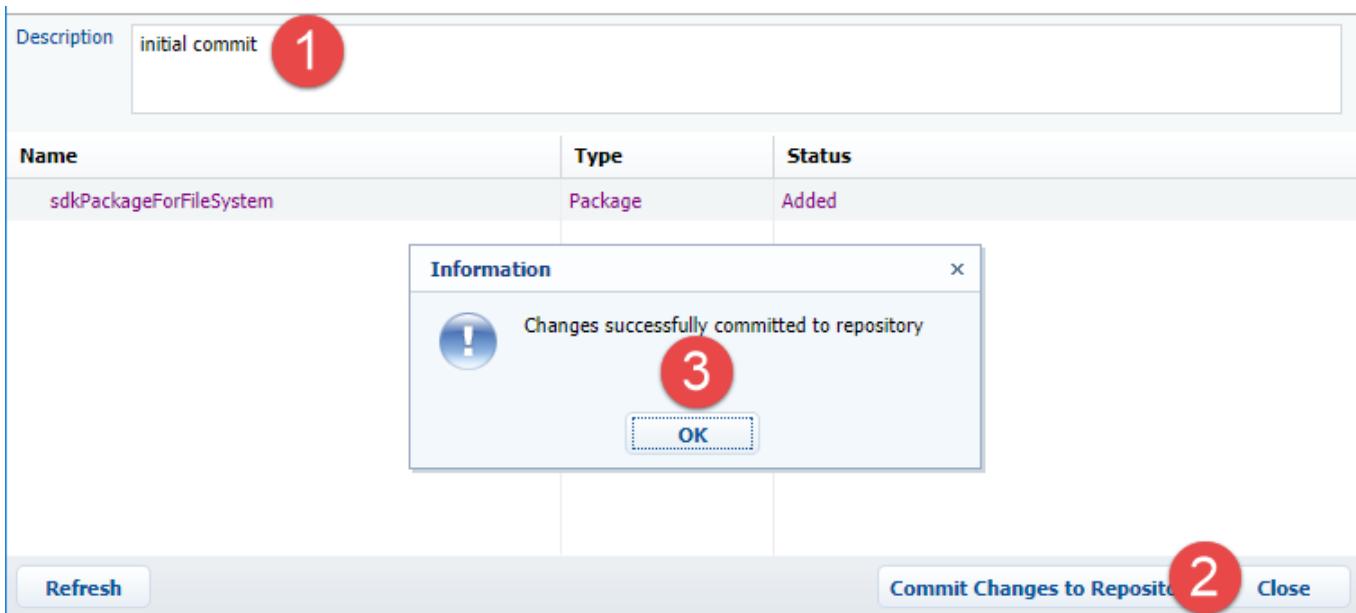
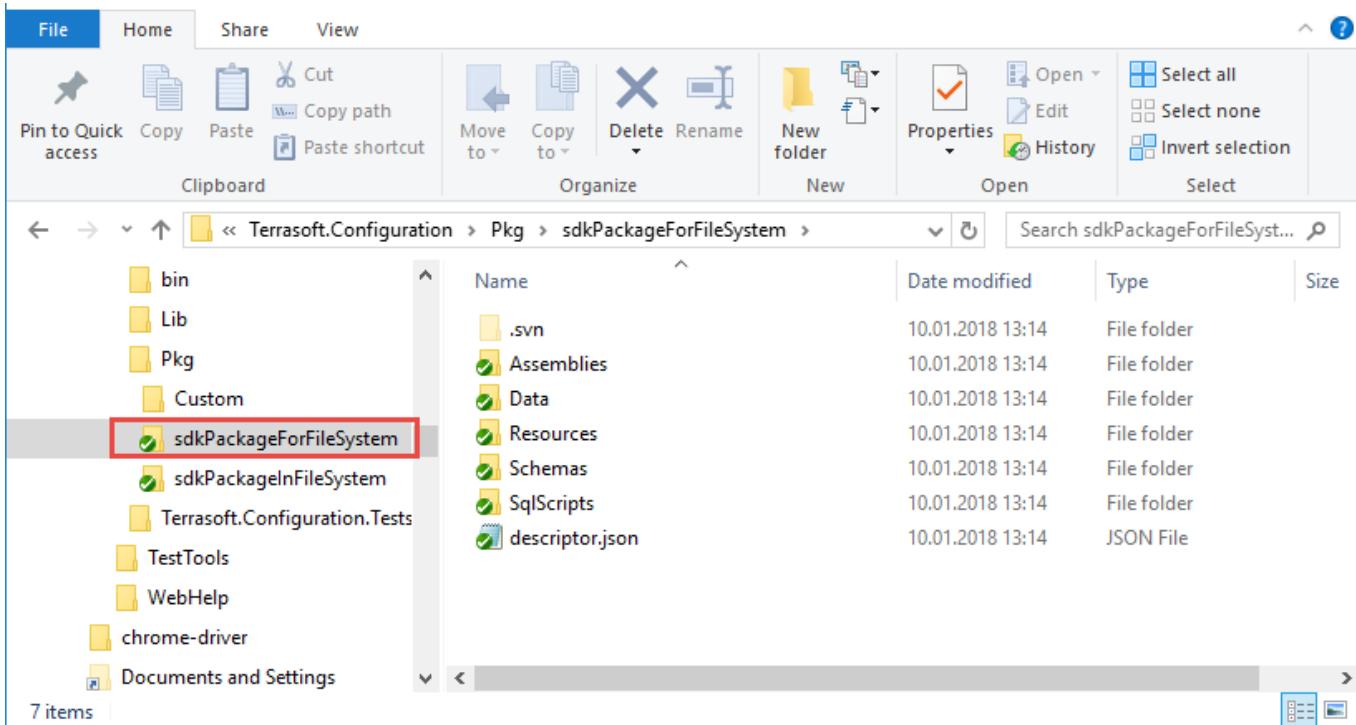


Fig. 3. Commit properties window



After the commit is complete, the ..\Terrasoft.WebApp\Terrasoft.Configuration\Pkg catalog will contain a local working copy of the package (Fig. 4).

Fig. 4. Package working copy



4. Switch to the file system development mode.

To enable the development in the file system mode, edit the Web.config file in the application root folder and set *enabled* attribute of the *fileDesignMode* element to *true*.

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

Disable the using of the static content (see “**Client static content in the file system**”).

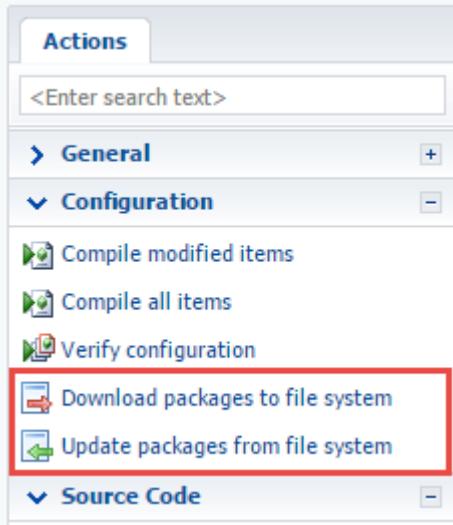
After the development in the file system mode is enabled, two buttons will appear on the [Actions] tab in the [Configuration] section (Fig. 1):

- [Download packages to file system] – exports the packages from the application database to the following

directory: ...|Terrasoft.WebApp|Terrasoft.Configuration|Pkg.

- [Update packages from file system] – imports the packages from the following catalog: ...|Terrasoft.WebApp|Terrasoft.Configuration|Pkg to the database.

Fig. 5. Actions in the [Configuration] section for development in the file system



5. Export the package to the file system.

If the package content was not changed after committing to the storage, this action is optional.

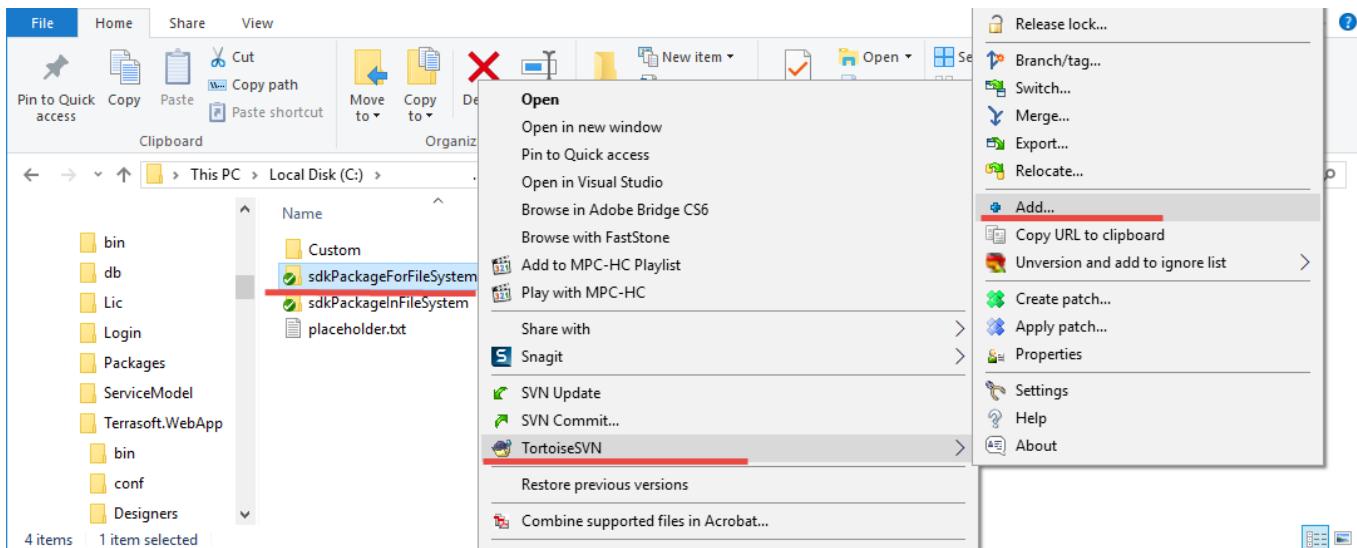
Perform the [Download packages to file system] action to download packages to the file system. As a result, all elements of the package that were created or modified via built-in tools in the [Configuration] section will be downloaded to the file system to the ..|Terrasoft.WebApp|Terrasoft.Configuration|Pkg folder.

Since in the development in the file system mode the built-in tools for working with SVN are disabled, the new elements of the package will not be bound to the storage.

6. Add new elements of the package to the SVN storage

To add the new elements of the package to the storage, select the folder of the package working copy and perform the [Add...] command of the SVN application (for example, *TortoiseSVN*) (Fig. 6).

Fig. 6. Command of adding the elements to the storage



After this the dialog box with selection of the elements to add will be displayed (Fig. 7). Select the necessary elements and click the [OK] button, after that the window with the results of command execution will be displayed (Fig. 8).

Fig. 7. Selection dialog box

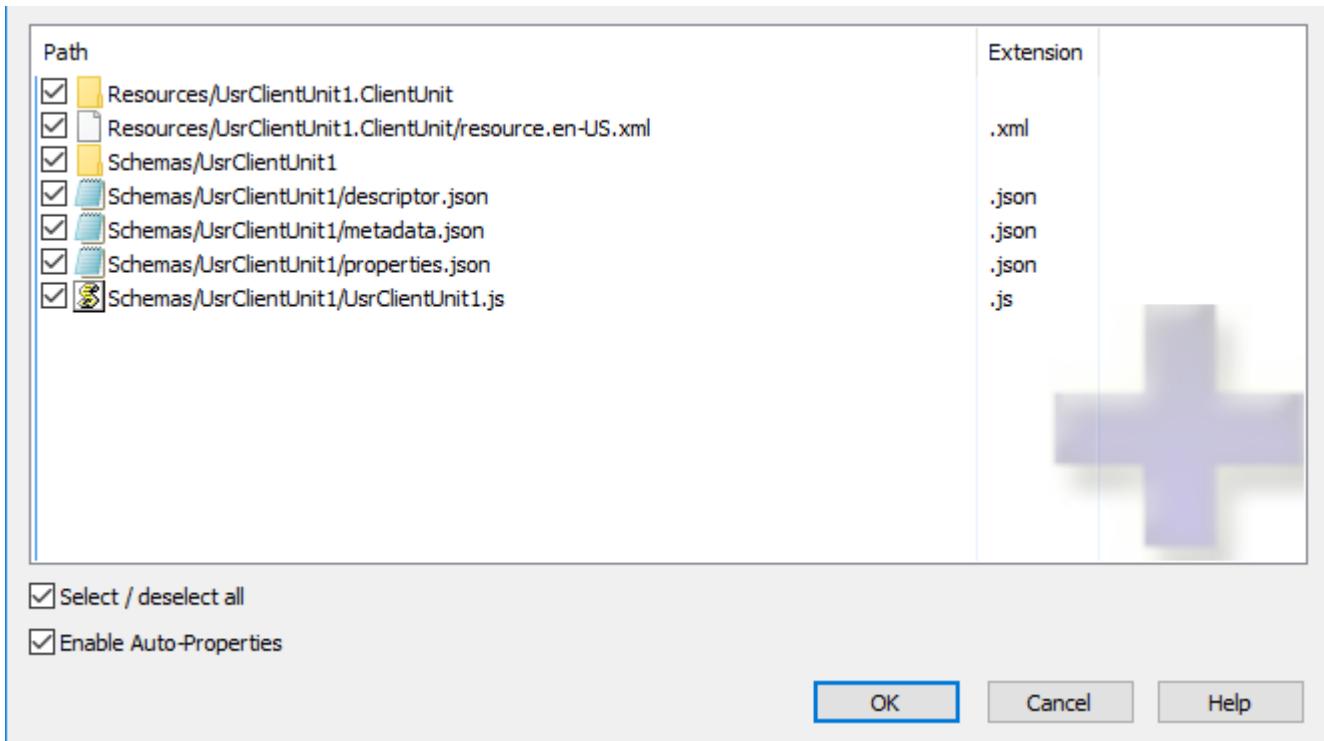


Fig. 8. Information window

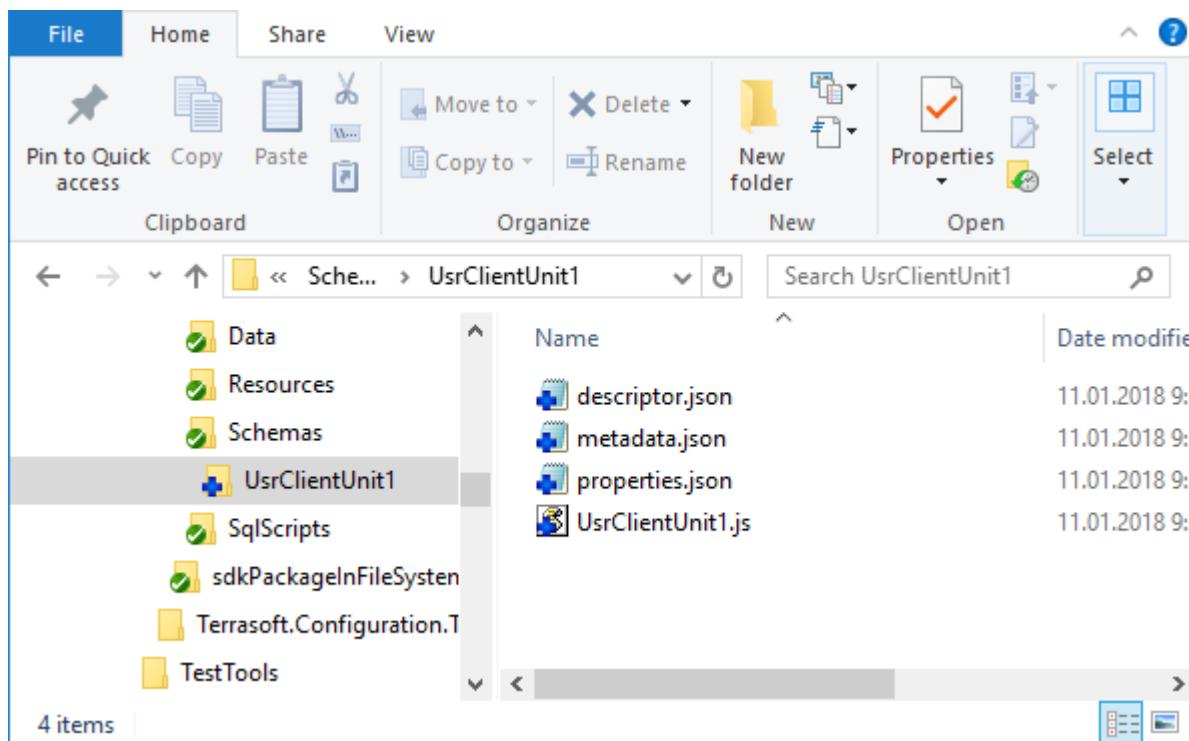
Action	Path	Mime
Command	Add	
Added	C:\bpmonline7.11.2\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageForFileSystem\Resources\UsrClientUnit1.ClientUnit	
Added	C:\bpmonline7.11.2\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageForFileSystem\Resources\UsrClientUnit1.ClientUnit\resource.en-US.xml	text
Added	C:\bpmonline7.11.2\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageForFileSystem\Schemas\UsrClientUnit1	
Added	C:\bpmonline7.11.2\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageForFileSystem\Schemas\UsrClientUnit1\descriptor.json	text
Added	C:\bpmonline7.11.2\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageForFileSystem\Schemas\UsrClientUnit1\metadata.json	text
Added	C:\bpmonline7.11.2\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageForFileSystem\Schemas\UsrClientUnit1\properties.json	text
Added	C:\bpmonline7.11.2\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageForFileSystem\Schemas\UsrClientUnit1\UsrClientUnit1.js	text
Completed!		

Added: 7

OK **Cancel**

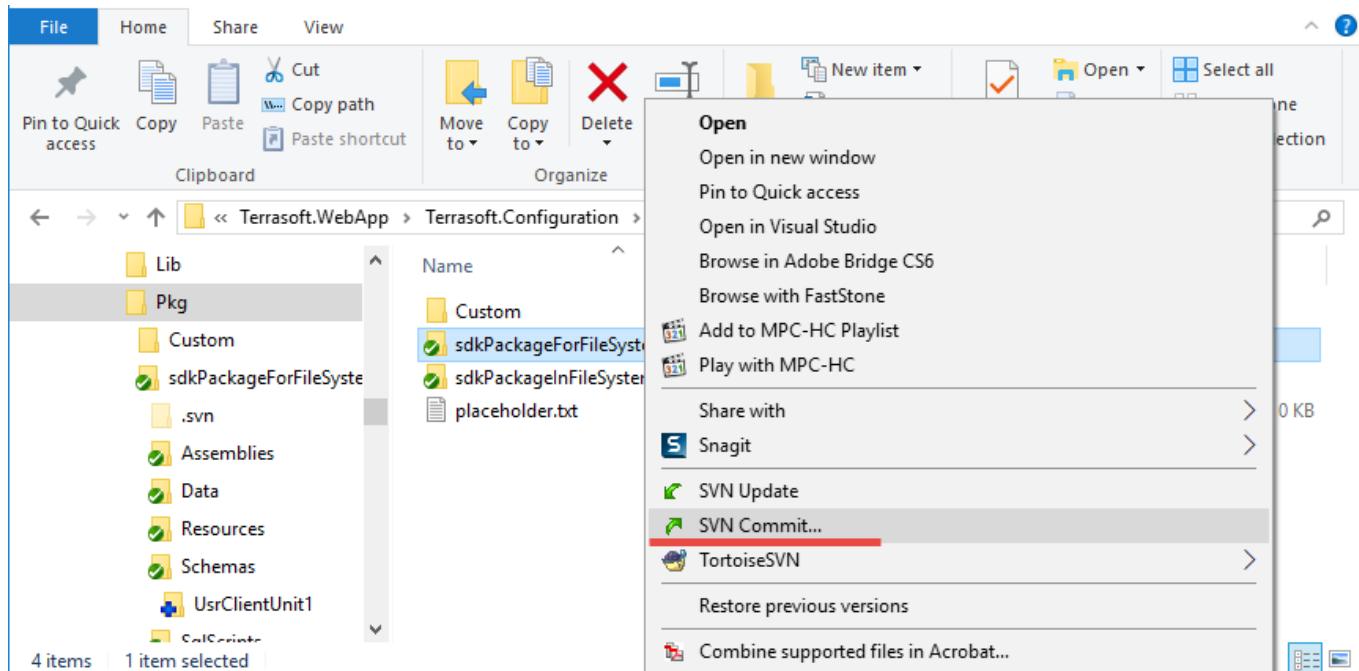
Added elements will be marked as bound but not committed to the SVN storage (Fig. 9).

Fig. 9. Displaying of the added but not committed package elements



Perform the [SVN Commit...] command to commit all modified elements of the package in the storage. (Fig. 10).

Fig. 10. Command of adding the elements to the storage



More information about committing the elements in the storage can be found in the “[Updating and committing changes to the SVN from the file system](#)” article.

See also

- [Working with SVN in the file system](#)
- [Creating a package in the file system development mode](#)
- [Installing an SVN package in the file system development mode](#)
- [Binding an existing package to SVN](#)
- [Updating and committing changes to the SVN from the file system](#)

Working with Git

Beginner

Easy

Medium

Advanced

Introduction

The main distinction of Git compared to any other version control system (including Subversion) is the approach to working with data. Most version control systems store data as a series of file changes. CVS, Subversion, Perforce, Bazaar and other version control systems store data as a set of files and individual file changes over time (i.e., utilize “delta-based” version control).

Git approach, on the other hand, is based on taking a series of file snapshots. Each time you commit your changes, Git creates a snapshot of each file and keeps a link to that snapshot. To boost efficiency, Git does not take snapshots of unmodified files. Rather, it creates a link pointing to an existing snapshot. Git represents data as a stream of snapshots.

Your files can have one of three statuses in Git: “committed,” “modified” or “staged.” A committed file is saved in your local repository. Modified files are different from the original files, but they are not yet committed. Staged files are modified files that are marked to be included in the next commit.

You can find out more about installing and working with Git in the official documentation [on the Git website](#).

You can also use any suitable GUI to work with Git, such as [Sourcetree](#), which can be seen on the screenshots in this article.

Working with Git in Creatio

Integrated Creatio tools are only designed to work with the Subversion version control system. However, the version control integration mechanism is disabled when the development mode is on. In this mode, you can use any version control system, including Git.

The general procedure for working with Git

Step 1. Enable the file system development mode

To enable the file system development mode, edit the Web.config file in the application root folder and set the *enabled* attribute of the *fileDesignMode* element to *true*.

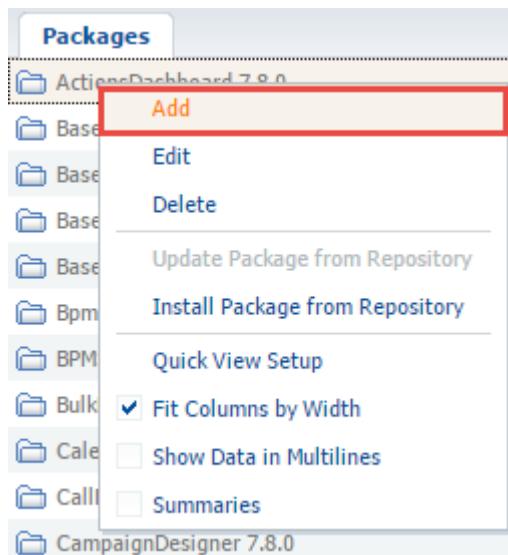
Currently, the file system development mode is not compatible with retrieving client content from pre-generated files. To ensure the correct operation of the file system development mode, you need to disable retrieving client static content from the file system. To disable this function, set the *UseStaticFileContent* flag in the Web.config file to *false*.

```
<fileDesignMode enabled="true" />
...
<add key="UseStaticFileContent" value="false" />
```

Step 2. Create a package without binding it to the repository

Go the [Packages] tab in the [Configuration] section and run the [Add] command in the context menu (fig. 1).

Fig. 1. – The [Add] package command



In the package summary (fig. 2), populate the primary fields of the package properties (see the “[Creating a package for development](#)” article) without specifying the name of the repository to bind the package.

Fig. 2. – Package summary

The screenshot shows the 'sdkPackageInFileSystem' package summary dialog. The 'Name' field is set to 'sdkPackageInFileSystem'. The 'Position' field is set to '0'. The 'Version Control System' section shows 'Repository' is empty and 'Version' is '1.0.0'. The 'Description' field contains 'Package in file system'. Below this, the 'Depends on Packages' tab is selected, showing an empty list of dependent packages. At the bottom are 'OK' and 'Cancel' buttons.

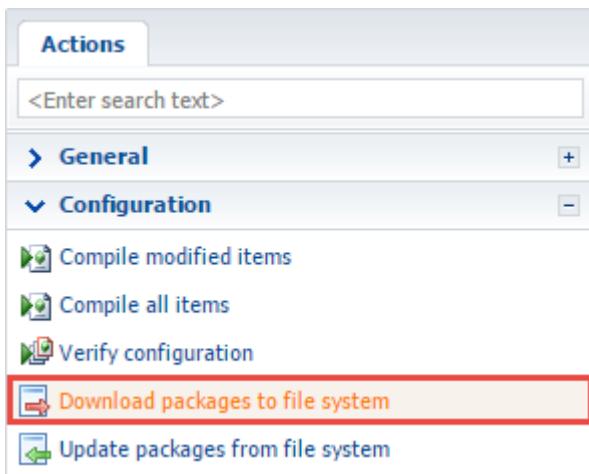
Step 3. Create the required package schemas.

Creating client and server schemas is described in the “[Creating a custom client module schema](#)” and “[Creating the \[Source code\] schema](#)” articles.

Step 4. Export the package to the file system.

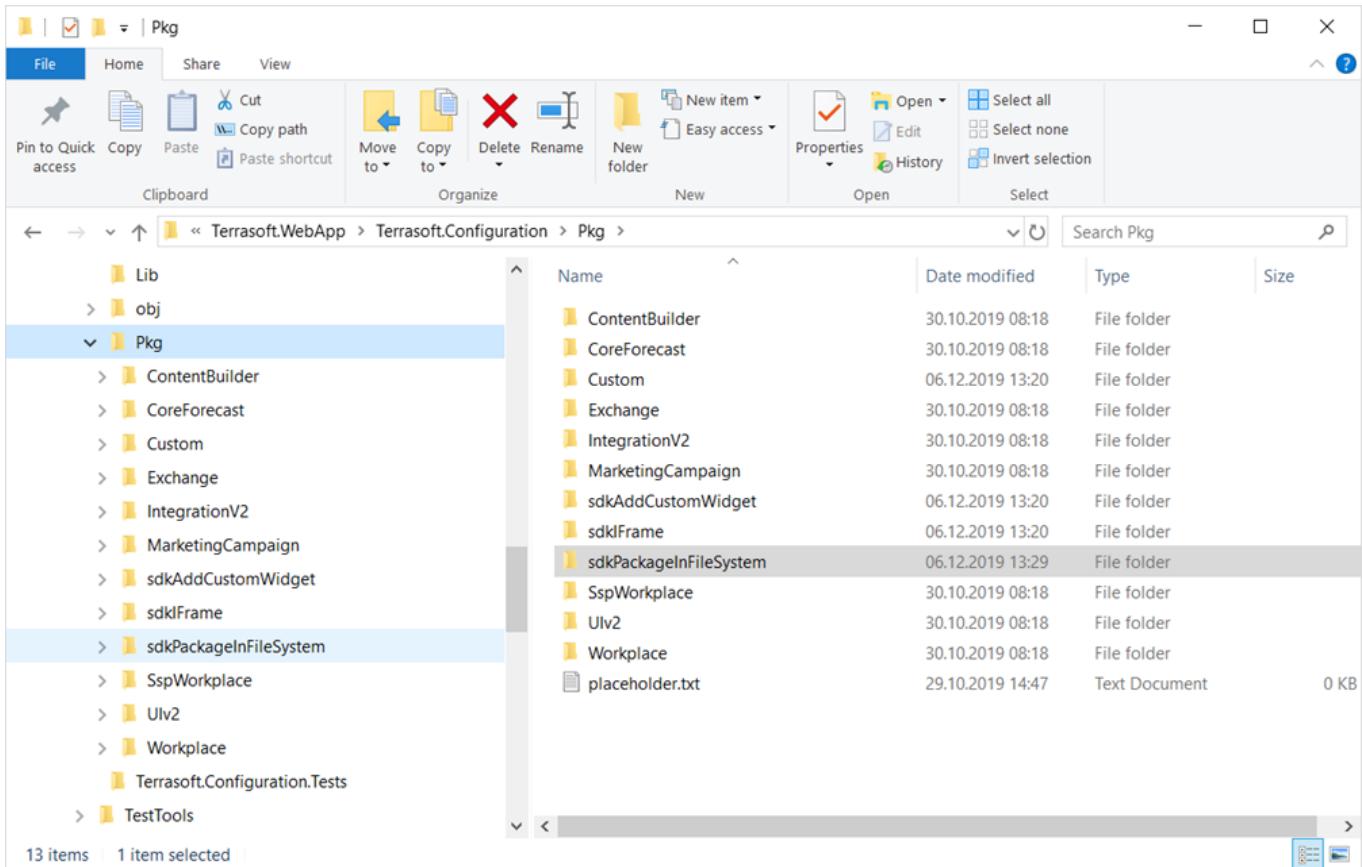
Run the [Download packages to file system] command (Fig. 3).

Fig. 3. – The [Download packages to file system] command



As a result, the package will be exported to the following directory: *[Path to the installed application]\Terrasoft.WebApp\Terrasoft.Configuration\Pkg\sdkPackageInFileSystem* (Fig. 4).

Fig. 4. The package in the file system



Step 5. Add the required source code

You can use any suitable IDE for working with the source code. You can find more information about working with the source code in the Microsoft Visual Studio IDE in the “**Working with the client code**” and “**Working with the server side source code**” articles.

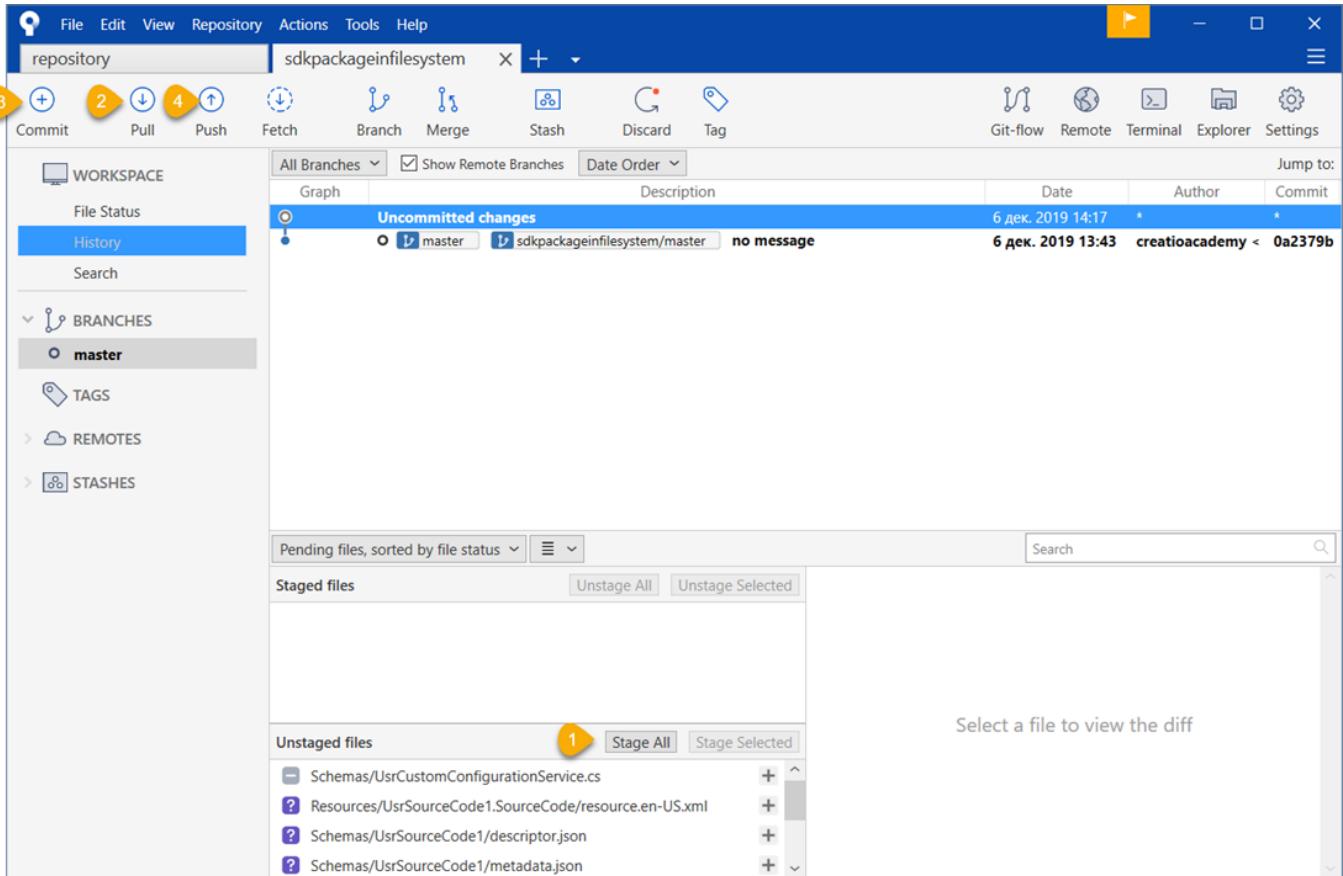
Step 6. Commit the changes to the version control system

The general procedure for committing your changes (fig. 5):

1. Stage all files due for a commit ([Stage All]).

2. Pull the changes made by the other team members from the global repository ([Pull]).
3. Commit the changes to the local repository ([Commit]).
4. Push the committed changes to the global repository ([Push]).

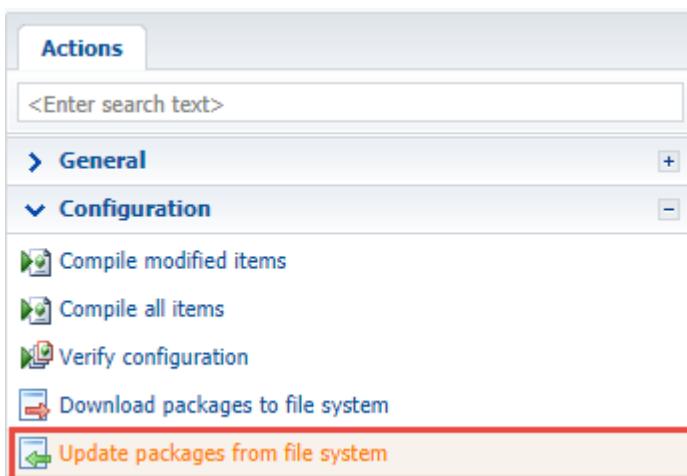
Fig. 5. – Committing your changes in the Git repository



Step 7. Install an updated package in the application

To install a package from the file system, go to the [Configuration] section and run the [Update packages from file system] command (Fig. 6).

Fig. 6. The [Update packages from file system] command



Step 8. Generate source codes

Run the [Generate where it is needed] command in the [Configuration] section to generate source code. For more

information on the [Source Code] group, see “**Built-in IDE. The [Configuration] section.**”

Step 9. Compiling the changes

To compile the changes, run the [Compile modified items] command. For more on configuration commands, see “**Built-in IDE. The [Configuration] section.**”

Creatio informs you about the need to update the database structure and/or install SQL scripts via the [Database update required] and [Needs to be installed in the database] columns in the list of the [Schemas], [SQL scripts] and [Data] tabs of **the [Configuration] section**. Please refer to the [Last error message text] column in case of errors during the database structure update and SQL script installation.

Note that the [Database update required] and [Needs to be installed in the database] columns may not display by default. Right-click the list and select the [Set up columns] command to add these columns to view.

Step 10. Updating the database structure

After the compilation is complete, run the [Update where it is needed] command. For more on database structure commands, see “**Built-in IDE. The [Configuration] section.**”

Step 11. Installing SQL scripts and bound data

If the package contains bound SQL scripts and data, you will need to run additional commands. For more on the SQL script and data commands, see “**Built-in IDE. The [Configuration] section.**”

As a result, the package functions will become available in your Creatio configuration.

Testing tools. NUnit

Beginner Easy Medium **Advanced**

Introduction

Unit-testing (module testing) is a software development process for verifying the operation capacity of the isolated program components (see “[Module testing](#)”). The tests are usually written by developers for every advanced method of the developed class. This allows to quickly reveal the source code recession – errors in the tested program components.

One of the .NET-application Unit-testing frameworks is [NUnit](#) – Unit-testing environment with an open source code. A special adapter has been developed to integrate it with Visual Studio. Such adapter can be installed as a Visual Studio extension or as a project NuGet package with the implemented Unit-tests. Use this [link](#) to access the 3.x version framework documentation.

To create Unit-tests for methods or Creatio custom package class properties:

1. Install NUnit Visual Studio adapter
2. Switch to the **file system development** mode
3. Set up the Unit-test project
4. Create the tests
5. Perform testing

Case description

Add tests for the custom class, implemented in the [Source code] type *UsrNUnitSourceCode* schema of the Creatio application *sdkNUnit* custom package.

Source code

You can access the custom class implementation package at [sdkNUnit](#) repository at Github.

Case implementation algorithm

1. Install NUnit Visual Studio adapter

You can install NUnit Visual Studio adapter either as a Visual Studio extension or as a NuGet package.

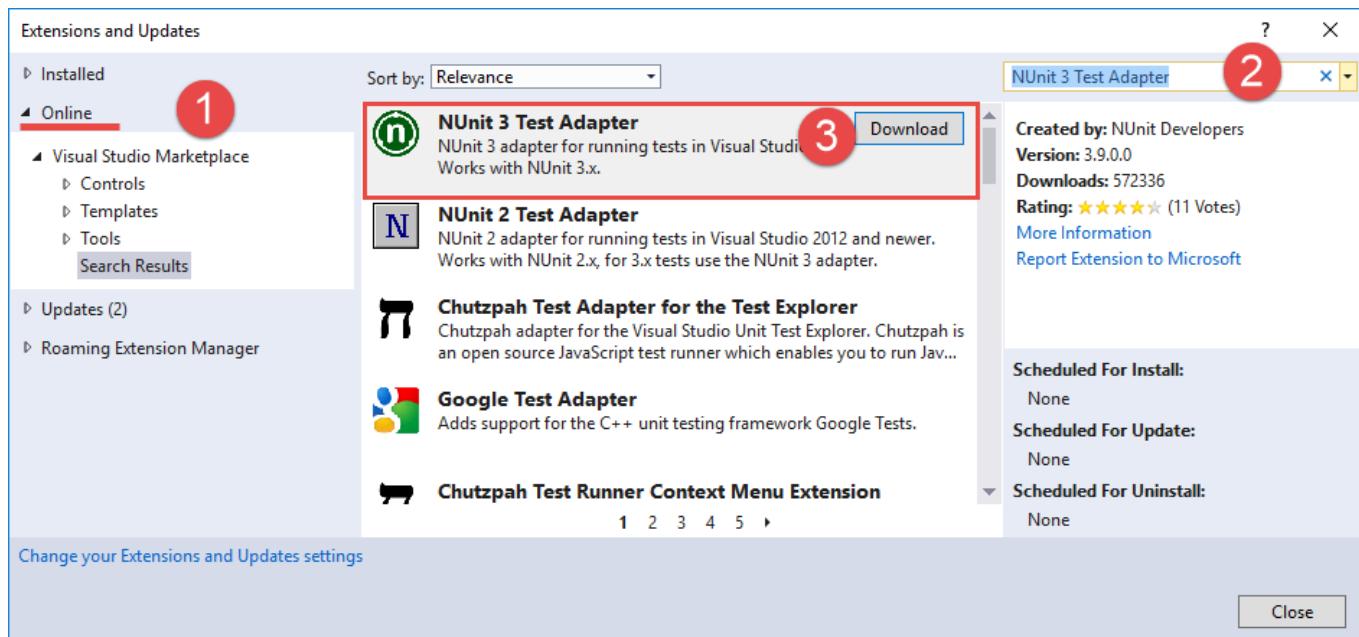
Installing NUnit adapter as a Visual Studio extension

The advantage of installing NUnit adapter as a Visual Studio extension is its availability for any test project since the adapter becomes part of IDE. Another advantage is the automatic extension update. The disadvantage is the necessity to install it for every test project team member.

To install NUnit adapter:

1. [Download extension](#) from Visual Studio Marketplace *.VSIX-file.
2. Double-click the *.VSIX-file and run the installation. Select the needed Visual Studio versions during installation.
As an alternative, you can install NUnit adapter via the Tools > Extensions and Updates menu. Select [Online] filter (Fig. 1. 1) and indicate “NUnit 3 Test Adapter” (2) in the search string. Select NUnit 3 Test Adapter extension in the search results and click [Download]. The extension installation starts automatically.

Fig. 1. Extension search by Visual Studio built-in tools



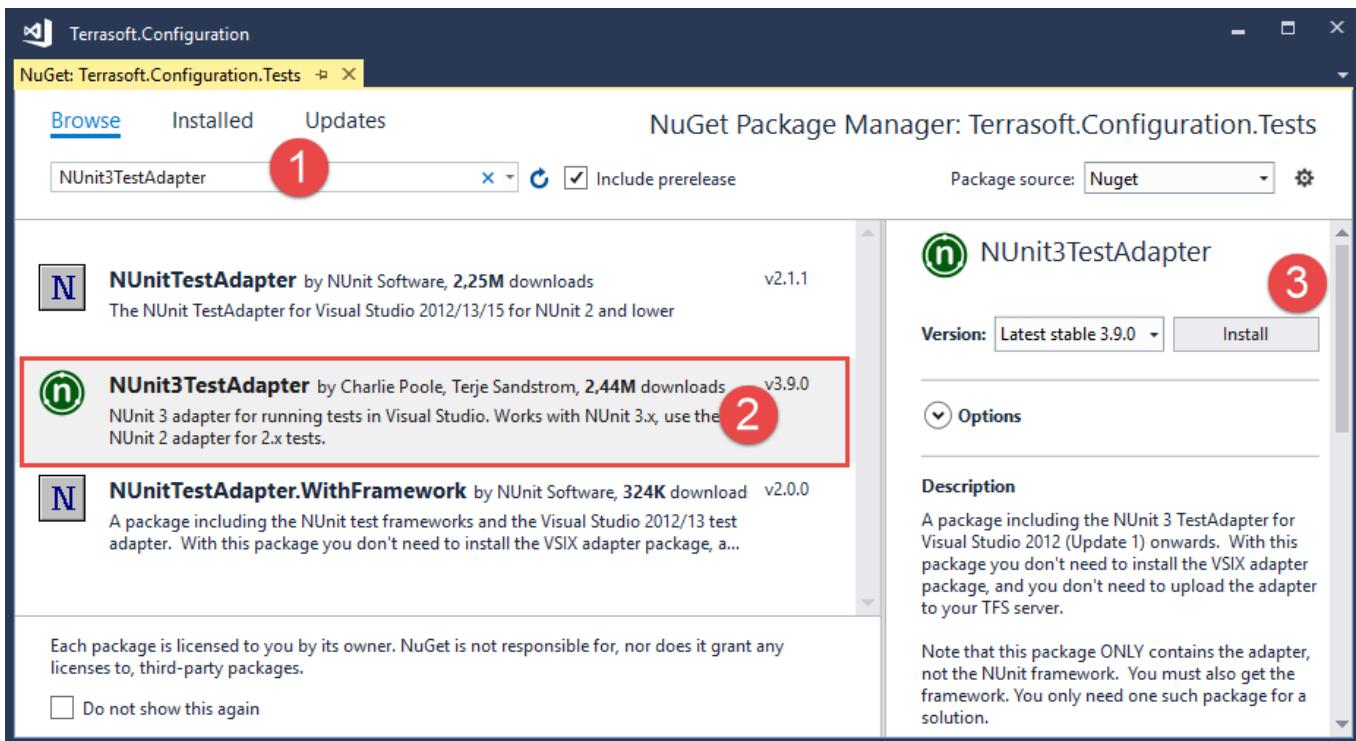
Installing NUnit adapter as a NuGet package

The advantage of NUnit adapter installation as a NuGet-package is that in this case it becomes part of Visual Studio project and is available for access to all developers who use the project. The disadvantage is the necessity to install it for all Unit-test projects.

To install NUnit adapter:

1. Right-click the test project (for instance, *Terrasoft.Configuration.Tests.csproj*) and select the [Manage NuGet Packages...] command.
2. Indicate “NUnit3TestAdapter” (1) in the search string of the opened NuGet package manager tab (Fig.2). Select the package in the search results (2) and install it (3).

Fig. 2. Installing NUnit3TestAdapter package in the NuGet package manager



You can find detailed description of NuGet-package installation into Visual Studio projects in the "[Package Manager UI](#)" Microsoft article.

2. Switch to the file system development mode

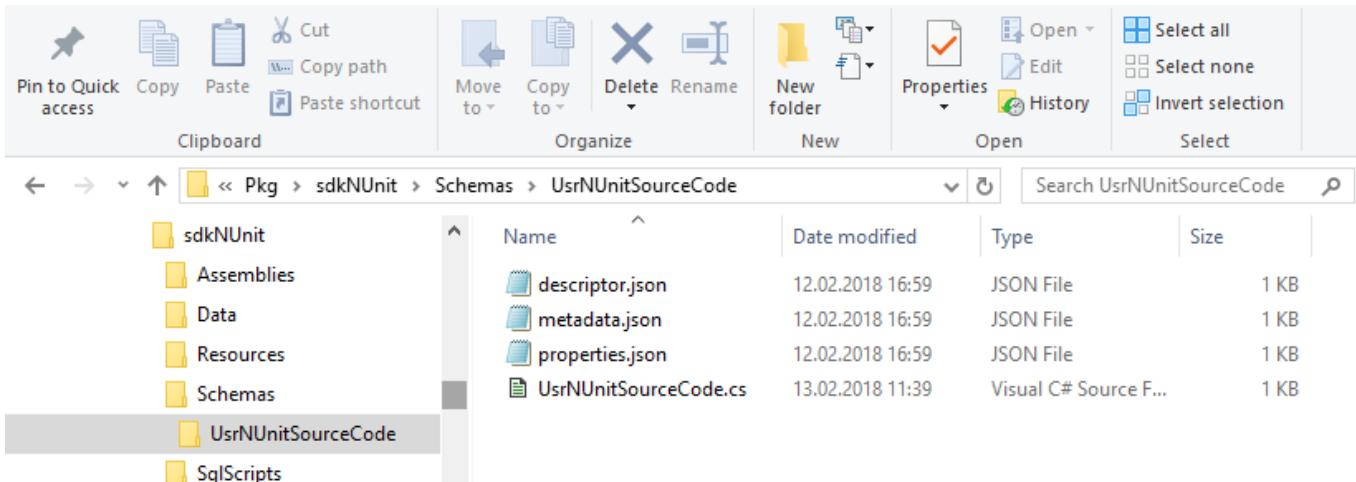
Creating Unit-tests for .NET classes, implemented in Creatio packages is only possible in the file system development mode. You can find more information about the Creatio configuration development in the file system, setting up Visual Studio and the server code operation case in "**Development in the file system**", "**IDE settings for development**" and "**Working with the server side source code**".

The *sdkNUnit* custom package containing [Source code] type *UsrNUnitSourceCode* schema is used in this case. The *UsrNUnitSourceCode* C# class containing methods that require writing tests is implemented in this schema source code.

You can access the custom class implementation package at [sdkNUnit](#) repository at Github.

The *sdkNUnit* custom package has the following view (see Fig.3) after it has been uploaded to the file system:

Fig. 3. The *sdkNUnit* package structure



Class source code for testing:

```

namespace Terrasoft.Configuration
{
    public class UsrNUnitSourceCode
    {
        // String property.
        public string StringToTest
        {
            get
            {
                return "String to test";
            }
        }
        // The method that verifies the equality of the two strings.
        public bool AreStringsEqual(string str1, string str2)
        {
            return str1 == str2;
        }
    }
}

```

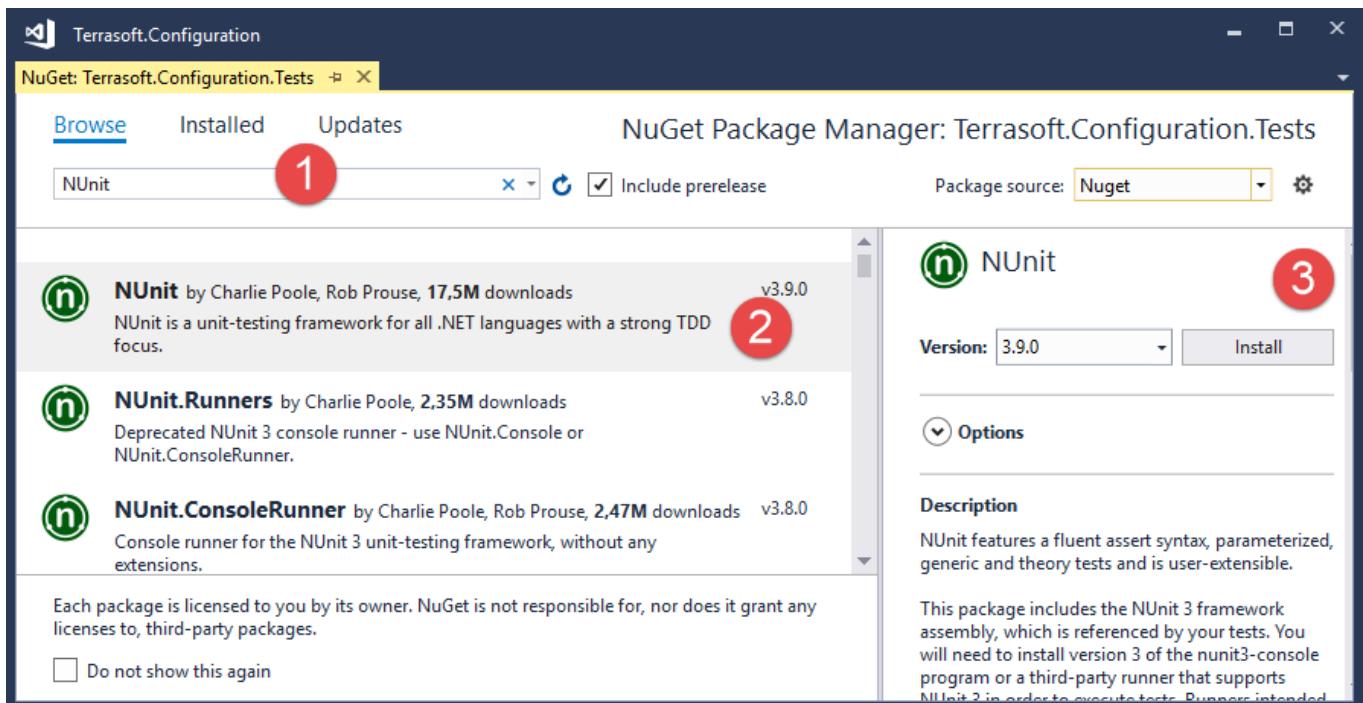
3. Set up the Unit-test project

The *Terrasoft.Configuration.Tests.csproj* pre-configured project is used for creating Unit-tests in this case. It is delivered with the *Terrasoft.Configuration.sln* solution (see “Server code operation in Visual Studio”).

Add the *NUnit* NuGet-package in the project dependency to use *NUnit* framework for creating tests in the *Terrasoft.Configuration.Tests.csproj* project. To do this:

1. Right-click the *Terrasoft.Configuration.Tests* test project in Solution Explorer and select the [Manage NuGet Packages...] command.
2. Indicate “*NUnit*” (1) in the search string of the opened NuGet package manager tab (Fig.4), select the package in the search results (2) and install it (3).

Fig. 4. Installing *NUnit* package in the NuGet package manager



4. Create the tests

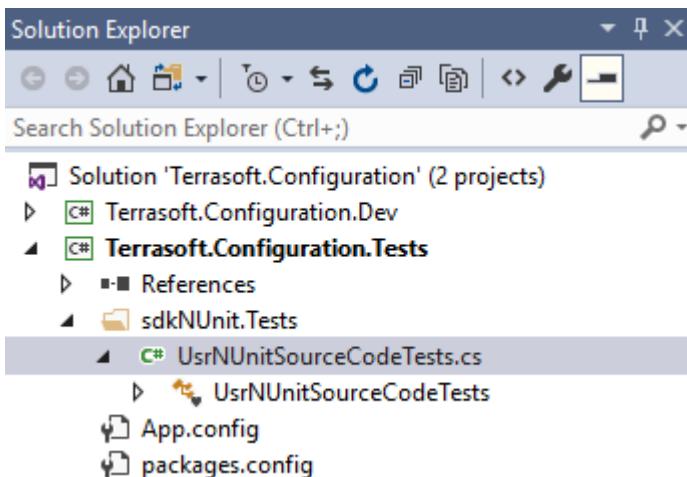
It is common practice that the test-containing class name must have the tested class name with “Tests” word in it. It is also convenient to place tests in catalogs to group them in a project. The catalog name should match the tested

package name and have “.Tests” ending in it.

To create tests for *UsrNUnitSourceCode* class:

1. Create *sdkNUnit.Tests* catalog in the *Terrasoft.Configuration.Tests.csproj* project.
2. Create the new *UsrNUnitSourceCodeTests.class* in the *sdkNUnit.Tests* catalog. This class source code will be stored in the *UsrNUnitSourceCodeTests.cs* file (Fig.5).

Fig. 5. Test project structure



3. Add the implementation test methods to the *UsrNUnitSourceCodeTests* class:

```
using NUnit.Framework;

namespace Terrasoft.Configuration.Tests.sdkNUnitTests
{
    [TestFixture]
    class UsrNUnitSourceCodeTests
    {
        // The tested class instance.
        UsrNUnitSourceCode objToTest = new UsrNUnitSourceCode();
        // Testing string.
        string str = "String to test";

        [Test]
        public void ClassReturnsCorrectStringProperty()
        {
            // Testing the string property value.
            // The value must be populated and match the required value.
            string res = objToTest.StringToTest;
            Assert.That(res, Is.Not.Null.And.EqualTo(str));
        }

        [Test]
        public void StringsMustBeEqual()
        {
            // Testing the value equality of the two strings.
            bool res = objToTest.AreStringsEqual(str, "String to test");
            Assert.That(res, Is.True);
        }

        [Test]
        public void StringsMustBeNotEqual()
        {
            // Testing the value inequality of the two strings.
            // This test will fail since the values are equal.
        }
    }
}
```

```
        bool res = objToTest.AreStringsEqual(str, "String to test");
        Assert.That(res, Is.False);
    }
}
```

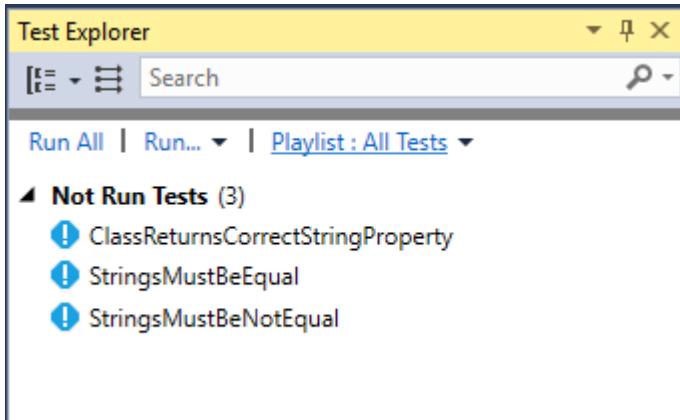
The `UserUnitSourceCodeTests` class is decorated by the `[TestFixture]` attribute, which marks it as a test-containing class. Every method testing a specific functionality must be decorated by the `[Test]` attribute. You can find the description of the NUnit framework attributes in the “[Attributes](#)” NUnit article.

The testing is performed via the `Assert.That()` method that accepts the tested value and such value limiting objects as arguments. You can find more information about the assertions, `Assert.That()` method and limiting model in the [“Assertions”](#) and [“Constraint Model”](#) NUnit articles.

5. Testing

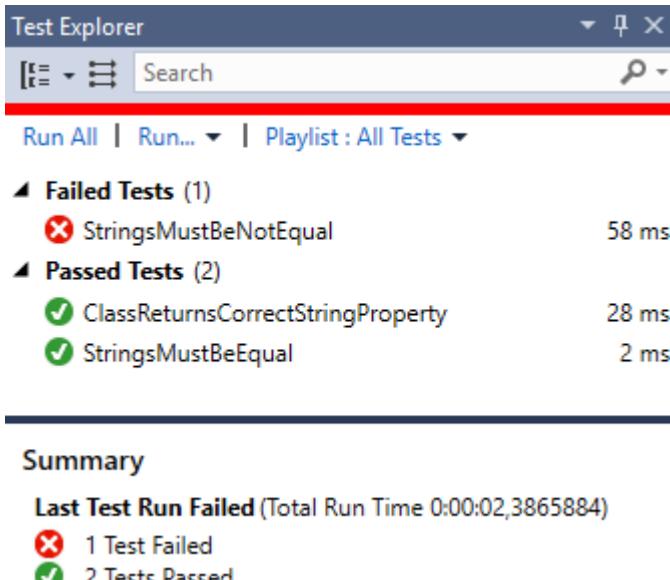
To perform testing, execute the [Test] > [Windows] > [Test Explorer] menu command to open the [Test Explorer] window in Visual Studio (Fig.6).

Fig. 6. [Test Explorer] window



Execute the [Run All] command to run the tests. The successfully passed tests will be moved to the [Passed Test] group, the failed tests will be moved to the [Failed Test] group (Fig.7).

Fig. 7. Passed and Failed tests



You can find more information about the [Test Explorer] window functionality in the “[Run unit tests with Test Explorer](#)” Visual Studio article.

Logging tools

Contents

- **Logging in Creatio. NLog**
- **Logging in Creatio. Log4net**

Logging in Creatio. NLog

Beginner

Easy

Medium

Advanced

Introduction

To test if the new functionality works as expected, we recommend enabling logging. Make sure you disable logging after testing and debugging are complete to prevent performance drop. In Creatio, logging is possible for all primary operations. We recommend using the [NLog](#) library, which is a free .NET logging library with ample routing features and high-quality log management. NLog is suitable for any application regardless of its size or complexity. The library can process diagnostic messages generated in any .NET programming language, enrich them with contextual data, format the messages according to the user preferences and send them to one or multiple message receivers, such as a file or database.

NLog logs events for the application loader and for the *Default* configuration independently. A developer can configure logging in the ..\Terrasoft.WebApp directory in the following configuration files:

- *nlog.config*.
- *nlog.targets.config*.
- *nlog.cloud.config* (for cloud applications).
- *nlog.cloud.settings.config* (for cloud applications).

NLog documentation is available on the [GitHub website](#).

Logging for on-site applications

Storing log data

The location of log files depends on the values of Windows system variables.

By default, the loader log files are stored in the following path:

```
[TEMP]\Creatio\Site_{SiteId}\{ApplicationName}\Log\{DateTime.Today}
```

For example:

```
C:\Windows\Temp\Creatio\Site_1\creatio7121\Log\2018_05_22
```

The *Default* configuration log files are stored in the following path:

```
[TEMP]\Creatio\Site_{SiteId}\{ApplicationName}\[ConfigurationNumber]\Log\{DateTime.Today}
```

For example:

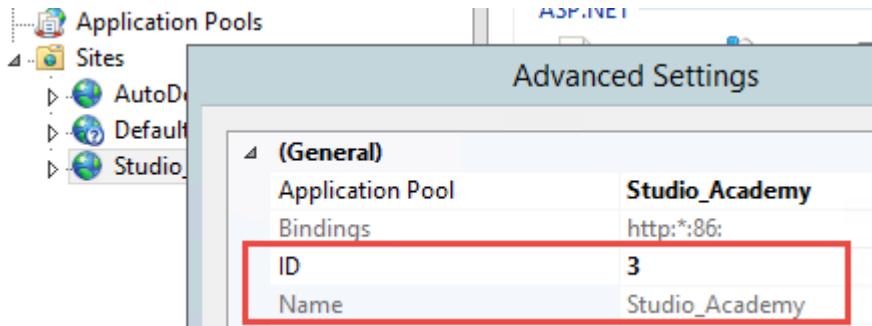
```
C:\Windows\Temp\Creatio\Site_1\creatio7121\0\Log\2018_05_22
```

In the examples above, the following variables are enclosed in square brackets:

- [TEMP] – base folder. By default, IIS uses the *C:\Windows\Temp* folder. Visual Studio (IIS Express) uses the *C:\Users\{User name}\AppData\Local\Temp* folder.
- [{SiteId}] – site number. For IIS, a developer can find the number in the advanced settings of the site (Fig. 1). For Visual Studio, the number is 2.
- [{ApplicationName}] – application name (Fig. 1).
- [ConfigurationNumber] – configuration number. The number for the *Default* configuration usually is 0.

- [{DateTime.Today}] — logging date.

Fig. 1. – Site advanced settings view in IIS



Log settings

You can set up logging in two ways:

- In the configuration file.
- In the *LoggingConfiguration* object.

Setting up the logging in the configuration file

1. Define the path to the logging configuration file

You can define the path to the *nlog.config* file in the ..\Terrasoft.WebApp\Web.config. file.

```
<common>
    <logging>
        <factoryAdapter type="Common.Logging.NLog.NLogLoggerFactoryAdapter,
Common.Logging.NLog45">
            <arg key="configType" value="FILE" />
            <arg key="configFile" value="~/nlog.config" />
        </factoryAdapter>
    </logging>
</common>
```

2 Define the log receivers

The log receivers display, store and transfer the log messages to other receivers. Certain receivers can receive and process the messages, while others can buffer or reroute them to another receiver.

The receivers are defined in the *nlog.targets.config* file using the following attributes:

- *name* — receiver name.
- *xsi:type* — receiver type. This setting can take one of these values: “File”, “Database”, “Mail”.
- *fileName* — the path to the logging file.
- *layout* — the layout of the logging file.

[NLog](#) documentation describes log receivers in detail.

Example: Setting up a log receiver

To do this, you need to define the properties of the log receiver in the *<target>* XML element in the ..\Terrasoft.WebApp\iLog.targets.config file.

```
<target name="universalAppender" xsi:type="File"
    layout="${DefaultLayout}"
    fileName="${LogDir}${LogDay}${logger:shortName=True}.log" />
```

3. Define the log rules

The log rules are defined in the *nlog.config* file with the following attributes:

- *name* — log name.
- *minlevel* — minimum log level.
- *maxlevel* — maximum log level.
- *level* — filter log events of a specific log level.
- *levels* — filter log events of several specific log levels (comma-separated list).
- *writeTo* — the name of the log receiver.
- *final* — if a log rule has this attribute, the subsequent rules are not processed.
- *enabled* — turn a log rule off (by specifying false) without deleting it.
- *ruleName* — the log rule identifier.

Log rule attributes are processed in the following order:

1. *level*.
2. *levels*.
3. *minlevel* and *maxlevel* (both have the same priority).

In the *minLevel = "Warn"* *level = "Info"* configuration, only the *Info* log level will be used. The *level* attribute has a higher priority than *minLevel*.

The log levels are defined in the *nlog.config* file. By default, the log level for all Creatio components is set to allow for the best performance of the application. Existing log levels in order of increasing priority:

- *Trace* — high-volume logging of all events, useful for debugging and tracing (the initial and final methods are displayed).
- *Debug* — logging all events, useful for debugging
- *Info* — logging Info messages normally triggered for regular application activity.
- *Warn* — logging warnings (when the application still functions).
- *Error* — logging errors (when the application may throw exceptions or crash).
- *Fatal* — logging errors that inevitably crash the application.
- *Off* — logging is disabled, not used for records.

An example log rule that enables logging events to a database:

```
<logger name="IncidentRegistration" writeTo="AdoNetBufferedAppender" minlevel="Trace"
final="true" />
```

Setting up the logging in the LoggingConfiguration object

To configure logging:

1. Create a *LoggingConfiguration* object to describe your configuration.
2. Create one or multiple log receivers.
3. Set up the properties of the log receivers.
4. Define the log rules using *LoggingRule* objects and add them to the *LoggingRules* configuration objects.
5. Activate the configuration by pointing to the newly created configuration object in *LogManager.Configuration*.

Examples of using C# for setting up the logging are available on the [GitHub website](#).

Logging for cloud applications

To set up logging in a cloud application, you contact Creatio technical support.

Logging in Creatio. Log4net

Beginner

Easy

Medium

Advanced

Introduction

Logging is useful for localization of application troubles. Creatio enables logging for all main operations.

The [Log4net](#) solution is used for logging. This tool enables to perform logging of parameters from different components of the application into separate log files.

Logging is performed separately for the application loader and for the *Default* configuration. To set up logging, modify the ..\Terrasoft.WebApp\log4net.config configuration file.

Storing log data

Location of the log files depends on the value of Windows system variables.

By default the loader log files are located by following path:

```
[TEMP]\Creatio\Site_{[SiteId]}\[{ApplicationName}]\Log\{DateTime.Today}
```

Example:

```
C:\Windows\Temp\Creatio\Site_1\Creatio\Log\2018_05_22
```

Files with the *Default* configuration logs are located by following path:

```
[TEMP]\Creatio\Site_{[SiteId]}\[{ApplicationName}]\[ConfigurationNumber]\Log\{DateTime.Today}
```

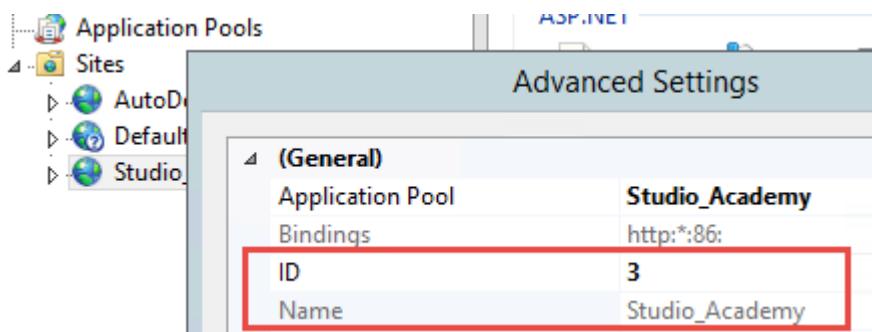
Example:

```
C:\Windows\Temp\Creatio\Site_1\Creatio\0\Log\2018_05_22
```

Variables specified in the square brackets:

- [TEMP] – the base folder. By default the *C:\Windows\Temp* folder is used by IIS and the *C:\Users\{User name}\AppData\Local\Temp* folder used by Visual Studio (IIS Express).
- [{SiteId}] – site number. For the IIS, the number is specified in the site advanced settings (Fig. 1). For the Visual Studio the number is 2.
- [{ApplicationName}] – application name (Fig. 1).
- [ConfigurationNumber] – configuration number. The number for the *Default*, configuration usually is 0.
- [{DateTime.Today}] – logging date.

Fig. 1. Advanced setting of the IIS site



Changing the logging level

By default the logging level for all Creatio components is set to provide maximal performance for the application. Possible levels of logging in order of increasing priority:

- ALL – logging of all events. Significantly reduces application performance.
- DEBUG – logging all events at debugging.
- INFO – logging of errors, warnings and messages.
- WARN – logging of errors and warnings.
- ERROR – logging of errors.
- FATAL – logging only errors that lead to the termination of the component being logged.

- OFF – logging disabled.

Example 1. Set the maximum level of logging for all components

To do this, specify the ALL level in the <root> XML element of the .\Terrasoft.WebApp\log4net.config file.

```
<root>
    <level value="ALL" />
    <appender-ref ref="commonAppender" />
</root>
```

Example 2. Set logging of errors when working with SVN

To do this, specify the ERROR level in the <logger name="Svn"> XML element of the .\Terrasoft.WebApp\log4net.config file.

```
<logger name="Svn" >
    <level value="ERROR" />
    <appender-ref ref="SvnAppender" />
</logger>
```

Server code debugging

Beginner

Easy

Medium

Advanced

Introduction

During the development process on the Creatio platform, developers often need to create the source code for the server schemas of the "source code" type. These may be, for example, existing base schemas, custom configuration classes, web services or business process scripts written in C#. Debug such code is easiest with integrated debugging features of the development environment, for example, Visual Studio. The Visual Studio debugger enables developers to freeze the execution of program methods, check variable values, modify them and monitor other activities performed by the program code.

To begin debugging an application, you need to perform a number of steps:

1. Export the Creatio configuration source code to the local directory files
2. Create a new Visual Studio project for debugging
3. Add the exported files with the source code to the Visual Studio project
4. From the project, attach to the working process of the IIS server and start debugging.

Debugging the code using the method described in this article is only possible for **applications deployed on-site**.

Debugging the code using the method described in this article is only possible if the development in the file system mode is turned off (see: "**Development in the file system**").

Enable the [Suppress JIT Optimization] checkbox (the [Options] menu, the [Debugging] > [General] tabs) to be able to get the values of variables during the debugging. More information about optimized and unoptimized code during debugging can be found in the "[JIT Optimization and Debugging](#)" Visual Studio guide.

1. Exporting the configuration source code

To do this, perform the application setup.

In the Web.config file located in the root of the application ("external" Web.config), set the "true" value for the *debug* attribute of the *compilation* element.

```
<compilation debug="true" targetFramework="4.5" />
```

Save the schema to apply changes.

In the Web.config file located in the Terrasoft.WebApp directory of the application ("internal" Web.config), specify the values for the following items:

- To configure *IncludeDebugInformation*, specify the "true" value.

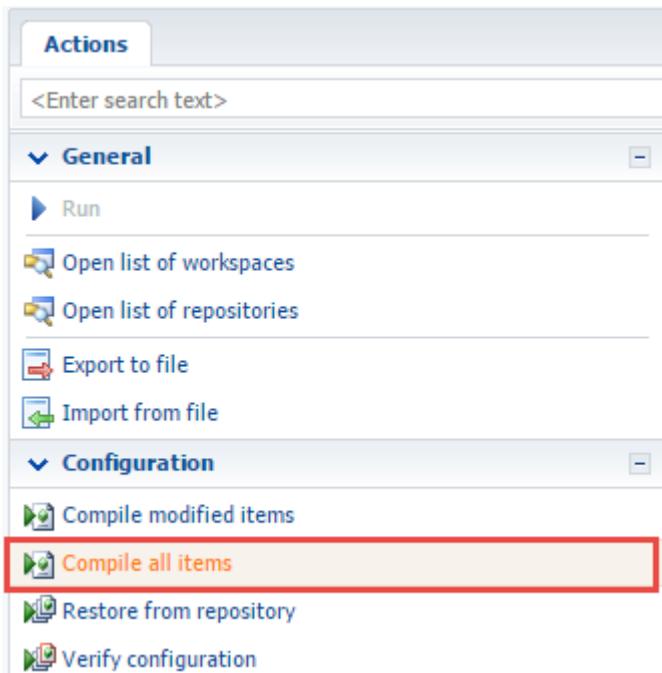
- To configure *CompilerSourcesTempFolderPath*, specify the path to the directory where the source files will be exported.
- To configure *ExtractAllCompilerSources*, set the value to “*true*” if you want to export all schemas when performing the [Compile modified items] action in the **[Configuration] section**. To export only the modified schemas, set the value to “*false*” (the default value).

```
<add key="IncludeDebugInformation" value="true" />
<add key="CompilerSourcesTempFolderPath" value="Path_to_local_catalog" />
<add key="ExtractAllCompilerSources" value="false" />
```

Save the schema to apply changes.

To export the files with server schema source code, perform the [Compile all items] (Fig. 1) action in the [Configuration] section.

Fig. 1 [Compile all items] action



At the time of compilation, the source code files for the application's configuration schemas, as well as configuration libraries, their modules and debug files (*.pdb) will be exported to the folder specified in the *CompilerSourcesTempFolderPath* configuration of the "internal" Web.config. The schema source code will be exported again every time you compile the application.

When compiling, the source code files of the schemas of the work space under which compilation was started will be exported. The files of the downloaded source codes of configuration schemas are named in a certain format: [Name of the schema in the configuration].[Package name]_[Schema type].cs.

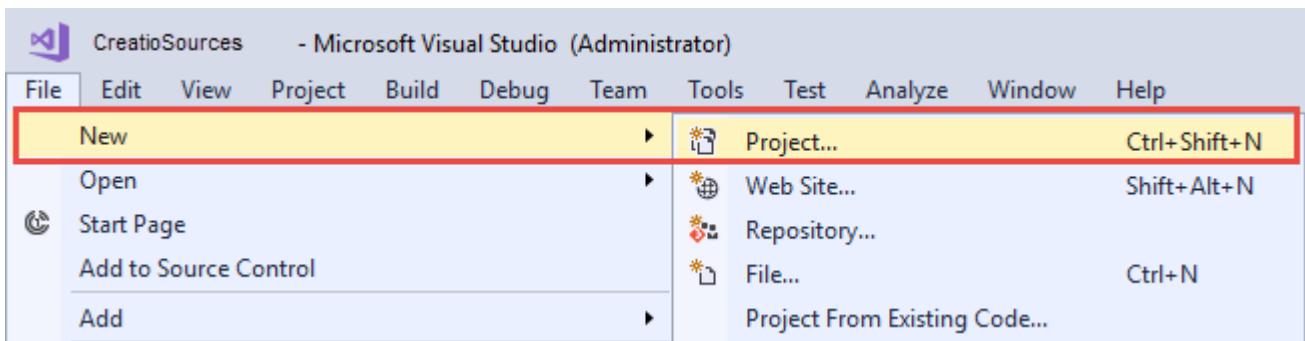
For example: Contact.Base_Entity.cs, ContractReport.Base_Report.cs.

2. Creating a Visual Studio project for debugging

Creating a Visual Studio project is unnecessary to debug the source code – opening the necessary files in Visual Studio is sufficient. However, if debugging is performed frequently, or you need to work with a large number of files at the same time, creating a project will make it easier.

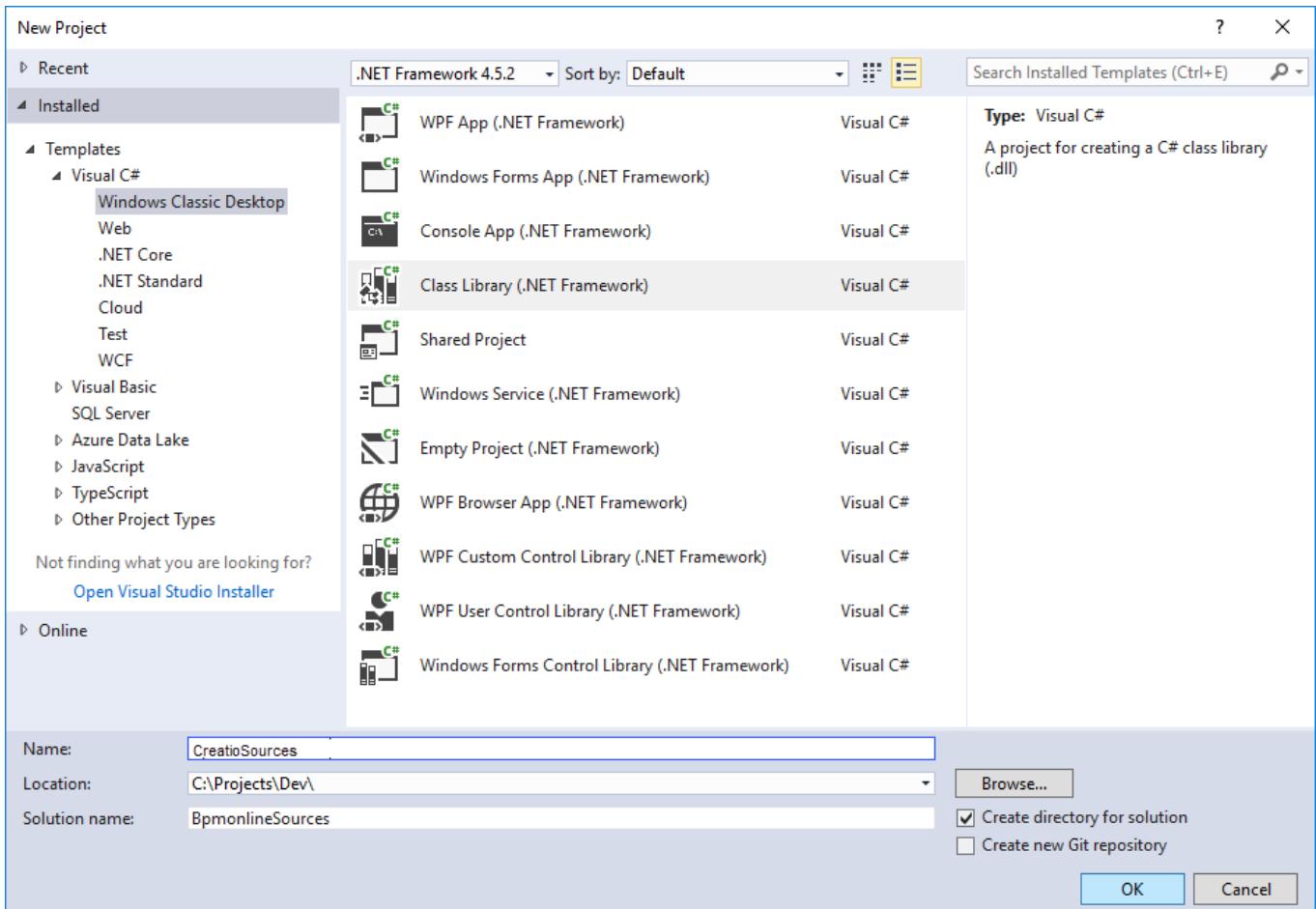
To create a project for debugging an application in Visual Studio, execute the *File > New > Project* menu command (Fig. 2).

Fig. 2 Creating a new project in Visual Studio



In the properties window of the created project, select the [Class Library (.NET Framework)] project type (class library for the classic Windows application), and specify the name and location of the project (Fig. 3).

Fig. 3 Visual Studio project properties

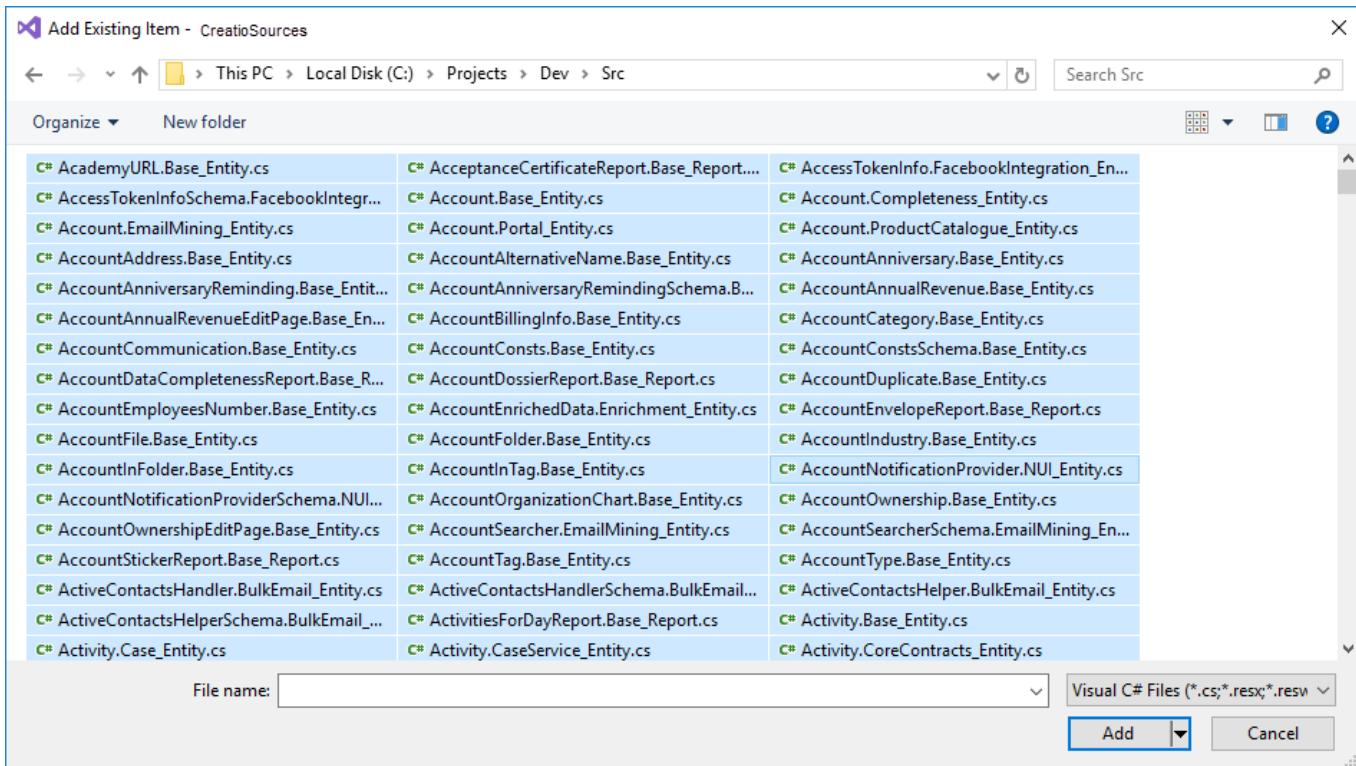


After creating the project, you need to remove an extra file from it (by default, the file Class1.cs is added to the new project) and save the project.

3. Adding the exported files with the source code to the Visual Studio project

To do this, select *Add > Existing Item* from the project's context menu in the solution explorer. In the dialog box that appears, you must go to the directory with the downloaded files with the source code and select all files (Fig. 4).

Fig. 4 Adding files to a project



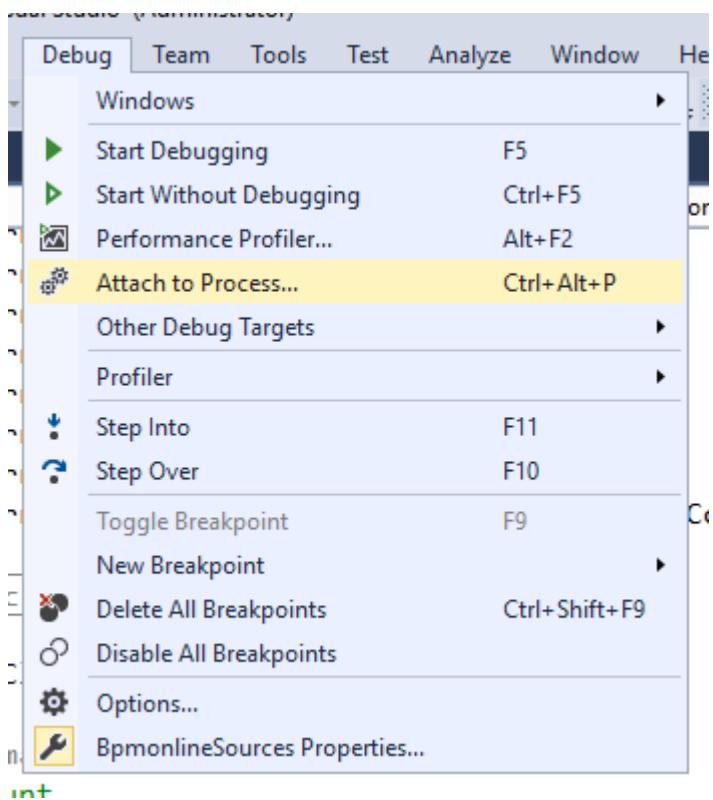
Add only the files needed for debugging to the Visual Studio project. However, the transition between methods during debugging will be limited only by the methods of classes implemented in the files added to the project.

Save the project after adding the files.

4. Attaching to the IIS process for debugging

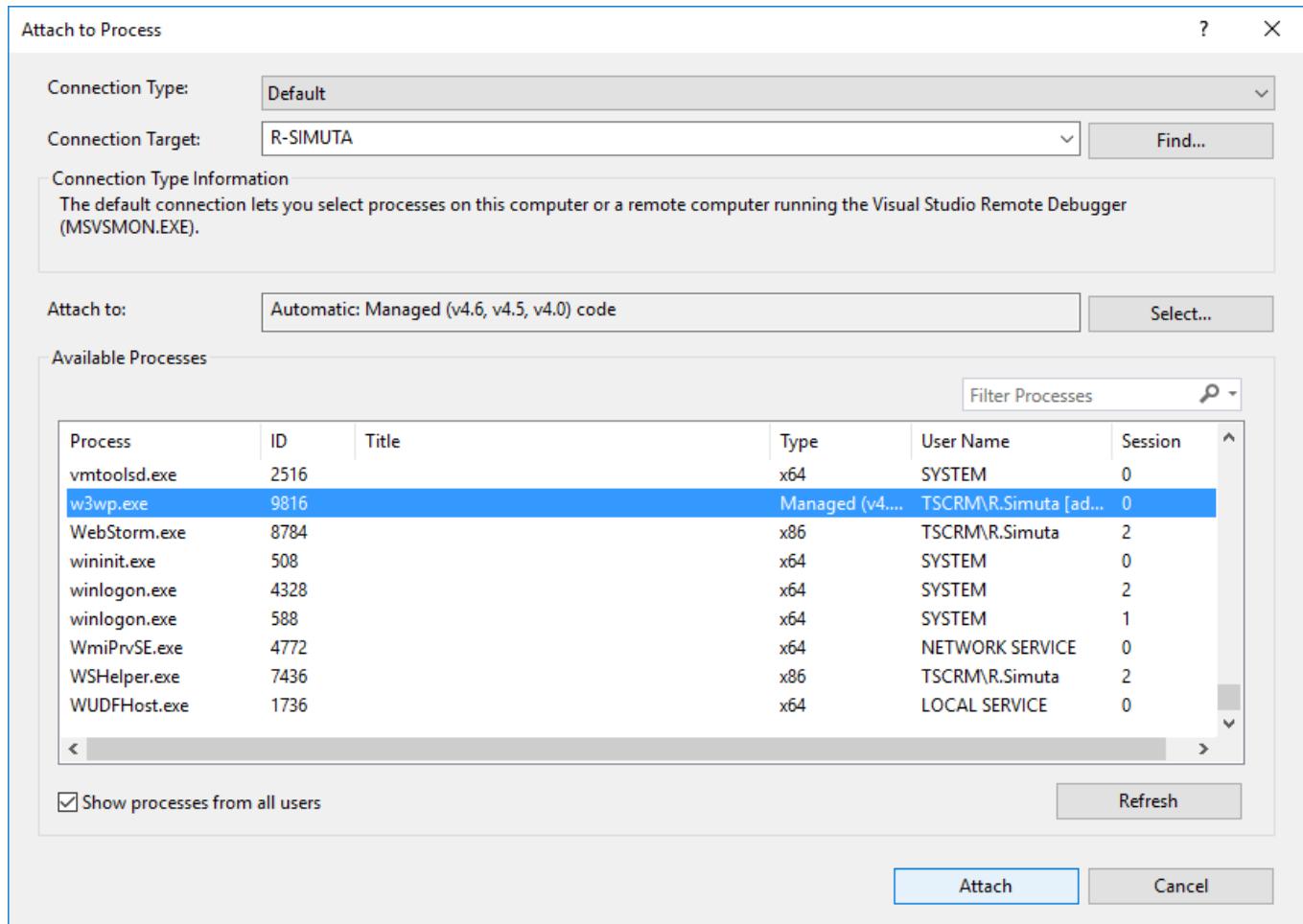
To begin debugging, attach to the IIS server process, where the application runs. To do this, select the *Debug > Attach to process* command in the Visual Studio menu (Fig. 5).

Fig. 5 Attaching to a process



In the opened window, select the working IIS process in the list of processes, where the application pool is running (Fig. 6).

Fig. 6 Attaching to an IIS process

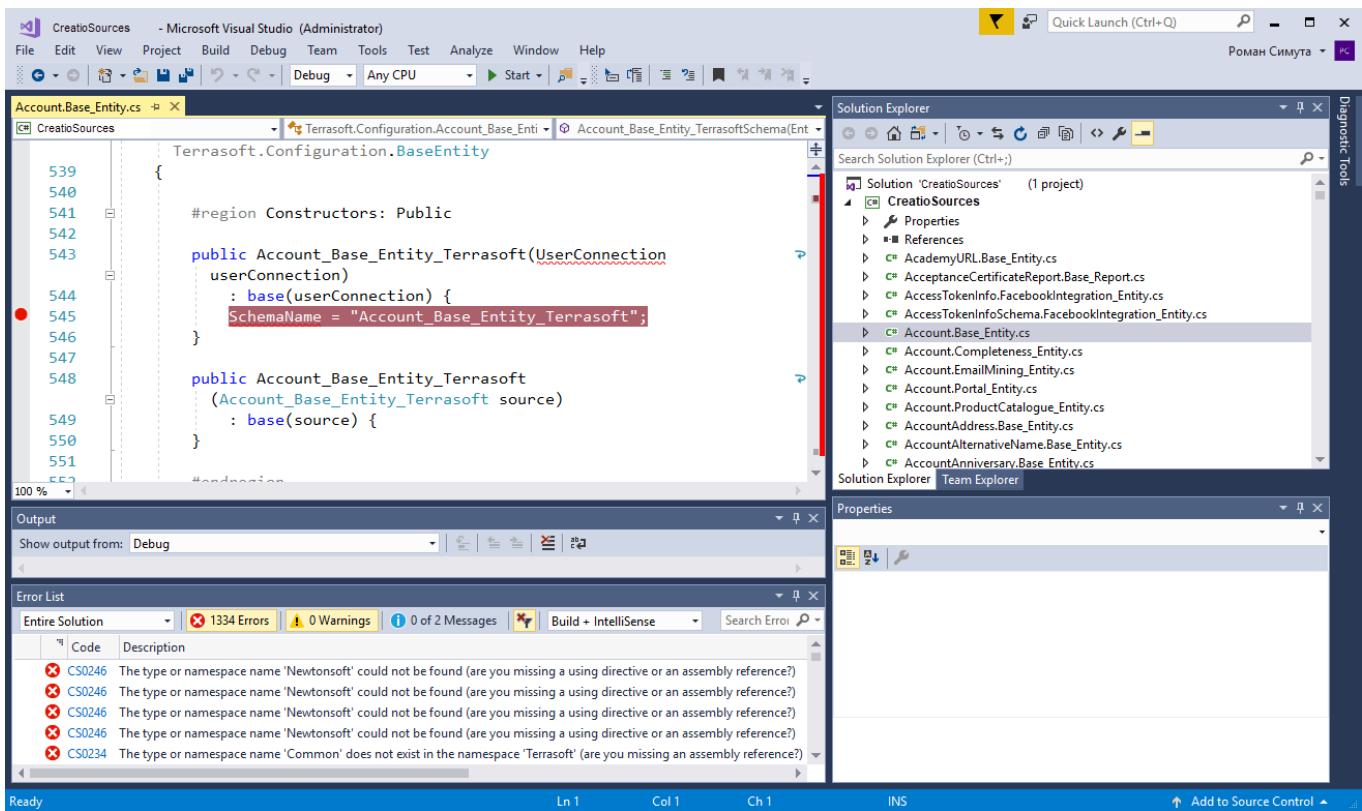


The name of the working process can be different, depending on the configuration of the IIS server being used. With a regular IIS, the process is w3wp.exe, but with IIS Express, the process name is iexpress.exe.

By default, the IIS working process is run under the account whose name matches the name of the application pool. To display processes of all users, set [Show processes from all users] checkbox (Fig. 6).

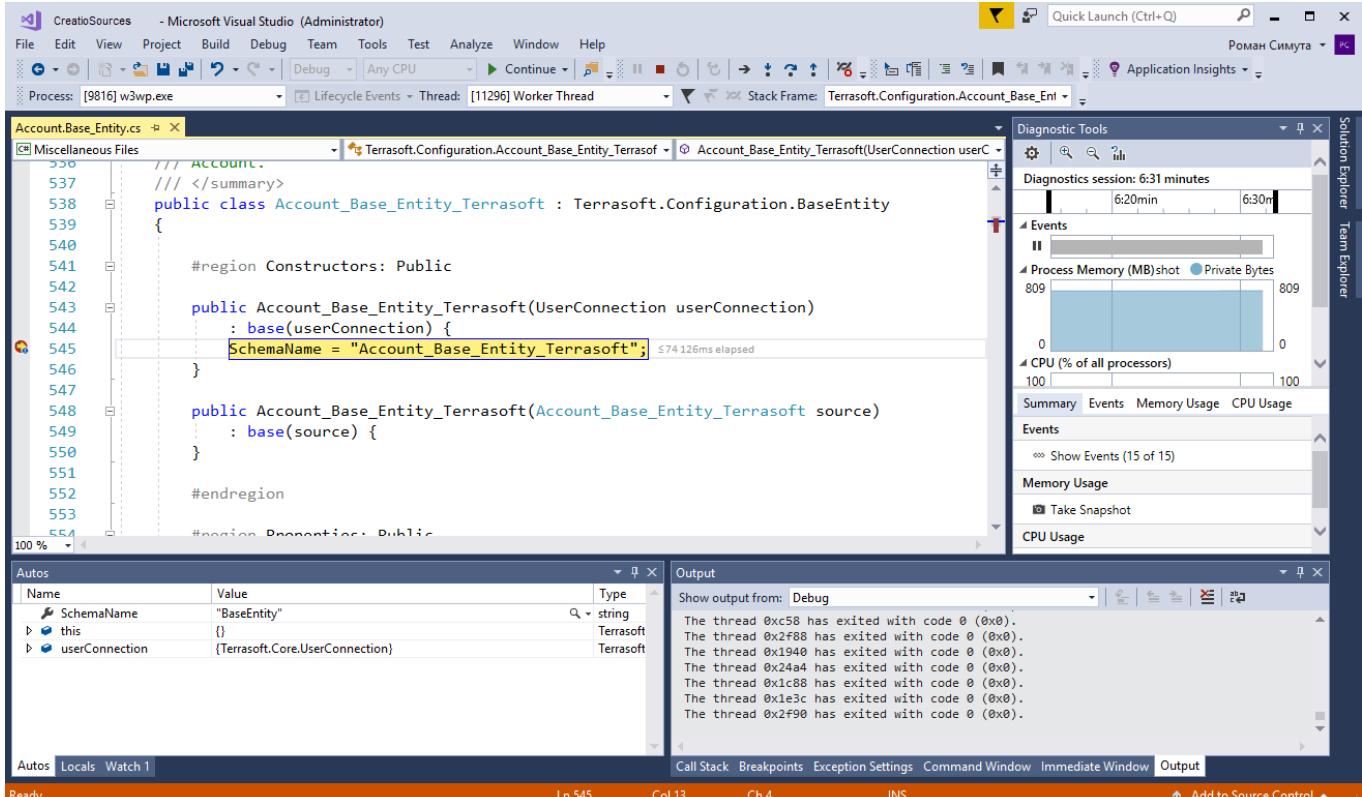
After you attach to the IIS process, you can start debugging. To do this, open the file with the desired source code and set a breakpoint (Fig. 7).

Fig. 7 Breakpoint in the constructor of the [Account] object



As soon as the method with the breakpoint is used, the program will be stopped and you can view the current state of the variables (see Fig. 8).

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

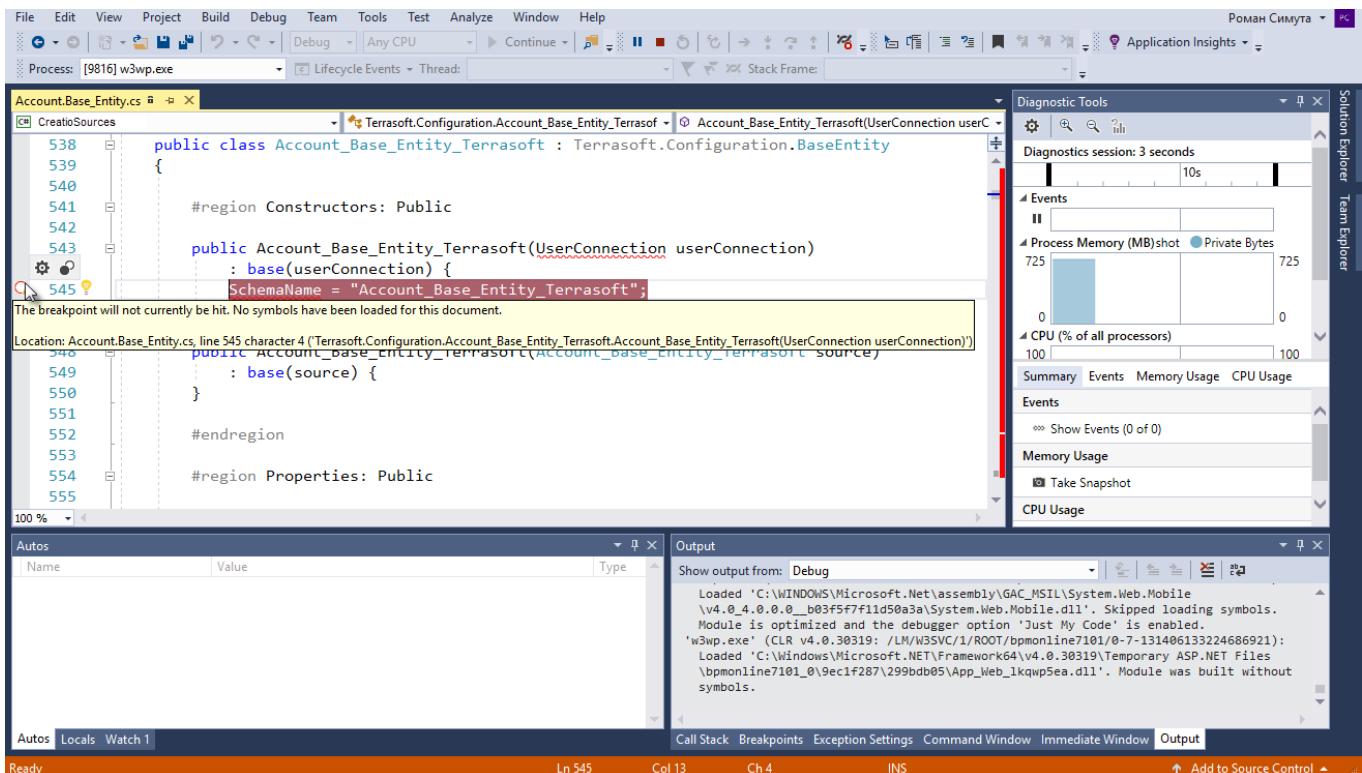


Possible debugging issues

After attaching to the IIS process, it is possible that the breakpoint symbol is displayed as a white circle bounded by

a red circle. A breakpoint is inactive and the application execution will not be interrupted because of it. When you hover the cursor on the symbol of the inactive breakpoint, a hint will appear and notify you of the problem (Fig. 9).

Fig. 9 Inactive breakpoint Characters not loaded



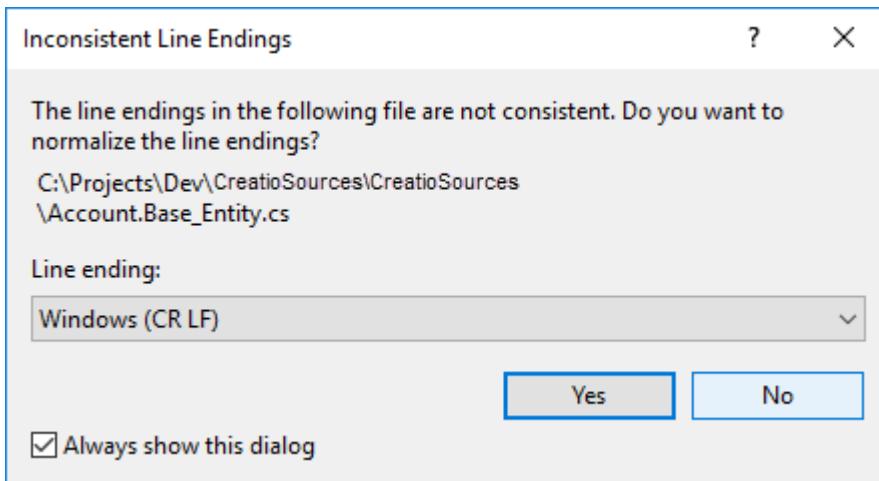
If the hint contains a message that the symbol information was not loaded (Fig. 8), it is necessary to do the following:

1. Finish debugging (*Debug > Stop Debugging*).
2. Close the source code file you are debugging.
3. Perform the [Compile all items] action in the [Configuration] section of the application (Fig. 1).
4. While compiling and re-exporting source files, attach to the IIS process again.
5. After the compilation is done, re-open the source code file you are debugging.

In some cases, it may be helpful to re-compile without detaching and attaching to IIS.

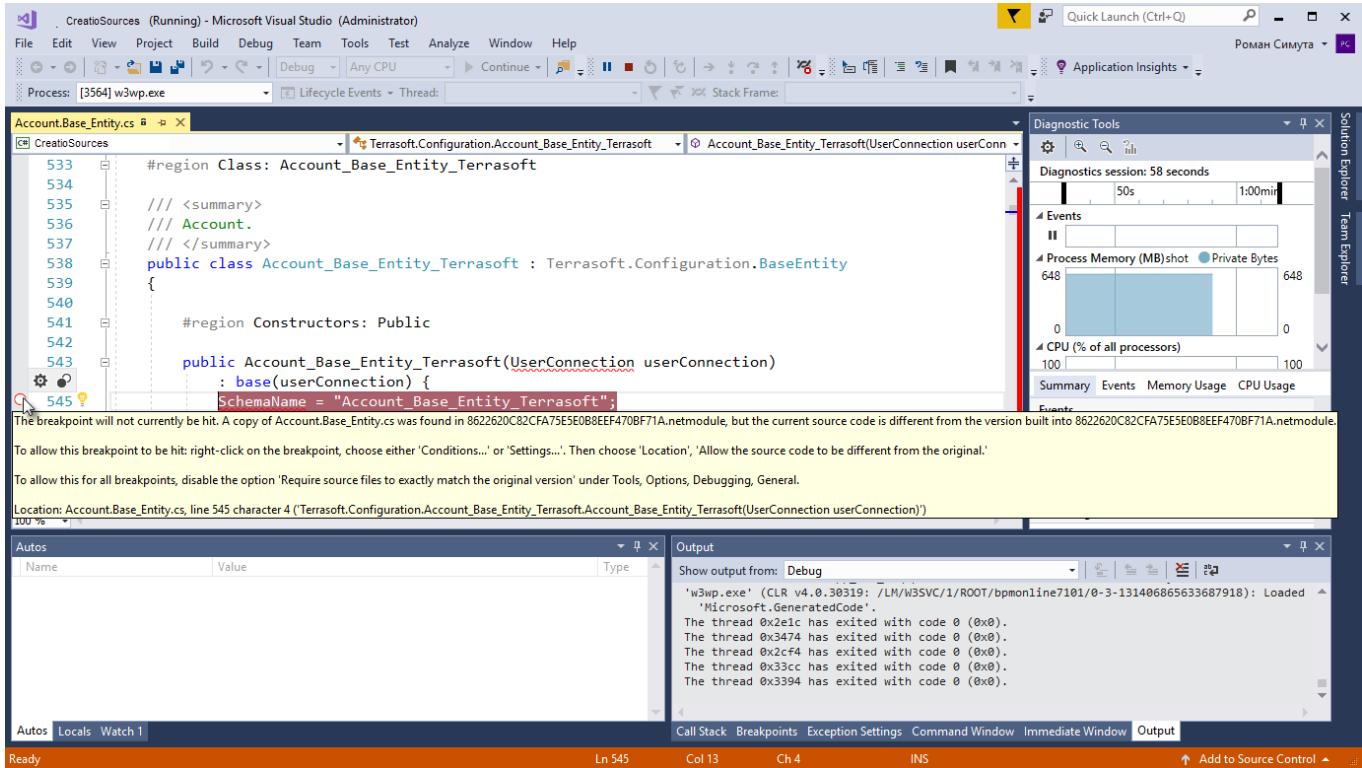
After the file with the source code is reopened, a message about non-uniform end-of-line characters may appear (Fig. 10).

Fig. 10 Message about non-uniform end-of-line characters



Press the [No] button If you accept the normalization of the characters (the [Yes] button), then the breakpoint may become inactive again. The cause of the problem is displayed in the tooltip - file version mismatch (Fig. 10). The options for solving the problem are also displayed in the tooltip.

Fig. 11 Inactive breakpoint. Version mismatch



Delivery tools

Contents

- **WorkspaceConsole utility**

WorkspaceConsole utility

Contents

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

WorkspaceConsole settings

Beginner

Easy

Medium

Advanced

Introduction

The WorkspaceConsole utility is designed for working with Creatio packages. Use the utility to:

- Export packages from development environments and migrate them to test environments or production

environments (the packages are saved as archives).

- Install new packages when upgrading or migrating from development environments.
- Import and export schema resources and data for localization.
- Work with configuration schemas.

The utility executable file (*Terrasoft.Tools.WorkspaceConsole.exe*) is located in the Creatio application directory:

```
[Path to the catalog with installed  
application]\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\
```

The build version of the utility must match the build version of the application.

When updating the application, the build version of WorkspaceConsole must correspond to the target build version. For example, if the current version of the Creatio build is 7.11.1.1794, and you need to update the packages to 7.11.2.1658, then you must use WorkspaceConsole version 7.11.2.1658.

Setting up the utility

WorkspaceConsole works directly with the Creatio application database. Thus, it is necessary to specify database connection string in the configuration file (*Terrasoft.Tools.WorkspaceConsole.exe.config*) for the utility to work properly.

Recommended sequence:

1. Check the *connectionStringName* attribute of the *<db>* element of the configuration file for the connection string name. In the current example, the *connectionStringName* attribute value is "db".

```
<terrashort>  
...  
  <db>  
    <general connectionStringName="db"  
securityEngineType="Terrasoft.DB.MSSql.MSSqlSecurityEngine, Terrasoft.DB.MSSql" ...  
/>  
  </db>  
...  
</terrashort>
```

2. Find the connection string in the *<connectionStrings>* element of the configuration file. The *name* attribute will match the *connectionStringName* attribute of the *<db>* element. In the current example, the *name* attribute value is "db".

```
<connectionStrings>  
...  
  <add name="db" ... />  
...  
</connectionStrings>
```

By default, the configuration file contains two connection strings. The "db" string is used for connecting to the MS SQL Server database. The "dbOracle" string is used for connecting to the Oracle database.

3. Modify the value of the *connectionString* attribute, so that it matches the value used in the connection string of the application's *ConnectionString.config* file (or simply is set to the correct database). For more information on modifying the settings in *ConnectionString.config* file, as well as their purpose, please refer to the "["Creatio setup FAQ"](#)" article. Example of the *<connectionStrings>* section:

```
<connectionStrings>  
  <add name="db" connectionString="Data Source=dbserver\MSSQL2016; Initial  
Catalog=YourDBName; Persist Security Info=True; MultipleActiveResultSets=True;  
Integrated Security = true; Pooling = true; Max Pool Size = 100" />  
  <add name="dbOracle" connectionString="Data Source=(DESCRIPTION = (ADDRESS_LIST =  
(ADDRESS = (PROTOCOL = TCP)(HOST = dbOracleServer.yourdomain.com)(PORT = 1521))  
(CONNECT_DATA = (SERVICE_NAME = TSOral0) (SERVER = DEDICATED)));User  
Id=CreatioUser;Password=CreatioUserPassword;" />  
</connectionStrings>
```

To perform a one-time operation with *WorkspaceConsole*, run the utility with the *webApplicationPath* parameter. Specify the path to the application directory in this parameter. In this case, the utility will independently determine all necessary database connection settings from the *ConnectionString.config* file. The database connection parameters in the *Terrasoft.Tools.WorkspaceConsole.exe.config* file will be ignored.

4. Run one of the two pre-installed .cmd files to install the proper bit version of the utility. For 32-bit operating systems, run *PrepareWorkspaceConsole.x86.bat*. For 64-bit operating systems, run *PrepareWorkspaceConsole.x64.bat*.

If you plan on using *WorkspaceConsole* for operations with SVN, then copy the following files from the ...\\Terrasoft.WebApp\\DesktopBin\\WorkspaceConsole\\x86 catalog (for 32-bit operating systems) or ...\\Terrasoft.WebApp\\DesktopBin\\WorkspaceConsole\\x64 catalog (for 64-bit operating systems):

- SharpPlink-x64.svnExe;
- SharpSvn.dll;
- SharpSvn-DB44-20-x64.svnDll.

Place these files in the ...\\Terrasoft.WebApp\\DesktopBin\\WorkspaceConsole catalog.

WorkspaceConsole parameters

Beginner Easy **Medium** Advanced

Introduction

The WorkspaceConsole utility is designed to work with Creatio packages. Use the utility to:

- Export packages from development environments and migrate them to test environments or production environments (the packages are saved as archives).
- Install new packages when upgrading or migrating from development environments.
- Import and export schema resources and data for localization.
- Create and transfer workspaces between applications.
- Work with configuration schemas.

Because the WorkspaceConsole utility is multifunctional, it must be run with certain parameters. Parameter values are passed as command-line arguments when the utility starts. Parameters are used to configure WorkspaceConsole to perform specific operations. Utility parameters are not case sensitive.

WorkspaceConsole parameters

The -help parameter

Run WorkspaceConsole with this parameter to see the full list of parameters with their brief description. If you specify other parameters, they will be ignored.

The -operation parameter

Specify the required operation name here. This parameter is required. The default value is *LoadLicResponse*. Possible method parameters are listed in table 1.

Table 1. WorkspaceConsole parameters

Operation	Description
<i>LoadLicResponse</i>	Saves licenses to the database (specified in the connection string). The only operation that does not require the <i>-workspaceName</i> parameter.
<i>SaveRepositoryContent</i>	Saves the contents of zip archives specified in the <i>-contentTypes</i> parameter from the directory specified in the <i>-sourcePath</i> parameter to the directory specified in the <i>-destinationPath</i> parameter.
<i>SaveDBContent</i>	Saves database content to the file system. Content type is determined by the <i>contentTypes</i> parameter value. The <i>destinationPath</i> parameter is used to

	specify the path in a file system. One of the following parameters must be specified: <i>-webApplicationPath</i> or <i>-configurationPath</i> . The <i>-confRuntimeParentDirectory</i> parameter must be specified.
<i>SaveVersionSvnContent</i>	Saves the package hierarchy (zip-archives) to the <i>destinationPath</i> directory from several SVN repositories, separated by commas in the <i>sourcePath</i> parameter.
<i>RegenerateSchemaSources</i>	Performs the regeneration of source codes and their compilation. The <i>-confRuntimeParentDirectory</i> parameter must be specified.
<i>InstallFromRepository</i>	Saves the latest version of the SVN structure and metadata into the database. Bound SQL-scripts, source code regeneration, and bound data installation are performed if necessary. This parameter only works with new or modified packages and their elements. One of the following parameters must be specified: <i>-webApplicationPath</i> or <i>-configurationPath</i> . The <i>-confRuntimeParentDirectory</i> parameter must be specified.
<i>InstallBundlePackages</i>	Installs the set of comma-separated packages specified in the <i>-packageName</i> parameter to the workspace specified in the <i>-workspaceName</i> parameter. One of the following parameters must be specified: <i>-webApplicationPath</i> or <i>-configurationPath</i> . The <i>-confRuntimeParentDirectory</i> parameter must be specified.
<i>PrevalidateInstallFromRepository</i>	Checks if zip archive package installation is available.
<i>ConcatRepositories</i>	Merges multiple repositories.
<i>ConcatSVNRepositories</i>	Merges multiple SVNrepositories.
<i>ExecuteProcess</i>	Starts the business process execution in the configuration (if the process is found). The <i>-confRuntimeParentDirectory</i> parameter must be specified.
<i>UpdatePackages</i>	Updates the packages (the <i>-packageName</i> parameter) that are located in the product package hierarchy (the <i>-productPackageName</i> parameter) in the application database. One of the following parameters must be specified: <i>-webApplicationPath</i> or <i>-configurationPath</i> . The <i>-confRuntimeParentDirectory</i> parameter must be specified.
<i>BuildWorkspace</i>	Compiles the workspace (configuration). Used for developing schemas in VisualStudio (see: " Working with the server side source code "). The <i>-confRuntimeParentDirectory</i> parameter must be specified.
<i>ReBuildWorkspace</i>	Compiles the workspace (configuration) entirely. Used for developing schemas in VisualStudio (see: " Working with the server side source code in Visual Studio "). The <i>-confRuntimeParentDirectory</i> parameter must be specified.
<i>UpdateWorkspaceSolution</i>	Updates the Visual Studio project solution and files (see: " Working with the server side source code in Visual Studio ").
<i>BuildConfiguration</i>	Generates static content in the file system (see: " Client static content in the file system "). Uses the following parameters: <i>-workspaceName</i> , <i>-destinationPath</i> , <i>-webApplicationPath</i> , <i>-logPath</i> , <i>-force</i> . If the <i>-force</i> parameter is set to "true", static content is generated for all schemas. If the <i>-force</i> parameter is set to "false", static content is generated for modified schemas only. One of the following parameters must be specified: <i>-webApplicationPath</i> or <i>-configurationPath</i> .

It is necessary to execute the operation *BuildConfiguration* after the operations *InstallFromRepository*, *InstallBundlePackages*, *UpdatePackages* (for versions above 7.11).

The **-user** parameter

Authorization username. Only specified if this information is missing from the configuration utility file or if it is necessary to perform the operation on behalf of another user.

The -password parameter

Authorization password. Only specified if this information is missing from the configuration utility file or if it is necessary to perform the operation on behalf of another user.

The -workspaceName parameter

The name of the workspace (configuration) used to perform the operation.

The -autoExit parameter

Used to automatically terminate the utility process after the operation is completed. Available values – *true* or *false*. Default value – *false*.

The -processName parameter

The name of the process that needs to start.

The -repositoryUri parameter

The SVN directory path for storing the package structure and metadata (optional). Overrides the same configuration property specified in the *-workspaceName* parameter.

The -sourceControlLogin parameter

SVN repository username.

The -sourceControlPassword parameter

SVN repository password.

The -workingCopyPath parameter

Local directory of working package copies, stored in SVN.

The -contentTypes parameter

Content type (for example, resources) extracted from packages. Possible values are listed in table 2.

Table 2. Possible content type values

Content type	Description
<i>SystemData</i>	System diagram data in JSON format. All system schemas and their columns are saved (except for those specified in the <i>-excludedSchemas</i> parameter).
<i>ConfigurationData</i>	Configuration schema data in JSON format. All system schemas and their columns are saved (except for those specified in the <i>-excludedSchemas</i> parameter).
<i>Resources</i>	Resources of localizable configuration schemas in XML format.
<i>LocalizableData</i>	Resources of localizable configuration schemas in XML format. Only text columns are saved. Additional restrictions are specified in the <i>-excludedSchemas</i> and <i>-excludedSchemaColumns</i> parameters.
<i>Repository</i>	Workspace data in zip format.
<i>SqlScripts</i>	Package SQL scripts.
<i>Data</i>	Both system and configuration data in JSON format. A combination of the <i>SystemData</i> and <i>ConfigurationData</i> values.
<i>LocalizableSchemaData</i>	Localizable object data.
<i>All</i>	All content types.

The **-sourcePath** parameter

Local disk catalog path with the necessary data (e.g. packages, schemas, resources). This parameter can take several comma-separated values for the *ConcatRepositories* and *SaveVersionSvnContent* operations.

The **-destinationPath** parameter

Local disk catalog path for the necessary data (e.g. packages, schemas, resources).

The **-webApplicationPath** parameter

The Creatio application path. This path is used by the *ConnectionString.config* file to read database connection data. If this parameter has not been indicated, the connection to the database specified in the connection string of the utility configuration file will be established. If this parameter has been indicated, the connection will be established with the database specified in the *ConnectionString.config* file of the Creatio application.

For *BuildWorkspace*, *ReBuildWorkspace*, and *UpdateWorkspaceSolution* operations, the *-webApplicationPath* parameter must contain the path to the *Terrasoft.WebApp* folder.

The **-configurationPath** parameter

Path to the *Terrasoft.Configuration* subfolder in the application folder. For example, *C:\creatio\Terrasoft.WebApp\Terrasoft.Configuration*. In this folder, source codes and resources of custom package schemas are exported in the **file system development mode**.

The **-filename** parameter

File name. This parameter is required for the *LoadLicResponse* operation.

The **-excludedSchemas** parameter

Names of excluded schemas.

The **-excludedSchemaColumns** parameter

Names of excluded schema columns.

The **-excludedWorkspaceNames** parameter

Names of excluded workspaces.

The **-includedSchemas** parameter

Names of forcibly used schemas.

The **-includedSchemaColumns** parameter

Names of forcibly used schema columns.

The **-cultureName** parameter

The language culture code. Required if you use the *Resources* and/or *LocalizableData* values of the *-contentTypes* parameter.

The **-schemaManagerNames** parameter

Names of schema managers. Default value – *EntitySchemaManager*.

The **-packageName** parameter

The workspace package name (optional parameter). The package is specified in the *-workspaceName* parameter. Please note that all dependent packages will be used as well. If this parameter has not been indicated, all workspace

packages will be used.

The **-clearWorkspace** parameter

Indicates whether the workspace needs to be cleared before updating. Available values – *true* or *false*. Default value – *false*.

The **-installPackageSqlScript** parameter

Indicates the need to execute SQL scripts before and after saving the packages. Available values – *true* or *false*. Default value – *true*.

The **-installPackageData** parameter

Indicates the need to install bound data before and after saving the packages. Available values – *true* or *false*. Default value – *true*.

The **-updateDBStructure** parameter

Indicates the need to update the database structure before and after saving the packages. Available values – *true* or *false*. Default value – *true*.

The **-regenerateSchemaSources** parameter

Indicates the need to regenerate source codes after saving the packages. Available values – *true* or *false*. Default value – *true*.

The **-continueIfError** parameter

Indicates the need to abort the installation process upon encountering the first error. If the parameter value is *true*, the user will receive the error list once the installation is complete. Available values – *true* or *false*. Default value – *false*.

The *InstallFromSvn* and *InstallFromRepository* operations work with new or modified packages and their elements. The system compares the new and modified package structures to identify modified element. If the user runs a command (e.g. *InstallFromSvn*) without specifying the *continueIfError=true* key and receives an error, the command will restart for same configuration without errors, but also without modifying the database. This happens because the previous operation synchronized the package structures and storage of the specified configuration, and the current operation does not have any modified elements.

The **-skipCompile** parameter

Indicates the need to perform a compilation phase. Works only if the *-updateDBStructure* parameter value is *false*. Available values – *true* or *false*. Default value – *false*.

The **-autoUpdateConfigurationVersion** parameter

Updates the configuration version value to the Creatio application version in the database. Available values – *true* or *false*. Default value – *false*.

The **-warningsOnly** parameter

The WorkspaceConsole utility only reports detected errors. Available values – *true* or *false*. Default value – *false*.

The **-confRuntimeParentDirectory** parameter

Path to the parent catalog of the *conf* directory (see "**Client static content in the file system**" and "**Server content in the file system (on-line documentation)**"). This directory is usually located in the *Terrasoft.WebApp* catalog of the deployed application. The parameter is used for compiling or startic content generation operations:

- *RegenerateSchemaSources*
- *InstallFromRepository*

- *InstallBundlePackages*
- *UpdatePackages*
- *BuildWorkspace*
- *RebuildWorkspace*
- *ExecuteProcess*

Exporting packages from database using WorkspaceConsole

Beginner

Easy

Medium

Advanced

Introduction

To transfer custom packages between non-shared environments (e.g. development and test environments), you must first export these packages to the file system. To save packages from the database, use the *SaveDBContent* operation of the *WorkspaceConsole* utility. Learn more about the *WorkspaceConsole* utility in the “[WorkspaceConsole parameters](#)” article.

Make sure the settings of the *WorkspaceConsole* utility are correct before you run it. Please refer to the “[WorkspaceConsole settings](#)” article for more details.

To save packages from the database, run the *WorkspaceConsole* utility with the following parameter values:

Table 1. *WorkspaceConsole* utility parameters for saving database packages

Parameter	Value	Description
<i>operation</i>	<i>SaveDBContent</i>	Saves database content to the file system. Content type is determined by the <i>contentTypes</i> parameter value. The <i>destinationPath</i> parameter is used to specify the path in a file system.
<i>contentTypes</i>	<i>Repository</i>	Type of content uploaded to a file system. The <i>Repository</i> value is used to upload the workspace to a catalog specified in the <i>destinationPath</i> parameter. The name of the workspace is specified in the <i>workspaceName</i> parameter.
<i>workspaceName</i>	<i>[Workspace name]</i>	The name of the workspace (configuration) with the saved packages. By default, all users work in the <i>Default</i> workspace.
<i>destinationPath</i>	<i>[Path to local directory]</i>	Path to a local directory in the file system. Packages with the <i>*.gz</i> format are saved in this directory.
<i>webApplicationPath</i>	<i>[Path to local directory]</i>	The Creatio application path. This path is used by the <i>ConnectionStrings.config</i> file to read database connection data. If this parameter has not been indicated, the connection to the database specified in the connection string of the utility configuration file will be established. If this parameter has been indicated, the connection will be established with the database specified in the <i>ConnectionStrings.config</i> file of the Creatio application.
<i>configurationPath</i>	<i>[Path to local directory]</i>	Path to the <i>Terrasoft.Configuration</i> subfolder in the application folder. For example, <i>C:\creatio\Terrasoft.WebApp\Terrasoft.Configuration</i> . In this folder, source codes and resources of custom package schemas are exported in the file system development mode .

All workspace packages are saved in the process. It may take up to 10 minutes to complete this operation.

Check data binding properties before saving. This includes system settings, lookups, section data etc.

Please refer to the "[Binding data to package](#)" article for more details.

Command signature for Windows command prompt that will export packages from the database:

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

We recommend using batch files (*.bat) to create and save commands.

Uploading packages to a file system

Case description

The Creatio application is installed in the C:\creatio directory. Export all *Default* workspace packages into the C:\SavedPackages directory.

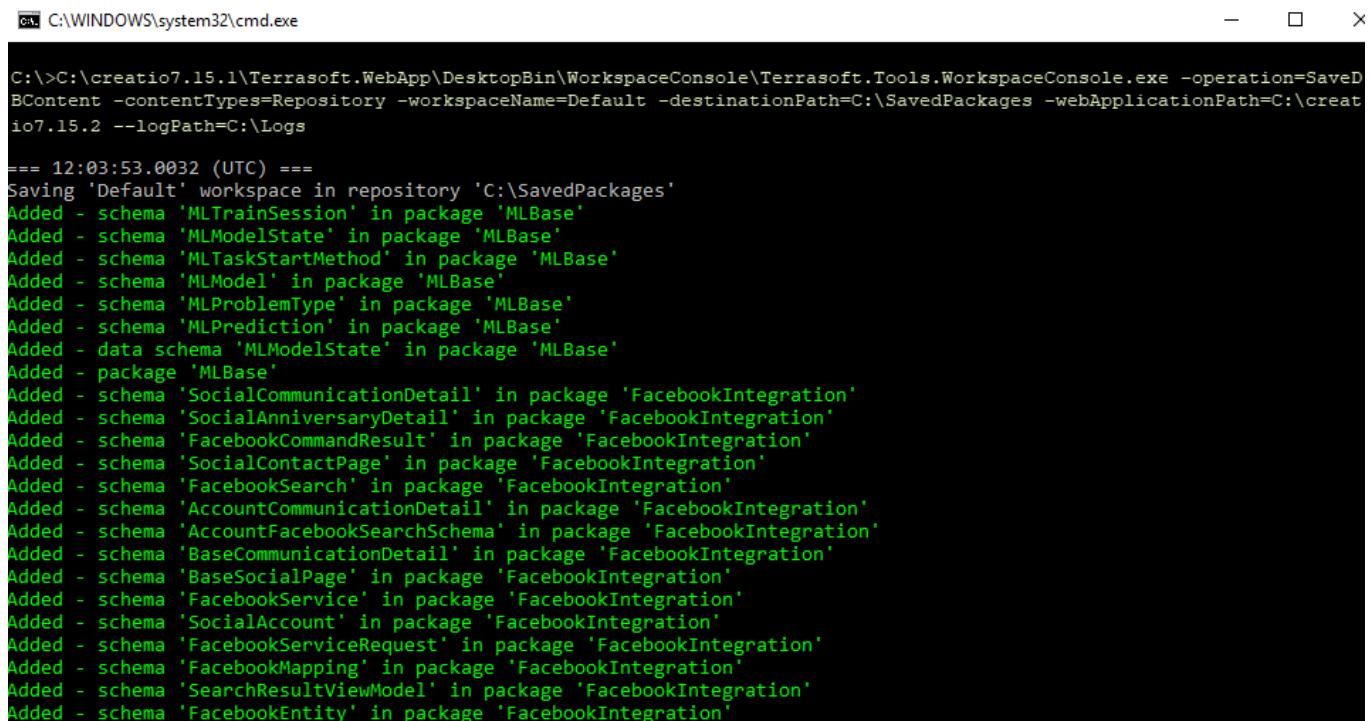
Case implementation:

Use any text editor to create a batch command file (*.bat or *.cmd) with a command that will launch the *WorkspaceConsole* utility. Enter the following command in the file:

```
C:\creatio\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe -operation=SaveDBContent -contentTypes=Repository -workspaceName=Default -destinationPath=C:\SavedPackages -webApplicationPath=C:\creatio --logPath=C:\Logs  
pause
```

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

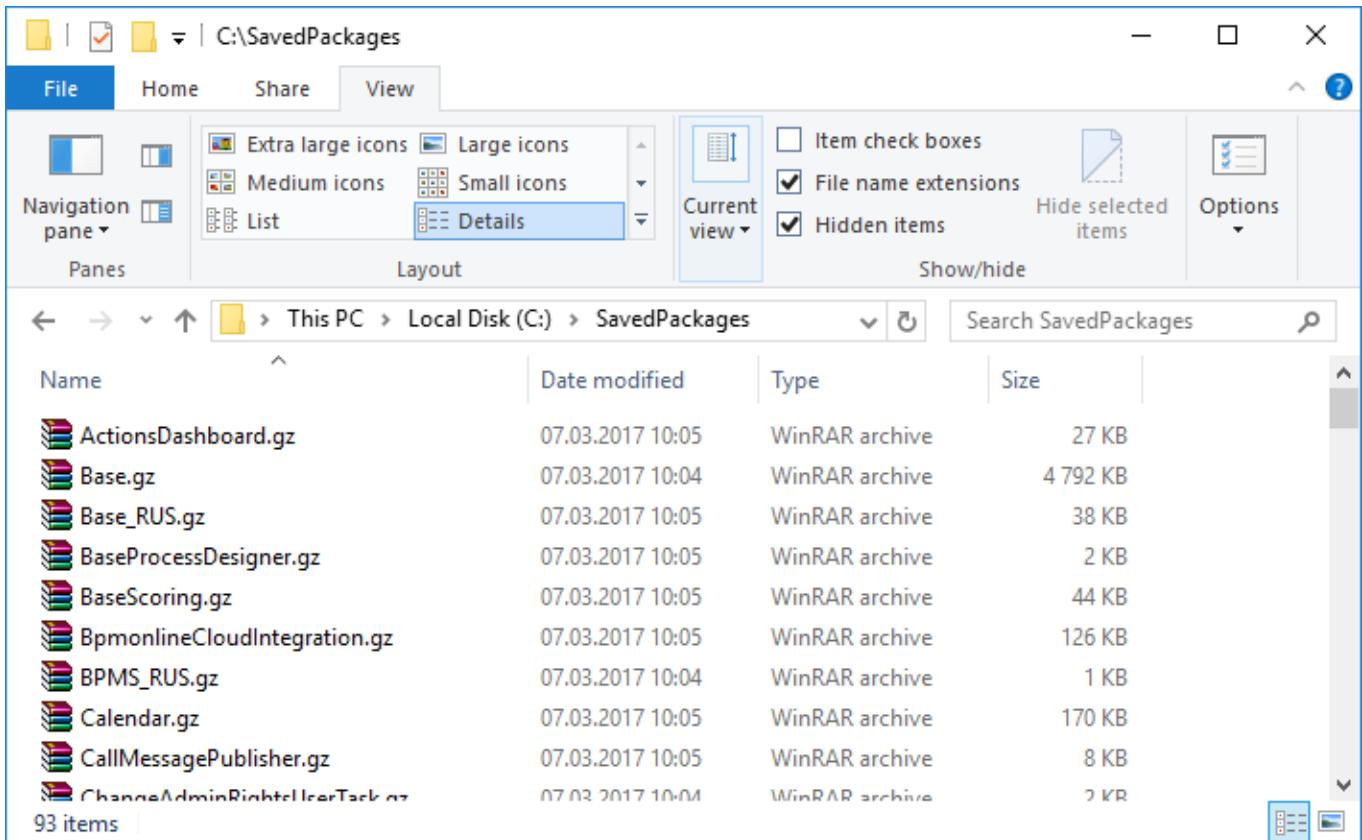
Fig. 1. WorkspaceConsole execution



```
C:\>C:\creatio7.15.1\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe -operation=SaveDBContent -contentTypes=Repository -workspaceName=Default -destinationPath=C:\SavedPackages -webApplicationPath=C:\creatio7.15.2 --logPath=C:\Logs  
== 12:03:53.0032 (UTC) ==  
Saving 'Default' workspace in repository 'C:\SavedPackages'  
Added - schema 'MLTrainSession' in package 'MLBase'  
Added - schema 'MLModelState' in package 'MLBase'  
Added - schema 'MLTaskStartMethod' in package 'MLBase'  
Added - schema 'MLModel' in package 'MLBase'  
Added - schema 'MLProblemType' in package 'MLBase'  
Added - schema 'MLPrediction' in package 'MLBase'  
Added - data schema 'MLModelState' in package 'MLBase'  
Added - package 'MLBase'  
Added - schema 'SocialCommunicationDetail' in package 'FacebookIntegration'  
Added - schema 'SocialAnniversaryDetail' in package 'FacebookIntegration'  
Added - schema 'FacebookCommandResult' in package 'FacebookIntegration'  
Added - schema 'SocialContactPage' in package 'FacebookIntegration'  
Added - schema 'FacebookSearch' in package 'FacebookIntegration'  
Added - schema 'AccountCommunicationDetail' in package 'FacebookIntegration'  
Added - schema 'AccountFacebookSearchSchema' in package 'FacebookIntegration'  
Added - schema 'BaseCommunicationDetail' in package 'FacebookIntegration'  
Added - schema 'BaseSocialPage' in package 'FacebookIntegration'  
Added - schema 'FacebookService' in package 'FacebookIntegration'  
Added - schema 'SocialAccount' in package 'FacebookIntegration'  
Added - schema 'FacebookServiceRequest' in package 'FacebookIntegration'  
Added - schema 'FacebookMapping' in package 'FacebookIntegration'  
Added - schema 'SearchResultViewModel' in package 'FacebookIntegration'  
Added - schema 'FacebookEntity' in package 'FacebookIntegration'
```

Zip-archives containing all *Default* configuration packages will be exported to the C:\SavedPackages directory (Fig. 2).

Fig. 2. Zip-archives with Creatio packages exported to the file system



Saving packages to the database using WorkspaceConsole

Beginner Easy **Medium** Advanced

Introduction

Saving packages from the file system to the application database is performed when transferring custom packages between non-shared environments (e.g. development and test environments). Usually, packages are saved from the development environment, and loaded into the test and production environments. Learn more about saving packages in the “[Exporting packages from database using WorkspaceConsole](#)” and “[Saving SVN packages using WorkspaceConsole](#)” articles.

To load packages to the database, run the *WorkspaceConsole* utility with the following parameters:

Table 1. *WorkspaceConsole* utility parameters for loading packages to the database

Parameter	Value	Description
<i>operation</i>	<i>InstallFromRepository</i>	It saves the contents of packages from archives in the database. Bound SQL-scripts, source code regeneration, and bound data installation are performed if necessary. The <i>InstallFromSvn</i> and <i>InstallFromRepository</i> operations work with new or modified packages and their elements.
<i>packageName</i>	<i>[Package Name]</i>	The name of the package specified in the <i>workspaceName</i> configuration parameter. All dependent packages are used as well. This parameter is optional. This parameter is optional. The <i>-clearWorkspace</i> parameter
<i>workspaceName</i>	<i>[Workspace name]</i>	The name of the workspace (configuration) with the saved packages. By default, all users work in the <i>Default</i> workspace.

<i>sourcePath</i>	<i>[Path to local directory]</i>	Path to a local directory in the file system. This directory should include the required packages in the *.gz format.
<i>destinationPath</i>	<i>[Path to local directory]</i>	Path to a local directory in the file system. The packages from the directory specified in the <i>sourcePath</i> parameter will be saved here.
<i>skipConstraints</i>	<i>false</i>	The option to skip foreign key creation in database tables. Available values – <i>true, false</i> .
<i>skipValidateActions</i>	<i>true</i>	The option to skip the process of table index creation verification when updating the database structure. Available values – <i>true or false</i> .
<i>regenerateSchemaSources</i>	<i>true</i>	Indicates the need to regenerate source codes after saving the packages. Available values – <i>true, false</i> .
<i>updateDBStructure</i>	<i>true</i>	Indicates the need to update the database structure before and after saving the packages. Available values – <i>true, false</i> .
<i>updateSystemDBStructure</i>	<i>true</i>	Indicates the need to update the database structure before and after saving the packages. Creates all missing system table indexes. Available values – <i>true, false</i> .
<i>installPackageSqlScript</i>	<i>true</i>	Indicates the need to execute SQL scripts before and after saving the packages. Available values – <i>true, false</i> .
<i>installPackageData</i>	<i>true</i>	Indicates the need to install bound data before and after saving the packages. Available values – <i>true, false</i> .
<i>continueIfError</i>	<i>true</i>	Indicates the need to abort the installation process upon encountering the first error. If the parameter value is <i>true</i> , the user will receive the error list once the installation is complete. Available values – <i>true, false</i> .
<i>logPath</i>	<i>[Path to local directory]</i>	Path to the operation log. The log name contains the start date and time of the operation.
<i>webApplicationPath</i>	<i>[Path to local directory]</i>	The Creatio application path. This path is used by the ConnectionStrings.config file to read database connection data. If this parameter has not been indicated, the connection to the database specified in the connection string of the utility configuration file will be established. If this parameter has been indicated, the connection will be established with the database specified in the ConnectionStrings.config file of the Creatio application.
<i>configurationPath</i>	<i>[Path to local directory]</i>	Path to the <i>Terrasoft.Configuration</i> subfolder in the application folder. For example, <i>C:\creatio\Terrasoft.WebApp\Terrasoft.Configuration</i> . In this folder, source codes and resources of custom package schemas are exported in the file system development mode .
<i>confRuntimeParentDirectory</i>	<i>[Path to local directory]</i>	Path to the parent catalog of the <i>conf</i> directory (see " Client static content in the file system " and " Server content in the file system (on-line documentation) "). This directory is usually located in the <i>Terrasoft.WebApp</i> catalog of the deployed application.

Command signature for Windows command prompt that will export packages from the database:

[The WorkspaceConsole utility path]\Terrasoft.Tools.WorkspaceConsole.exe -

```
packageName=[Package name] -workspaceName=Default -operation=InstallFromRepository -  
sourcePath=[Path to package archives] -destinationPath=[Archive extraction path] -  
skipConstraints=false -skipValidateActions=true -regenerateSchemaSources=true -  
updateDBStructure=true -updateSystemDBStructure=true -installPackageSqlScript=true -  
installPackageData=true -continueIfError=true -webApplicationPath=[Path to  
application folder] -confRuntimeParentDirectory=[Path to application  
folder]\Terrasoft.WebApp] -logPath=[Log path]
```

It is necessary to execute the operation *BuildConfiguration* after the operation *InstallFromRepository* (for versions above 7.11).

Operation *BuildConfiguration* generates static content in the file system. Uses the following parameters: *-workspaceName*, *-destinationPath*, *-webApplicationPath*, *-logPath*, *-force*. If the *-force* parameter is set to “true”, static content is generated for all schemas. If the *-force* parameter is set to “false”, static content is generated for modified schemas only. One of the following parameters must be specified: *-webApplicationPath* or *-configurationPath* (see: "**WorkspaceConsole parameters**").

The *WorkspaceConsole* utility makes direct changes to the database, and therefore they become available only after restarting the application in IIS.

Packages loaded into the application using *WorkspaceConsole* are considered pre-installed and can not be modified (see: "**Package structure and contents**").

Best practices of loading packages with enabled development mode in the file system

If the *WorkspaceConsole* is used for loading packages to the application with the **development mode in the file system** enabled (*fileDesignMode=true*), the installation of the packages works in a special way. Source code of the modified schemas will be modified in the database but will remain unchanged in the file system. In this regard, if you open the schema in the designer, the unchanged source code from the file system will be displayed. At the same time, the schema modification date is changed, which brings more confusion, as the schema transfer looks completed.

Due to this, it is not recommended to use the *InstallFromRepository* operation for transferring changes from the application with enabled development mode in the file system (*fileDesignMode=true*). If this operation is required, then you will need to perform the [Download packages to file system] action to upload source codes to the file system after operation is complete.

Saving packages to the database

Case description

The Creatio application is installed in the C:\creatio directory. Save the *userPackage* package to the *Default* workspace. The package archive is located in the C:\SavedPackages directory. Extract package contents to the C:\TempPackages directory. Save the operation log file to the C:\Log directory.

Case implementation:

Use any text editor to create a batch command file (*.bat or *.cmd) with a command that will launch the *WorkspaceConsole* utility. Enter the following command in the file:

```
C:\creatio\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe -packageName=sdkBookExample -workspaceName=Default -  
operation=InstallFromRepository -sourcePath=C:\SavedPackages -  
destinationPath=C:\TempPackages -skipConstraints=false -skipValidateActions=true -  
regenerateSchemaSources=true -updateDBStructure=true -updateSystemDBStructure=true -  
installPackageSqlScript=true -installPackageData=true -continueIfError=true -  
webApplicationPath=C:\creatio -confRuntimeParentDirectory=C:\creatio\Terrasoft.WebApp -  
logPath=C:\Log  
pause
```

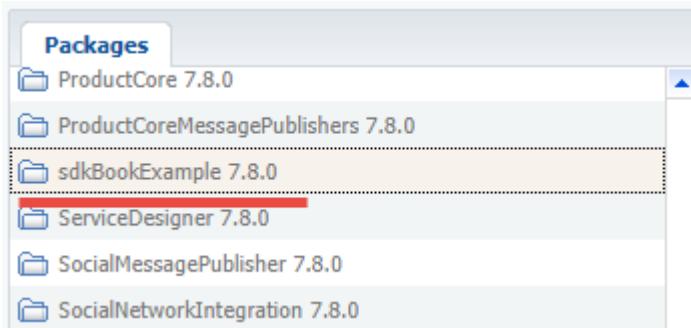
Upon saving the batch file and running it, a console window will appear, and the *WorkspaceConsole* execution process with specified parameter values will be displayed (Fig. 1).

Fig. 1. Saving a package to the application database

```
C:\>C:\creatio7.15.1\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe -p  
ackageName=sdkBookExample -workspaceName=Default -operation=InstallFromRepository -sourcePath=C:\SavedPac  
kages -destinationPath=C:\TempPackages -skipConstraints=false -skipValidateActions=true -regenerateSchema  
Sources=true -updateDBStructure=true -updateSystemDBStructure=true -installPackageSqlScript=true -install  
PackageData=true -continueIfError=true -webApplicationPath=C:\creatio7.15.1 -confRuntimeParentDirectory=C:  
\creatio7.15.1\Terrasoft.WebApp -logPath=C:\Log  
== 13:23:14.6288 (UTC) ==  
Installing package from repository 'C:\SavedPackages' to workspace 'Default'  
  
== 13:23:16.6009 (UTC) ==  
Updating structure of system tables in database  
Structure saved: SysLookup in 0.020 sec  
Structure saved: Account in 0.005 sec  
Structure saved: Contact in 0.007 sec  
Structure saved: SysCulture in 0.003 sec  
Structure saved: SysLanguage in 0.003 sec  
Structure saved: SysAdminUnit in 0.005 sec  
Structure saved: SysAdminUnitGrantedRight in 0.005 sec  
Structure saved: SysUserInRole in 0.005 sec  
Structure saved: SysEntitySchemaColumnRight in 0.007 sec  
Structure saved: SysEntitySchemaRecordDefRight in 0.007 sec  
Structure saved: SysEntitySchemaOperationRight in 0.008 sec  
Structure saved: SysExtServiceOperationRight in 0.010 sec  
Structure saved: SysEntitySchemaReference in 0.047 sec  
Structure saved: SysSettingsRights in 0.005 sec  
Structure saved: SysSettings in 0.006 sec  
Structure saved: SysSettingsValue in 0.002 sec  
Structure saved: SysSettingsReferenceSchema in 0.004 sec  
Structure saved: SysSettingsFolder in 0.002 sec
```

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

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



Saving SVN packages using WorkspaceConsole

Beginner Easy **Medium** Advanced

Introduction

To transfer custom packages between non-shared environments (e.g. development and test environments), you must first export these packages to the file system. To save packages from the SVN repository, use the *SaveVersionSVNContent* operation of the *WorkspaceConsole* utility. Learn more about the *WorkspaceConsole* utility in the “**WorkspaceConsole parameters**” article.

Make sure the settings of the *WorkspaceConsole* utility are correct before you run it. Please refer to the “**WorkspaceConsole settings**” article for more details.

To save SVN packages, run the *WorkspaceConsole* utility with the following parameter values:

Table 1. *WorkspaceConsole* utility parameters for saving SVN packages

Parameter	Value	Description
-----------	-------	-------------

<i>operation</i>	<i>SaveVersionSvnContent</i>	Saves the package hierarchy (zip-archives) to the <i>destinationPath</i> directory from several SVN repositories, separated by commas in the <i>sourcePath</i> parameter.
<i>destinationPath</i>	<i>[Path to local directory]</i>	Path to a local directory in the file system. Packages with the *.gz format are saved in this directory.
<i>workingCopyPath</i>	<i>[Path to local directory]</i>	Local directory of working package copies, stored in SVN.
<i>sourcePath</i>	<i>[SVN repository path]</i>	The SVN directory path for storing the package structure and metadata. Separate the values with a comma to specify multiple directories.
<i>packageName</i>	<i>[Package Name]</i>	The name of the repository package used in the operation. All dependent packages are used as well.
<i>packageVersion</i>	<i>[Package version]</i>	The version of the repository package used in the operation.
<i>sourceControlLogin</i>	<i>[SVN username]</i>	SVN repository username.
<i>sourceControlPassword</i>	<i>[SVN password]</i>	SVN repository password.
<i>cultureName</i>	<i>[Language culture]</i>	The language culture code. For example, <i>es-ES</i> .
<i>excludeDependentPackages</i>	<i>true or false</i>	This checkbox identifies whether the packages need to be saved. The package specified in the <i>packageName</i> parameter depends on this checkbox.
<i>logPath</i>	<i>[Path to local directory]</i>	The path to the directory in which the operation log file will be saved (optional).

Command signature for Windows command prompt that will export packages from the SVN:

```
[Path to WorkspaceConsole]\Terrasoft.Tools.WorkspaceConsole.exe -
operation=SaveVersionSvnContent -destinationPath=[Path to local folder] -
workingCopyPath=[Path to local folder] -sourcePath=[Path to SVN storage] -
packageName=somePackage -packageVersion=7.8.0 -sourceControlLogin=User -
sourceControlPassword=Password -logPath=[Path to local folder] -cultureName=ru-RU -
excludeDependentPackages=true
```

Uploading packages to a file system

Case description

Save the userPackage package to the C:\SavedPackages directory from the SVN repository, found at <http://server-svn:8050/svn/Packages>. Language culture - Russian. Save the operation log file to the C:\Log directory. Place the working copy of the package in the C:\WorkingCopy folder. SVN username – “User”. SVN password – “Password”. The Creatio application is installed in the C:\creatio directory.

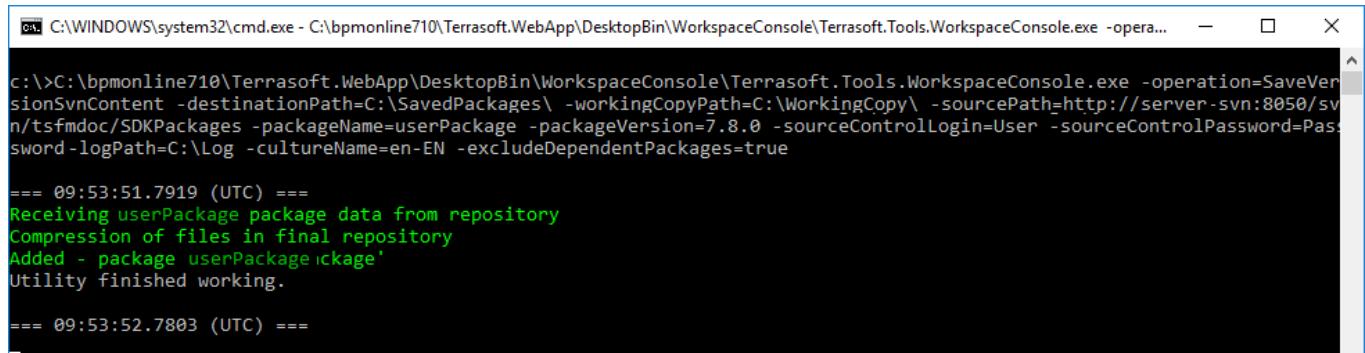
Case implementation:

Use any text editor to create a batch command file (*.bat or *.cmd) with a command that will launch the *WorkspaceConsole* utility. Enter the following command in the file:

```
C:\creatio\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe -
operation=SaveVersionSvnContent -destinationPath=C:\SavedPackages\ -
workingCopyPath=C:\WorkingCopy\ -sourcePath=http://server-svn:8050/svn/Packages -
packageName=userPackage -packageVersion=7.8.0 -sourceControlLogin=User -
sourceControlPassword=Password -logPath=C:\Log -cultureName=ru-RU -
excludeDependentPackages=true
pause
```

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

Fig. 1. The process of saving a package from the repository



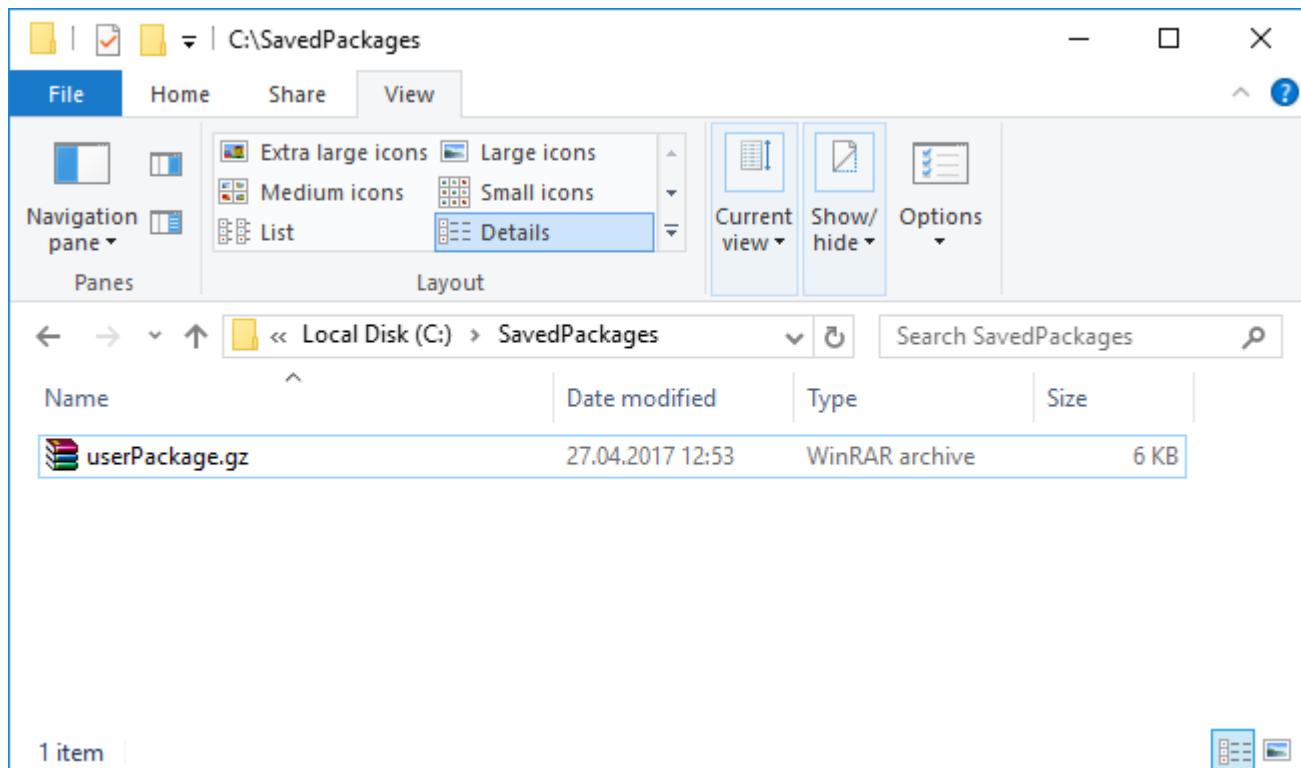
```
c:\>C:\bpmonline710\Terrasoft.WebApp\DesktopBin\WorkspaceConsole\Terrasoft.Tools.WorkspaceConsole.exe -operation=SaveVersionSvnContent -destinationPath=C:\SavedPackages\ -workingCopyPath=C:\WorkingCopy\ -sourcePath=http://server-svn:8050/svn/tsfmdoc/SDKPackages -packageName=userPackage -packageVersion=7.8.0 -sourceControlLogin=User -sourceControlPassword=Password -logPath=C:\Log -cultureName=en-EN -excludeDependentPackages=true

== 09:53:51.7919 (UTC) ==
Receiving userPackage package data from repository
Compression of files in final repository
Added - package userPackage package
Utility finished working.

== 09:53:52.7803 (UTC) ==
```

The *userPackage* package will be exported to the C:\SavedPackages directory (Fig. 2).

Fig. 2. Saved zip-archive with userPackage package



Transfer changes using *WorkspaceConsole*

Beginner Easy **Medium** Advanced

Introduction

The **WorkspaceConsole** utility is designed to work with Creatio packages. Use the utility to:

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

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

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

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

Benefits of transferring changes via the WorkspaceConsole

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

The recommended steps for transferring changes via WorkspaceConsole

1. Check the data binding.

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

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

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

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

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

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

Table 1. WorkspaceConsole parameters for exporting database packages

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

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

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

```
destinationPath=[Path to local directory]
```

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

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

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

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

Table 2. *WorkspaceConsole* parameters for database packages load

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

logPath

[Path to local directory]

errors are detected. The user will receive the list of errors after the installation. Can be *True* or *false*.

A path to the directory where the log of the completed operation will be created. The file name consists of the date and time of the operation launch.

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

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

5. Restart the application in IIS

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

Server-side development

Contents

- **Overview**
- **Class Description**
- **Examples**

Overview

Contents

- **Replacement class object factory**
- **Configuration web-services**
- **Working with database**
- **Localizable configuration resources**
- **Entity event layer**
- **Developing the source code in the file content (project package)**

Replacement class object factory

Beginner

Easy

Medium

Advanced

Introduction

Creatio development is based on the main practices of object-oriented programming, including the open-closed principle. According to this approach, all entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that new logic and features must be implemented by introducing new entities rather than modifying existing ones. The open-closed principle is the basis of the Creatio **package mechanism**.

Configuration elements in the pre-installed packages cannot be modified. Developing additional functionality and modifying the existing functionality is done solely via **custom packages**. Learn more about working with packages in the “**Development tools. Packages**” block of articles.

Customizing the configuration may require changing or extending the base functionality. Retrofitting the logic of preinstalled objects is restricted system-wide. The user should use the replacement mechanism to solve such problems.

To change the behavior of a preinstalled object, create a new custom package object that extends the preinstalled

object. For the new object, set a property implying that the object will replace the parent object in the object hierarchy. All modifications that have to be applied to the preinstalled object are to be implemented in the replacement object. When calling the preinstalled object, the system will then run the logic of the corresponding replacement object.

The same base object can be replaced in multiple custom packages. The resulting replacing object implementation in the compiled configuration is based on the hierarchy of packages that contain the replacing objects.

Replacement mechanism description

Creating a replacing configuration element

To create a replacing configuration element:

1. In a custom package that will contain the replacing entity, set a dependency on the package that contains the replaceable entity.
2. Create a replacing object or replacing page. To create a replacing object, run **[Add] → [Replacing Object]** in the **[Configuration]** section. To create a replacing page, run **[Add] → [Replacing Page]** in the **[Configuration]** section.

Creating a replacement class

Specific configuration tasks may require modification of the program code of such schemas as **[Source Code]**. In that case, use classes implemented in the schemas of custom packages as replacement objects.

Replacement classes follow roughly the same procedure as replacement configuration elements, however, creating and using replacement class instances has certain specifics.

To create a replacement class:

1. In a custom package that will contain the replacement class, set a dependency on the package that contains the class to replace.
2. In a custom package, create a replacement class that extends the replaceable class.
3. Mark the replacement class with an *[Override]* attribute.
4. Implement the logic of the replacement class (for example, add new properties and methods that extend the functionality of the replaceable class, overload a method of the replaceable class to change its behavior, etc).

A replacement class is instantiated by the replacement class object factory. The factory is queried for an instance of the replaceable class, and then returns an instance of a corresponding replacement class, which is computed using a dependency tree in the source code schemas.

The specifics of the implementation and use of the factory are described below, as well as a use case for creating and using replacement class instances.

Replacement class object factory

Override attribute

The *[Override]* attribute type is described in the [Terrasoft.Core.Factories](#) namespace. The attribute extends the [System.Attribute](#) base type and will only apply to classes. Classes marked with this attribute are used by the factory to build a dependency tree of replaceable and replacement classes.

A use case of applying the *[Override]* attribute to the *MySubstituteClass*, which replaces *SubstitutableClass*, is available below.

```
[Override]
public class MySubstituteClass : SubstitutableClass
{
    // Class implementation.
}
```

Replacement factory details. The ClassFactory class

The [ClassFactory](#) static class is an implementation of a software factory for Creatio replacement objects. The software factory relies on the [Ninject](#) open-source framework for dependency injection. Essentially, the factory

collects data about all replaceable configuration types during the initialization phase and configures the framework core accordingly. The framework then returns instances of the required types based on the configured dependencies.

The factory is initialized at the moment of the first call, i.e. when attempting to retrieve an instance of a replacement. This is what happens during the call:

1. The replacement factory checks if replacement types are found among the types of the configuration build. The factory interprets a descendant class marked with the *[Override]* attribute as a replacement class and the parent as a replaceable class. The factory builds a dependency tree as a list of *[Replaced type] -> [Replacing type]* pairs. Transitional types are not accounted for in the replacement tree. As a result, the origin class is replaced with the last descendant in the replacement hierarchy.

You can find a class hierarchy below.

```
// Source class.  
public class ClassA { }  
  
// Class that replaces ClassA.  
[Override]  
public class ClassB : ClassA { }  
  
// Class that replaces ClassB.  
[Override]  
public class ClassC : ClassB { }
```

Using this hierarchy, the replacement factory will build the following dependency tree:

ClassA → ClassC

ClassB → ClassC

Then, the type replacement hierarchy will look like this: ClassA → ClassB → ClassC. That means, ClassA will be replaced with ClassC, which is the last descendant in the replacement hierarchy, not with the transitional ClassB type. Consequently, when queried for a ClassA or ClassB instance, the replacement factory will return a ClassC instance.

2. Using Ninject, the factory binds replacement types based on the constructed type dependency tree.

Creating a replaceable type instance

ClassFactory provides the [Get<T>](#) parametrized public static method to get an instance of a replacement type. The replaceable type serves as a generic method parameter.

Getting an instance of a class that replaces the *SubstitutableClass* type is covered below. Explicit typing of a new class instance is unnecessary. Due to the preliminary initialization, the factory will determine what type will substitute the replaceable type, and then create an instance of the replacement type.

```
var substituteObject = ClassFactory.Get<SubstitutableClass>();
```

The *Get<T>* method can accept an array of *ConstructorArgument* objects as an input parameter. Each of the objects is an argument for the constructor of the class created by the factory. This way, the factory enables instantiation of replacement objects with parameterized constructors. The factory core independently resolves all of the dependencies required for the creation or operation of the object.

In most situations, we recommend that the constructors of the replacement classes have the exactly same signature as the parent class. If the implementation logic of a replacement class requires declaring a constructor with a custom signature, make sure to comply with the following rules for creating and calling the constructors of the replaceable and replacement classes:

1. If the replaceable class does not have an explicitly parameterized constructor (the class only has a default constructor), then the replacement class may have an explicitly parameterized constructor without any restrictions. The standard order of calls to the constructors of the parent and child classes must be respected. When instantiating the original (replaceable) class via the factory, make sure to pass the correct parameters to the factory to initialize the properties of the class that will eventually replace the original one. Failure to comply with this rule will result in a runtime error.
2. If the replaceable class has a parameterized constructor, then the replacement class must implement the constructor of the parent class. The constructor of the replacement class must explicitly call the parameterized constructor of the parent (replaceable) class and pass parameters for the correct initialization

of the parent properties. Other than that, the constructor of the replacement class may initialize own properties or remain empty. Failure to comply with this rule will result in a runtime error. The responsibility for the correct initialization of the properties of the replaceable and replacement objects ultimately lies with the developer. Below is a series of sample cases that illustrate instantiation cases of replaceable types with parameterized constructors.

Cases

Case example 1

The `SubstitutableClass` replaceable class has one default constructor. Two constructors are defined in the `SubstituteClass` replacement class: a default constructor and a parameterized constructor. The replacement class overloads the `GetMultipliedValue()` parent method. Options for instantiating and invoking the `GetMultipliedValue()` method with the default and parameterized constructors are listed below.

```
// Declaring a replaceable class.
public class SubstitutableClass
{
    // Class property to initialize in the constructor.
    public int OriginalValue { get; private set; }

    // Default constructor, which initializes the OriginalValue property with 10.
    public SubstitutableClass()
    {
        OriginalValue = 10;
    }

    // The method returns OriginalValue multiplied by 2.
    // This method can be reloaded
    // reloaded in replacement classes.
    public virtual int GetMultipliedValue()
    {
        return OriginalValue * 2;
    }
}

// Declaring a class to replace SubstitutableClass.
[Terrasoft.Core.Factories.Override]
public class SubstituteClass : SubstitutableClass
{
    // SubstituteClass property.
    public int AdditionalValue { get; private set; }

    // Default constructor, which initializes the AdditionalValue property with 15.
    // Calling
    // the constructor of the SubstitutableClass parent class is not required, since
    // the parent class only provides
    // the default constructor (called implicitly when instantiating
    // SubstituteClass).
    public SubstituteClass()
    {
        AdditionalValue = 15;
    }

    // Parameterized constructor, which initializes the AdditionalValue property with
    // a value
    // passed as the parameter. Likewise, the parent constructor is not called
    // explicitly.
    public SubstituteClass(int paramValue)
    {
        AdditionalValue = paramValue;
    }
}
```

```
// Replacing a parent method. // The method returns AdditionalValue multiplied by  
3.  
public override int GetMultipliedValue()  
{  
    return AdditionalValue * 3;  
}  
}
```

Options for getting an instance of the replacement class using the factory are listed below.

```
// Getting an instance of the class that replaces SubstitutableClass.  
// The factory will return an instance of SubstituteClass initialized by the  
constructor without parameters.  
var substituteObject = ClassFactory.Get<SubstitutableClass>();  
  
// The variable value will equal 10. The OriginalValue property is initialized by the  
default  
// parent constructor, which is implicitly called during the instantiation of the  
replacement class.  
var originalValue = substituteObject.OriginalValue;  
  
// A call to the replacement class method that will return the value of  
AdditionalValue  
// Multiplied by 3. The variable value will equal 45 since AdditionalValue is  
// initialized with 15.  
var additionalValue = substituteObject.GetMultipliedValue();  
  
// Getting an instance of the replacement class initialized with a parameterized  
// constructor. Note that the name of the ConstructorArgument parameter must be the  
same as the name of  
// the parameter in the class constructor.  
var substituteObjectWithParameters = ClassFactory.Get<SubstitutableClass>(  
    new ConstructorArgument("paramValue", 20));  
  
// The variable value will equal 10.  
var originalValueParametrized = substituteObjectWithParameters.OriginalValue;  
  
// The variable value will equal 60 since AdditionalValue is  
// initialized 20.  
var additionalValueParametrized =  
substituteObjectWithParameters.GetMultipliedValue();
```

Case example 2

The *SubstitutableClass* replaceable class has one parameterized constructor. The *SubstituteClass* replacement class has one parameterized constructor as well.

```
// Declaring a replaceable class  
public class SubstitutableClass  
{  
    // Class property to initialize in the constructor  
    public int OriginalValue { get; private set; }  
  
    // Parameterized constructor, which initializes the OriginalValue property with a  
value  
    // passed as the originalParamValue parameter  
    public SubstitutableClass(int originalParamValue)  
    {  
        OriginalValue = originalParamValue;  
    }
```

```
// The method returns OriginalValue multiplied by 2. // This method can be
reloaded
    // reloaded in replacement classes
    public virtual int GetMultipliedValue()
    {
        return OriginalValue * 2;
    }
}

// Declaring a class to replace SubstitutableClass.
[Terrasoft.Core.Factories.Override]
public class SubstituteClass : SubstitutableClass
{
    // SubstituteClass property.
    public int AdditionalValue { get; private set; }

    // Parameterized constructor, which initializes the AdditionalValue property with
    // a value
    // passed as the parameter. You must call the parent constructor explicitly to
    // initialize the
    // the parent property. If you do not do that, the compilation will end with an
    // error.
    public SubstituteClass(int paramValue) : base(paramValue + 8)
    {
        AdditionalValue = paramValue;
    }

    // Replacing a parent method. // The method returns AdditionalValue multiplied by
    // 3.
    public override int GetMultipliedValue()
    {
        return AdditionalValue * 3;
    }
}
```

Below, you can find an example of getting and using an instance of the replacement class using the factory.

```
// Getting an instance of the replacement class initialized with a parameterized
// constructor. Note that the name of the ConstructorArgument parameter must be the
// same as the name of
// the parameter in the class constructor.
var substituteObjectWithParameters = ClassFactory.Get<SubstitutableClass>(
    new ConstructorArgument("paramValue", 10));

// The variable value will equal 18.
var originalValueParametrized = substituteObjectWithParameters.OriginalValue;

// The variable value will equal 30.
var additionalValueParametrized =
substituteObjectWithParameters.GetMultipliedValue();
```

Case example 3

The *SubstitutableClass* replaceable class has one parameterized constructor as well. The replacement *SubstituteClass* class does not introduce any new properties but reloads the *GetMultipliedValue()* method, which will now return a fixed value. The *SubstituteClass* class does not require the initialization of its properties. It must explicitly declare a constructor that calls the parent constructor with the parameters to initialize the parent properties correctly.

```
// Declaring a replaceable class
public class SubstitutableClass
{
```

```
// Class property to initialize in the constructor
public int OriginalValue { get; private set; }

// Parameterized constructor, which initializes the OriginalValue property with a
value
// passed as the originalParamValue parameter
public SubstitutableClass(int originalParamValue)
{
    OriginalValue = originalParamValue;
}

// The method returns OriginalValue multiplied by 2. // This method can be
reloaded
// reloaded in replacement classes
public virtual int GetMultipliedValue()
{
    return OriginalValue * 2;
}

// Declaring a class to replace SubstitutableClass.
[Terrasoft.Core.Factories.Override]
public class SubstituteClass : SubstitutableClass
{
    // Empty default constructor that calls the parent class constructor explicitly
    // to initialize the parent properties correctly.
    public SubstituteClass() : base(0)
    {

    }

    // You can also use an empty parameterized constructor to pass the parameters
    // to the parent class constructor.
    public SubstituteClass(int someValue) : base(someValue)
    {

    }

    // Replacing a parent method. The method will return the hardcoded value.
    public override int GetMultipliedValue()
    {
        return 111;
    }
}
```

You can find an example of creating a replacement class in the "[Creating replacement classes in packages](#)" article.

Repositories. Types and recommendations on use

Beginner

Easy

Medium

Advanced

Introduction

In Creatio, business data storage is implemented as a relational database (MS SQL, Oracle or PostgreSQL). The data is used for solutions of different user tasks, execution of business processes, etc. The application ensures execution of the following operations:

1. Storage of user and application data (user profile, session data etc.).
2. Data communication between web farms.
3. Intermediate data storage at the time of restart of application or webfarm.
4. Execution of intermediate actions with data before their placing in repository.

For the purpose of these tasks, data repository technology is implemented in Creatio's architecture. The object model of classes, being a unified API for access from application to data, located in an external repository (currently [Redis](#) is used in Creatio as the external repository) forms the base of this technology.

Creatio repositories focus on execution of service functions or arrangements of data handling but they also can be used for solving user tasks (in configuration business logic).

Redis as a Creatio repository server

Creatio repository server is represented by [Redis](#) – a high-efficient non-relational data repository.

The data model of Redis is based on "key-object" pairs. Redis supports access only with a unique key and can be successfully used for storage of serialized objects. Data, placed into the repository, is stored as binary serialized objects in Creatio.

Redis supports several data storage strategies:

- Storage of data only in memory. Persistent database is converted to caching server.
- Periodical saving of data to disc (by default) – periodical copy saving (snapshot) (once per 1-15 minutes depending on time for creation of previous copy and number of changed keys).
- Transaction log. A synchronous record of each change in special append-only log-file.
- Replication. You can assign a primary to each server. Then all changes in master will be reproduced in the replica.

The definite data storage method is determined by configuring the Redis server. In Creatio (at present), data is stored in memory with periodic saving of data to disc but data replication function is not supported.

For more details on Redis see [official documentation](#).

Creatio repository type. Data storage and cache.

Creatio supports two types of repositories, i.e. data and cache repositories.

A *data repository* is designed for intermediate storage of rarely modified "long-term" data. A *cache* repository stores operation data.

Individual logical levels of data placement are additionally determined for each type of repository (table 1, 2).

Table 1. Data repository levels

Level	Details
<i>Request</i>	The query level. The data of this level is available only in the course of current query processing. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Request</i> enumeration value.
<i>Session</i>	The session level. The data of this level is available only in the session of the current user. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Session</i> enumeration value.
<i>Application</i>	The application level. The data is available for the entire application. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Application</i> enumeration value.

Table 2. Cache levels

Level	Details
<i>Session</i>	The session level. The data of this level is available only in the session of the current user. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Session</i> enumeration value.
<i>Workspace</i>	The level of the workspace. The data of this level is available for all users of one and the same workplace. Corresponds to the <i>Terrasoft.Core.Store.CacheLevel</i> enumeration value.
<i>Application</i>	The application level. The data of this level is available for all

application users regardless of their workspace.
Corresponds to the *Terrasoft.Core.Store.CacheLevel* enumeration value.

Such differentiation of repositories is implemented for the purpose of logical separation of units in data repository and convenience of further use of the source code. Additionally, such separation creates solutions for the following tasks:

- Isolation of data between workspaces and user sessions
- Conditional classification of data
- Control of the data life cycle

All repository and cache data can be located physically on an abstract data storage server. The exception is the data of the *Request* level repository that are stored directly in the memory.

At present, the Creatio repository server is Redis. In common cases, it may be a user repository, accessed through unified interfaces. It is necessary to take into account the fact that repository access operations are resource-intensive since they are associated with serialization / deserialization of data and network communication.

The data repository and cache represent the following possibilities for data handling:

- Access to data by key for reading / recording
- Deletion of data from repository by key

The key difference between a data repository and cache is the control of the object life cycle within them.

Data will be stored in the storage till the explicit deletion. The life cycle of such objects is limited by query execution time (for data of *Request* level repository) or session existence time (for data of *Session* level repository).

There is such a notion as *aging time* for data. It determines time limit of validity of cache items. All items are deleted from cache, regardless of aging time, in the following cases:

- Session ending (items of cache and data repository of *Session* level)
- Implicit deletion of workspace (items of *Workspace* level cache)

The items of the *Application* level cache are stored for the entire period of existence of the application and can be deleted only by clearing the external repository.

Data can be deleted from cache in any period of time. This may result in situations when the code tries to get cached data that has already been deleted from cache at the time of access. In this case, the calling code should only receive data from the persistent repository and place them in cache.

Object model of Creatio repositories

Creatio classes and interfaces for data repository and cache operations

Creatio uses a series of classes and interfaces (located in *Terrasoft.Core.Store* name space) to work with the database and cache (“**.NET class libraries of platform core (on-line documentation)**”). The main ones are listed below.

IBaseStore base repository

Base functions of all types of repositories are represented by *IBaseStore*. Properties and methods of this interface are used to implement the following:

- Access to data by key for reading/recording (*this[string key]* indexer)
- Deletion of data from the repository by set key (*Remove(string key)* method)
- Initialization of the repository by set parameter list (*Initialize(IDictionary<string, string> parameters)* method). At present, parameters for initialization of Creatio repositories are read from configuration file. Parameter lists are set in *storeDataAdapter* (for data repository) and *storeCacheAdapter* (for cache) sections. In common cases, parameters can be set randomly.

IDataStore data repository

The interface determines the specifics of data repository operations. It is an inheritor of *IBaseStore* repository base interface. The interface provides an additional possibility for getting the list of all repository keys (the *Keys*

property).

We recommend using the *Keys* property upon data repository operations only in exceptional cases when it is impossible to solve tasks by alternative methods.

ICacheStore cache storage

The interface determines the specifics of cache operations. It is an inheritor of *IBaseStore* repository base interface. The *GetValues(IEnumerable keys)* property returns cache object dictionary with set keys. This method optimizes repository operation upon simultaneous receiving of data set. This method enables the optimization of working with the repository while simultaneously receiving a dataset.

The Store class

Static class for access to caches and data repositories of different levels. Levels of data repositories and cache are determined in the *Terrasoft.Core.Store.DataLevel* and *CacheLevel* enumerations (table 1 and 2).

The *Store* class has 2 static properties:

- The *Data* property returns an instance of the data repository provider.
- The *Cache* property returns an instance of the cache provider.

Example 1 demonstrates working with the cache and the data store using the *Store* class.

Example 1

```
// Getting a link to the session-level data repository.  
IDataStore dataStore = Store.Data[DataLevel.Session];  
  
// Place the "Data Test Value" value with the "DataKey" key in the data repository.  
dataStore["DataKey"] = "Data Test Value";  
  
// Getting a link to the application-level cache of the workspace.  
ICacheStore cacheStore = Store.Cache[CacheLevel.Workspace];  
  
// Removing an item with the "CacheKey" key from the cache.  
cacheStore.Remove("CacheKey");
```

Access to data repositories and cache from UserConnection

Access to the data repositories and application caches from the configuration code can be obtained using the properties of the static *Store* class of the *Terrasoft.Core.Store* name space. An alternative way to access the data repository and cache in the configuration logic, which avoids the use of long property names and the connection of additional assemblies, is accessed through an instance of the *UserConnection* class.

The *UserConnection* class implements a number of helper properties that enable quick access to data repositories and caches of various levels:

- *ApplicationCache* returns a link to the application-level cache.
- *WorkspaceCache* returns a link to the workspace cache.
- *SessionCache* returns a link to the session-level cache.
- *RequestData* returns a link to the query-level data repository.
- *SessionData* returns a link to the session-level data repository.
- *ApplicationData* returns a link to the application-level data repository.

In most cases, accessing the *UserConnection* properties is identical to accessing the *Store.Cache* and *Store.Data* properties, indicating the appropriate levels. However, in some situations (e.g., when running business processes using a scheduler), another implementation may be used. Therefore, it is recommended to use the properties of the *UserConnection* object in the configuration logic for accessing the repositories.

Example 2

```
// The key with which the value will be placed in the cache.
```

```
string cacheKey = "SomeKey";  
  
// Putting the value into the session-level cache via the UserConnection property.  
UserConnection.SessionCache[cacheKey] = "SomeValue";  
  
// Retrieving a value from the cache via the Store property.  
// As a result, the variable valueFromCache will contain the value "SomeValue".  
string valueFromCache = Store.Cache[CacheLevel.Session][cacheKey] as String;
```

Using cache in EntitySchemaQuery

EntitySchemaQuery implements a mechanism for working with the repository (Creatio cache or user-defined arbitrary repository). Working with the cache allows you to optimize the efficiency of operations by accessing cached query results without additional access to the data repository. When the *EntitySchemaQuery* query is executed, the data received from the database on the server is placed in the cache, which is determined by the *Cache* property with the key that is set by the *CacheItemName* property. By default, the cache of the *EntitySchemaQuery* request is the session-level Creatio cache with local storage. In general, the query cache can be an arbitrary repository that implements the *ICacheStore* interface.

Example 3 shows the process of working with the application cache when executing the *EntitySchemaQuery* request. The code in the example builds a query that returns a list of all cities in the *City* schema. When you receive the results of the query (after calling the *GetEntityCollection()* method), these results are placed in the cache, which can then be used to retrieve the collection of query elements without additional access to the repository.

Example 3

```
// Creating an instance of the EntitySchemaQuery request with the root City schema.  
var esqResult = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");  
  
// Add the city name column to the query.  
esqResult.AddColumn("Name");  
  
// Identifying the key under which the results of query execution will be stored in  
// the cache.  
// Session level cache with local data caching (since Cache property of the object is  
// not determined)  
// is used as cache.  
esqResult.CacheItemName = "EsqResultItem";  
  
// Execution of query to database for receiving resultant object collection.  
// Query results will be placed in cache after completion of this operation. Upon  
// further access to  
// esqResult for receiving object collection query (if query was not changed) these  
// objects will  
// be taken from session cache.  
esqResult.GetEntityCollection(UserConnection);
```

Repository and cache proxy classes

Proxy class notion and purpose

Repositories and caches can be accessed either directly (through the *Store* class properties) or through proxy classes in Creatio.

Proxy classes are special objects that act as intermediate links between repositories and calling code. Proxy classes allow intermediate operations with data before they are read / edited in the repository. Each proxy class is a repository in itself.

Proxy class applications

1. Initial setup and configuration of the application

You can configure the use of proxy classes for data repositories and caches in the Web.config configuration file. The proxy classes for the corresponding data repository or cache are configured in the *storeDataAdapter* and *storeCacheAdapter* sections, respectively. They add a *proxies* section, which lists all the proxy classes that are applied to the repository (example 4).

Example 4

```
<storeDataAdapters>
    <storeAdapter levelName="Request" type="RequestDataAdapterClassName">
        <proxies>
            <proxy name="requestDataProxyName1" type="requestDataProxyClassName1" />
            <proxy name="requestDataProxyName2" type="requestDataProxyClassName2" />
            <proxy name="requestDataProxyName3" type="requestDataProxyClassName3" />
        </proxies>
    </storeAdapter>
</storeDataAdapters>

<storeCacheAdapters>
    <storeAdapter levelName="Session" type="SessionCacheAdapterClassName">
        <proxies>
            <proxy name="sessionCacheProxyName1" type="sessionCacheProxyClassName1" />
            <proxy name="sessionCacheProxyName2" type="sessionCacheProxyClassName2" />
        </proxies>
    </storeAdapter>
</storeCacheAdapters>
```

When the application is downloaded, the settings are read from the configuration file and applied to the corresponding repository type. This enables you to build chains of proxy classes that will be executed sequentially one after another. The order of the proxy classes in the execution chain corresponds to their order in the configuration file. In this case, the proxy class listed last in the *proxies* section is the first in the chain of execution, i.e. the execution is done "from the bottom up".

Take into account the following:

- The "final point" of the application of proxy classes is a definite cache or data repository, for which this chain is determined.
- A separate proxy class knows the cache or repository it works with. The link to it is determined by *ICacheStoreProxy.CacheStore* property or *IDataStoreProxy.DataStore* property. A proxy class doesn't know what exactly this property refers to, i.e. other proxy class, final repository or cache. At the same time, this proxy can act as repository, with which other proxy classes can work.

Thus, according to the settings shown in Example 4, the chain of execution of proxy classes (e.g., data repository) will be as follows: *requestDataProxyName3* > *requestDataProxyName2* > *requestDataProxyName1* > *requestDataAdapterClassName* (final query level data repository).

2. Delimiting data of different application users

Isolating data between different application users is done with the help of proxy classes. The simplest solution to this problem is to transform the value keys before placing them in the repository (e.g., by adding an additional prefix to the key that is specific to the user). Using such proxy classes ensures the uniqueness of storage keys. This, in turn, helps to avoid data loss and corruption while different users simultaneously try to write different values in the repository with the same key. An example of working with key transformation proxy classes is shown below. Generally speaking, you can implement as complex logic as you like in proxy classes.

3. Performing other intermediate actions with the data before placing it in the repository.

In proxy classes, you can implement the logic of performing any arbitrary actions with data before placing or getting it from the repository. Putting the logic of processing data into a proxy class helps avoid code duplicates, which in turn facilitates its maintenance.

Basic proxy class interfaces

A class must implement one (or both) of the *Terrasoft.Core.Store* namespace interfaces in order to be used as a proxy for working with repositories:

- *IDataStoreProxy* is an interface for proxy classes of the data storage
- *ICacheStoreProxy* is the interface of cache proxy classes.

Each of these interfaces has one property - a reference to that storage (or cache) with which the given proxy class is working. For the *IDataStoreProxy* interface, it is the *DataStore* property. For the *ICacheStoreProxy* interface, it is the *CacheStore* property.

Key transformation proxy classes

In Creatio, there is a number of proxy classes that implement the logic of transformation of value keys placed in the repository.

1. The KeyTransformerProxy class

This is an abstract class which acts as a base class for all proxy classes that convert cache keys. It implements the methods and properties of the *ICacheStoreProxy* interface. When creating custom proxy classes for key transformation, it is recommended to inherit from this class to avoid logic duplicates.

2. The PrefixKeyTransformerProxy

A proxy class that converts cache keys by adding a prefix to them.

Example 5 demonstrates working with the session-level cache via the *PrefixKeyTransformerProxy* proxy class.

Example 5

```
// Key, with which a value will be placed in cache through proxy.
string key = "Key";

// Prefix that will be added to value key by proxy class.
string prefix = "customPrefix";

// Creation of proxy class that will be used for recording of values into session
level cache.
ICacheStore proxyCache = new PrefixKeyTransformerProxy(prefix,
Store.Cache[CacheLevel.Session]);

// Recording value with the "key" key in cache through proxy class.
// This value is recorded into global cache of session level with "prefix+key" key.
proxyCache[key] = "CachedValue";

// Receiving the value on "key" key through proxy.
var valueFromProxyCache = (string)proxyCache[key];

// Receiving the value on "prefix+key" key directly from session level cache.
var valueFromGlobalCache = (string)Store.Cache[CacheLevel.Session][prefix + key];
```

As a result, the *valueFromProxyCache* and *valueFromGlobalCache* variables will contain the same value of "CachedValue".

3. The DataStoreKeyTransformerProxy class

A proxy class that converts repository keys by adding a prefix to them.

Example 6 demonstrates working with the repository via the *PrefixKeyTransformerProxy* proxy class.

Example 6

```
// Key with which value will be placed into the repository through proxy.  
string key = "Key";  
  
// Prefix that will be added to the value key by means of a proxy class.  
string prefix = "customPrefix";  
  
// Creation of proxy class that will be used for recording values into session level  
repository.  
IDataStore proxyStorage = new DataStoreKeyTransformerProxy(prefix) { DataStore =  
Store.Data[DataLevel.Session] };  
  
// Recording a value with the key into the repository through proxy class.  
// This value is recorded into global cache of session level with "prefix+key".  
proxyStorage[key] = "StoredValue";  
  
// Receiving a value for the key through proxy.  
var valueFromProxyStorage = (string)proxyStorage[key];  
  
// Receiving a value on the "prefix+key" key directly from session level cache.  
var valueFromGlobalStorage = (string)Store.Data[DataLevel.Session][prefix + key];
```

As a result, the *valueFromProxyStorage* and *valueFromGlobalStorage* variables will contain the same value of "StoredValue".

Local data caching

The mechanism of local data caching is implemented on the basis of proxy classes in Creatio. The main purpose of data caching is to reduce the load on the repository server and the time it takes to run queries when dealing with data that is rarely changed.

The logic of the local caching mechanism is implemented by the internal *LocalCachingProxy* proxy class. This proxy class caches the data on the current web farm node. The class monitors the lifetime of cached objects and receives data from the global cache only if the cached data is not relevant.

The *ICacheStore* interface from the static *CacheStoreUtilities* class are used to apply the local caching mechanism:

- *WithLocalCaching()* is an overloaded method that returns the *LocalCachingProxy* class instance which executes the local caching.
- *WithLocalCachingOnly(string)* is a method that executes the local caching of data for a set group of elements while monitoring their validity.
- *ExpireGroup(string)* is a method that sets an aging checkbox for a set group of items. All items of the set group became invalid and do not return upon data query upon calling of this method.

Example 7 demonstrates a workspace cache operation with the use of local caching.

Example 7

```
// Creating the first proxy class that executes local data caching. All items  
// recorded in cache through this proxy refer to Group1 validity control group.  
ICacheStore cacheStore1 =  
Store.Cache[CacheLevel.Workspace].WithLocalCaching("Group1");  
  
// Adding the element to cache through proxy class.  
cacheStore1["Key1"] = "Value1";  
  
// Creating the second proxy class that executes local data caching.  
// All items recorded in cache through this proxy also refer to Group1 validity  
control group.  
ICacheStore cacheStore2 =  
Store.Cache[CacheLevel.Workspace].WithLocalCaching("Group1");  
cacheStore2["Key2"] = "Value2";
```

```
// Aging checkbox is set for all items of Gtoup1. Items with Key1 and Key2 are  
considered to be aged  
// since they refer to one validity control group, Group1, regardless of the fact  
that  
// they are added to cache through different proxies.  
cacheStore2.ExpireGroup("Group1");  
  
// The attempt to receive values from cache on Key1 and Key 2 after these items were  
marked as  
// aged. As a result, cachedValue1 and cachedValue 2 variables will contain null  
value.  
var cachedValue1 = cacheStore1["Key1"];  
var cachedValue2 = cacheStore1["Key2"];
```

Specific features of database and cache usage

To improve the efficiency of working with data repositories and cache in the configuration logic and in the business processes implementation logic, consider the following:

1. All objects placed in the repository must be serializable. This requirement is due to the specifics of working with the core of the application. If you save data to the repository (except for data repositories of the *Request* level), the object is pre-serialized, and deserialized when received.
2. Accessing the repository is a relatively resource-intensive operation. The code should avoid unnecessary access to the repository. Examples 8 and 9 provide correct and incorrect versions of the cache.

Example 8

Incorrect:

```
// Network access and data deserialization are executed.  
if (UserConnection.SessionData["SomeKey"] != null)  
{  
    // Network access and data deserialization are executed repeatedly.  
    return (string)UserConnection.SessionData["SomeKey"];  
}
```

Correct (1st option):

```
// Getting the object from the repository to the intermediate variable.  
object value = UserConnection.SessionData["SomeKey"];  
  
// Verification of the intermediate variable value.  
if (value != null)  
{  
    // Value return.  
    return (string)value;  
}
```

Correct (2nd option):

```
// The use of GetValue() extending method.  
return UserConnection.SessionData.GetValue<string>("SomeKey");
```

Example 9

Incorrect:

```
// Network access and data deserialization are executed.  
if (UserConnection.SessionData["SomeKey"] != null)  
{  
    // Network access and data deserialization are executed repeatedly.  
    UserConnection.SessionData.Remove("SomeKey");
```

}

Correct:

```
// Deletion is executed immediately without any preliminary verification.  
UserConnection.SessionData.Remove("SomeKey");
```

3. Any changes made to the state of an object obtained from a repository or cache occur locally in memory and are not automatically stored in the repository. Therefore, for these changes to appear in the repository, the modified object must be explicitly written to it (example 10).

Example 10

```
// Receiving the value dictionary form session data repository on "SomeDictionary" key.  
Dictionary<string, string> dic = (Dictionary<string,  
string>) Store.Data[DataLevel.Session]["SomeDictionary"];  
  
// Change of the dictionary value item. Changes in the repository are not fixed.  
dic["Key"] = "ChangedValue";  
  
// Adding a new item to the dictionary. Changes are not fixed in the dictionary.  
dic.Add("NewKey", "newValue");  
  
// The dictionary is recorded in the data repository on the "SomeDinctionary" key.  
// All changes are now fixed in the repository.  
Store.Data[DataLevel.Session]["SomeDictionary"] = dic;
```

Repositories. Types and recommendations on use

Beginner

Easy

Medium

Advanced

Introduction

In Creatio, business data storage is implemented as a relational database (MS SQL, Oracle or PostgreSQL). The data is used for solutions of different user tasks, execution of business processes, etc. The application ensures execution of the following operations:

1. Storage of user and application data (user profile, session data etc.).
2. Data communication between web farms.
3. Intermediate data storage at the time of restart of application or webfarm.
4. Execution of intermediate actions with data before their placing in repository.

For the purpose of these tasks, data repository technology is implemented in Creatio's architecture. The object model of classes, being a unified API for access from application to data, located in an external repository (currently [Redis](#) is used in Creatio as the external repository) forms the base of this technology.

Creatio repositories focus on execution of service functions or arrangements of data handling but they also can be used for solving user tasks (in configuration business logic).

Redis as a Creatio repository server

Creatio repository server is represented by [Redis](#) – a high-efficient non-relational data repository.

The data model of Redis is based on "key-object" pairs. Redis supports access only with a unique key and can be successfully used for storage of serialized objects. Data, placed into the repository, is stored as binary serialized objects in Creatio.

Redis supports several data storage strategies:

- Storage of data only in memory. Persistent database is converted to caching server.
- Periodical saving of data to disc (by default) – periodical copy saving (snapshot) (once per 1-15 minutes depending on time for creation of previous copy and number of changed keys).

- Transaction log. A synchronous record of each change in special append-only log-file.
- Replication. You can assign a primary to each server. Then all changes in master will be reproduced in the replica.

The definite data storage method is determined by configuring the Redis server. In Creatio (at present), data is stored in memory with periodic saving of data to disc but data replication function is not supported.

For more details on Redis see [official documentation](#).

Creatio repository type. Data storage and cache.

Creatio supports two types of repositories, i.e. data and cache repositories.

A *data repository* is designed for intermediate storage of rarely modified "long-term" data. A *cache* repository stores operation data.

Individual logical levels of data placement are additionally determined for each type of repository (table 1, 2).

Table 1. Data repository levels

Level	Details
<i>Request</i>	The query level. The data of this level is available only in the course of current query processing. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Request</i> enumeration value.
<i>Session</i>	The session level. The data of this level is available only in the session of the current user. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Session</i> enumeration value.
<i>Application</i>	The application level. The data is available for the entire application. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Application</i> enumeration value.

Table 2. Cache levels

Level	Details
<i>Session</i>	The session level. The data of this level is available only in the session of the current user. Corresponds to <i>Terrasoft.Core.Store.DataLevel.Session</i> enumeration value.
<i>Workspace</i>	The level of the workspace. The data of this level is available for all users of one and the same workplace. Corresponds to the <i>Terrasoft.Core.Store.CacheLevel</i> enumeration value.
<i>Application</i>	The application level. The data of this level is available for all application users regardless of their workspace. Corresponds to the <i>Terrasoft.Core.Store.CacheLevel</i> enumeration value.

Such differentiation of repositories is implemented for the purpose of logical separation of units in data repository and convenience of further use of the source code. Additionally, such separation creates solutions for the following tasks:

- Isolation of data between workspaces and user sessions
- Conditional classification of data
- Control of the data life cycle

All repository and cache data can be located physically on an abstract data storage server. The exception is the data of the *Request* level repository that are stored directly in the memory.

At present, the Creatio repository server is Redis. In common cases, it may be a user repository, accessed through unified interfaces. It is necessary to take into account the fact that repository access operations are resource-intensive since they are associated with serialization / deserialization of data and network communication.

The data repository and cache represent the following possibilities for data handling:

- Access to data by key for reading / recoding
- Deletion of data from repository by key

The key difference between a data repository and cache is the control of the object life cycle within them.

Data will be stored in the storage till the explicit deletion. The life cycle of such objects is limited by query execution time (for data of *Request* level repository) or session existence time (for data of *Session* level repository).

There is such a notion as *aging time* for data. It determines time limit of validity of cache items. All items are deleted from cache, regardless of aging time, in the following cases:

- Session ending (items of cache and data repository of *Session* level)
- Implicit deletion of workspace (items of *Workspace* level cache)

The items of the *Application* level cache are stored for the entire period of existence of the application and can be deleted only by clearing the external repository.

Data can be deleted from cache in any period of time. This may result in situations when the code tries to get cached data that has already been deleted from cache at the time of access. In this case, the calling code should only receive data from the persistent repository and place them in cache.

Object model of Creatio repositories

Creatio classes and interfaces for data repository and cache operations

Creatio uses a series of classes and interfaces (located in *Terrassot.Core.Store* name space) to work with the database and cache (“**.NET class libraries of platform core (on-line documentation)**”). The main ones are listed below.

IBaseStore base repository

Base functions of all types of repositories are represented by *IBaseStore*. Properties and methods of this interface are used to implement the following:

- Access to data by key for reading/recording (*this[string key]* indexer)
- Deletion of data from the repository by set key (*Remove(string key)* method)
- Initialization of the repository by set parameter list (*Initialize(IDictionary<string, string> parameters)* method). At present, parameters for initialization of Creatio repositories are read from configuration file. Parameter lists are set in *storeDataAdapter* (for data repository) and *storeCacheAdapter* (for cache) sections. In common cases, parameters can be set randomly.

IDataStore data repository

The interface determines the specifics of data repository operations. It is an inheritor of *IBaseStore* repository base interface. The interface provides an additional possibility for getting the list of all repository keys (the *Keys* property).

We recommend using the *Keys* property upon data repository operations only in exceptional cases when it is impossible to solve tasks by alternative methods.

ICacheStore cache storage

The interface determines the specifics of cache operations. It is an inheritor of *IBaseStore* repository base interface. The *GetValues(IEnumerable keys)* property returns cache object dictionary with set keys. This method optimizes repository operation upon simultaneous receiving of data set. This method enables the optimization of working with the repository while simultaneously receiving a dataset.

The Store class

Static class for access to caches and data repositories of different levels. Levels of data repositories and cache are determined in the *Terrassot.Core.Store.DataLevel* and *CacheLevel* enumerations (table 1 and 2).

The *Store* class has 2 static properties:

- The *Data* property returns an instance of the data repository provider.

- The *Cache* property returns an instance of the cache provider.

Example 1 demonstrates working with the cache and the data store using the *Store* class.

Example 1

```
// Getting a link to the session-level data repository.  
IDataStore dataStore = Store.Data[DataLevel.Session];  
  
// Place the "Data Test Value" value with the "DataKey" key in the data repository.  
dataStore["DataKey"] = "Data Test Value";  
  
// Getting a link to the application-level cache of the workspace.  
ICacheStore cacheStore = Store.Cache[CacheLevel.Workspace];  
  
// Removing an item with the "CacheKey" key from the cache.  
cacheStore.Remove("CacheKey");
```

Access to data repositories and cache from UserConnection

Access to the data repositories and application caches from the configuration code can be obtained using the properties of the static *Store* class of the *Terrasoft.Core.Store* name space. An alternative way to access the data repository and cache in the configuration logic, which avoids the use of long property names and the connection of additional assemblies, is accessed through an instance of the *UserConnection* class.

The *UserConnection* class implements a number of helper properties that enable quick access to data repositories and caches of various levels:

- *ApplicationCache* returns a link to the application-level cache.
- *WorkspaceCache* returns a link to the workspace cache.
- *SessionCache* returns a link to the session-level cache.
- *RequestData* returns a link to the query-level data repository.
- *SessionData* returns a link to the session-level data repository.
- *ApplicationData* returns a link to the application-level data repository.

In most cases, accessing the *UserConnection* properties is identical to accessing the *Store.Cache* and *Store.Data* properties, indicating the appropriate levels. However, in some situations (e.g., when running business processes using a scheduler), another implementation may be used. Therefore, it is recommended to use the properties of the *UserConnection* object in the configuration logic for accessing the repositories.

Example 2

```
// The key with which the value will be placed in the cache.  
string cacheKey = "SomeKey";  
  
// Putting the value into the session-level cache via the UserConnection property.  
UserConnection.SessionCache[cacheKey] = "SomeValue";  
  
// Retrieving a value from the cache via the Store property.  
// As a result, the variable valueFromCache will contain the value "SomeValue".  
string valueFromCache = Store.Cache[CacheLevel.Session][cacheKey] as String;
```

Using cache in EntitySchemaQuery

EntitySchemaQuery implements a mechanism for working with the repository (Creatio cache or user-defined arbitrary repository). Working with the cache allows you to optimize the efficiency of operations by accessing cached query results without additional access to the data repository. When the *EntitySchemaQuery* query is executed, the data received from the database on the server is placed in the cache, which is determined by the *Cache* property with the key that is set by the *CacheItemName* property. By default, the cache of the *EntitySchemaQuery* request is the session-level Creatio cache with local storage. In general, the query cache can be an arbitrary repository that implements the *ICacheStore* interface.

Example 3 shows the process of working with the application cache when executing the *EntitySchemaQuery* request. The code in the example builds a query that returns a list of all cities in the *City* schema. When you receive the results of the query (after calling the *GetEntityCollection()* method), these results are placed in the cache, which can then be used to retrieve the collection of query elements without additional access to the repository.

Example 3

```
// Creating an instance of the EntitySchemaQuery request with the root City schema.
var esqResult = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

// Add the city name column to the query.
esqResult.AddColumn("Name");

// Identifying the key under which the results of query execution will be stored in
// the cache.
// Session level cache with local data caching (since Cache property of the object is
// not determined)
// is used as cache.
esqResult.CacheItemName = "EsqResultItem";

// Execution of query to database for receiving resultant object collection.
// Query results will be placed in cache after completion of this operation. Upon
// further access to
// esqResult for receiving object collection query (if query was not changed) these
// objects will
// be taken from session cache.
esqResult.GetEntityCollection(UserConnection);
```

Repository and cache proxy classes

Proxy class notion and purpose

Repositories and caches can be accessed either directly (through the *Store* class properties) or through proxy classes in Creatio.

Proxy classes are special objects that act as intermediate links between repositories and calling code. Proxy classes allow intermediate operations with data before they are read / edited in the repository. Each proxy class is a repository in itself.

Proxy class applications

1. Initial setup and configuration of the application

You can configure the use of proxy classes for data repositories and caches in the Web.config configuration file. The proxy classes for the corresponding data repository or cache are configured in the *storeDataAdapter* and *storeCacheAdapter* sections, respectively. They add a *proxies* section, which lists all the proxy classes that are applied to the repository (example 4).

Example 4

```
<storeDataAdapters>
    <storeAdapter levelName="Request" type="RequestDataAdapterClassName">
        <proxies>
            <proxy name="RequestDataProxyName1" type="RequestDataProxyClassName1" />
            <proxy name="RequestDataProxyName2" type="RequestDataProxyClassName2" />
            <proxy name="RequestDataProxyName3" type="RequestDataProxyClassName3" />
        </proxies>
    </storeAdapter>
</storeDataAdapters>

<storeCacheAdapters>
```

```
<storeAdapter levelName="Session" type="SessionCacheAdapterClassName">
    <proxies>
        <proxy name="SessionCacheProxyName1" type="SessionCacheProxyClassName1"
    />
        <proxy name="SessionCacheProxyName2" type="SessionCacheProxyClassName2"
    />
    </proxies>
</storeAdapter>
</storeCacheAdapters>
```

When the application is downloaded, the settings are read from the configuration file and applied to the corresponding repository type. This enables you to build chains of proxy classes that will be executed sequentially one after another. The order of the proxy classes in the execution chain corresponds to their order in the configuration file. In this case, the proxy class listed last in the *proxies* section is the first in the chain of execution, i.e. the execution is done "from the bottom up".

Take into account the following:

- The "final point" of the application of proxy classes is a definite cache or data repository, for which this chain is determined.
- A separate proxy class knows the cache or repository it works with. The link to it is determined by *ICacheStoreProxy.CacheStore* property or *IDataStoreProxy.DataStore* property. A proxy class doesn't know what exactly this property refers to, i.e. other proxy class, final repository or cache. At the same time, this proxy can act as repository, with which other proxy classes can work.

Thus, according to the settings shown in Example 4, the chain of execution of proxy classes (e.g., data repository) will be as follows: *RequestDataProxyName3* > *RequestDataProxyName2* > *RequestDataProxyName1* > *RequestDataAdapterClassName* (final query level data repository).

2. Delimiting data of different application users

Isolating data between different application users is done with the help of proxy classes. The simplest solution to this problem is to transform the value keys before placing them in the repository (e.g., by adding an additional prefix to the key that is specific to the user). Using such proxy classes ensures the uniqueness of storage keys. This, in turn, helps to avoid data loss and corruption while different users simultaneously try to write different values in the repository with the same key. An example of working with key transformation proxy classes is shown below. Generally speaking, you can implement as complex logic as you like in proxy classes.

3. Performing other intermediate actions with the data before placing it in the repository.

In proxy classes, you can implement the logic of performing any arbitrary actions with data before placing or getting it from the repository. Putting the logic of processing data into a proxy class helps avoid code duplicates, which in turn facilitates its maintenance.

Basic proxy class interfaces

A class must implement one (or both) of the *Terrasoft.Core.Store* namespace interfaces in order to be used as a proxy for working with repositories:

- *IDataStoreProxy* is an interface for proxy classes of the data storage
- *ICacheStoreProxy* is the interface of cache proxy classes.

Each of these interfaces has one property - a reference to that storage (or cache) with which the given proxy class is working. For the *IDataStoreProxy* interface, it is the *DataStore* property. For the *ICacheStoreProxy* interface, it is the *CacheStore* property.

Key transformation proxy classes

In Creatio, there is a number of proxy classes that implement the logic of transformation of value keys placed in the repository.

1. The KeyTransformerProxy class

This is an abstract class which acts as a base class for all proxy classes that convert cache keys. It implements the methods and properties of the *ICacheStoreProxy* interface. When creating custom proxy classes for key transformation, it is recommended to inherit from this class to avoid logic duplicates.

2. The PrefixKeyTransformerProxy

A proxy class that converts cache keys by adding a prefix to them.

Example 5 demonstrates working with the session-level cache via the *PrefixKeyTransformerProxy* proxy class.

Example 5

```
// Key, with which a value will be placed in cache through proxy.
string key = "Key";

// Prefix that will be added to value key by proxy class.
string prefix = "customPrefix";

// Creation of proxy class that will be used for recording of values into session
level cache.
ICacheStore proxyCache = new PrefixKeyTransformerProxy(prefix,
Store.Cache[CacheLevel.Session]);

// Recording value with the "key" key in cache through proxy class.
// This value is recorded into global cache of session level with "prefix+key" key.
proxyCache[key] = "CachedValue";

// Receiving the value on "key" key through proxy.
var valueFromProxyCache = (string)proxyCache[key];

// Receiving the value on "prefix+key" key directly from session level cache.
var valueFromGlobalCache = (string)Store.Cache[CacheLevel.Session][prefix + key];
```

As a result, the *valueFromProxyCache* and *valueFromGlobalCache* variables will contain the same value of "CachedValue".

3. The DataStoreKeyTransformerProxy class

A proxy class that converts repository keys by adding a prefix to them.

Example 6 demonstrates working with the repository via the *PrefixKeyTransformerProxy* proxy class.

Example 6

```
// Key with which value will be placed into the repository through proxy.
string key = "Key";

// Prefix that will be added to the value key by means of a proxy class.
string prefix = "customPrefix";

// Creation of proxy class that will be used for recording values into session level
repository.
IDataStore proxyStorage = new DataStoreKeyTransformerProxy(prefix) { DataStore =
Store.Data[DataLevel.Session] };

// Recording a value with the key into the repository through proxy class.
// This value is recorded into global cache of session level with "prefix+key".
proxyStorage[key] = "StoredValue";

// Receiving a value for the key through proxy.
var valueFromProxyStorage = (string)proxyStorage[key];
```

```
// Receiving a value on the "prefix+key" key directly from session level cache.  
var valueFromGlobalStorage = (string)Store.Data[DataLevel.Session][prefix + key];
```

As a result, the *valueFromProxyStorage* and *valueFromGlobalStorage* variables will contain the same value of "StoredValue".

Local data caching

The mechanism of local data caching is implemented on the basis of proxy classes in Creatio. The main purpose of data caching is to reduce the load on the repository server and the time it takes to run queries when dealing with data that is rarely changed.

The logic of the local caching mechanism is implemented by the internal *LocalCachingProxy* proxy class. This proxy class caches the data on the current web farm node. The class monitors the lifetime of cached objects and receives data from the global cache only if the cached data is not relevant.

The *ICacheStore* interface from the static *CacheStoreUtilities* class are used to apply the local caching mechanism:

- *WithLocalCaching()* is an overloaded method that returns the *LocalCachingProxy* class instance which executes the local caching.
- *WithLocalCachingOnly(string)* is a method that executes the local caching of data for a set group of elements while monitoring their validity.
- *ExpireGroup(string)* is a method that sets an aging checkbox for a set group of items. All items of the set group became invalid and do not return upon data query upon calling of this method.

Example 7 demonstrates a workspace cache operation with the use of local caching.

Example 7

```
// Creating the first proxy class that executes local data caching. All items  
// recorded in cache through this proxy refer to Group1 validity control group.  
ICacheStore cacheStore1 =  
Store.Cache[CacheLevel.Workspace].WithLocalCaching("Group1");  
  
// Adding the element to cache through proxy class.  
cacheStore1["Key1"] = "Value1";  
  
// Creating the second proxy class that executes local data caching.  
// All items recorded in cache through this proxy also refer to Group1 validity  
// control group.  
ICacheStore cacheStore2 =  
Store.Cache[CacheLevel.Workspace].WithLocalCaching("Group1");  
cacheStore2["Key2"] = "Value2";  
  
// Aging checkbox is set for all items of Gtoupl. Items with Key1 and Key2 are  
// considered to be aged  
// since they refer to one validity control group, Group1, regardless of the fact  
// that  
// they are added to cache through different proxies.  
cacheStore2.ExpireGroup("Group1");  
  
// The attempt to receive values from cache on Key1 and Key 2 after these items were  
// marked as  
// aged. As a result, cachedValue1 and cachedValue 2 variables will contain null  
// value.  
var cachedValue1 = cacheStore1["Key1"];  
var cachedValue2 = cacheStore1["Key2"];
```

Specific features of database and cache usage

To improve the efficiency of working with data repositories and cache in the configuration logic and in the business processes implementation logic, consider the following:

1. All objects placed in the repository must be serializable. This requirement is due to the specifics of working with the core of the application. If you save data to the repository (except for data repositories of the *Request* level), the object is pre-serialized, and deserialized when received.

2. Accessing the repository is a relatively resource-intensive operation. The code should avoid unnecessary access to the repository. Examples 8 and 9 provide correct and incorrect versions of the cache.

Example 8

Incorrect:

```
// Network access and data deserialization are executed.  
if (UserConnection.SessionData["SomeKey"] != null)  
{  
    // Network access and data deserialization are executed repeatedly.  
    return (string)UserConnection.SessionData["SomeKey"];  
}
```

Correct (1st option):

```
// Getting the object from the repository to the intermediate variable.  
object value = UserConnection.SessionData["SomeKey"];  
  
// Verification of the intermediate variable value.  
if (value != null)  
{  
    // Value return.  
    return (string)value;  
}
```

Correct (2nd option):

```
// The use of GetValue() extending method.  
return UserConnection.SessionData.GetValue<string>("SomeKey");
```

Example 9

Incorrect:

```
// Network access and data deserialization are executed.  
if (UserConnection.SessionData["SomeKey"] != null)  
{  
    // Network access and data deserialization are executed repeatedly.  
    UserConnection.SessionData.Remove("SomeKey");  
}
```

Correct:

```
// Deletion is executed immediately without any preliminary verification.  
UserConnection.SessionData.Remove("SomeKey");
```

3. Any changes made to the state of an object obtained from a repository or cache occur locally in memory and are not automatically stored in the repository. Therefore, for these changes to appear in the repository, the modified object must be explicitly written to it (example 10).

Example 10

```
// Receiving the value dictionary form session data repository on "SomeDictionary"  
key.  
Dictionary<string, string> dic = (Dictionary<string,  
string>) Store.Data[DataLevel.Session]["SomeDictionary"];  
  
// Change of the dictionary value item. Changes in the repository are not fixed.
```

```
dic["Key"] = "ChangedValue";

// Adding a new item to the dictionary. Changes are not fixed in the dictionary.
dic.Add("NewKey", "newValue");

// The dictionary is recorded in the data repository on the "SomeDictionary" key.
// All changes are now fixed in the repository.
Store.Data[DataLevel.Session]["SomeDictionary"] = dic;
```

Configuration web-services

Contents

- **Creating a user configuration service**
- **Creating anonymous web service**
- **Calling configuration services with ServiceHelper**

Creating a user configuration service

Beginner

Easy

Medium

Advanced

Introduction

Creatio service model implements the base set of web services, which you can use for integration with third-party applications and systems. Examples of these system web services are:

- **EntityDataService.svc** – exchanging data with Creatio via the [OData 3](#) protocol.
- **odata** – exchanging data with Creatio via the [OData 4](#) protocol.
- **ProcessEngineService.svc** – launch Creatio business processes from external applications.

These services are implemented based on the [WCF](#) technology and are managed at IIS level.

There are also configuration web services in Creatio that can be called from the client part of the application. You can implement specific integration tasks via configuration web services.

Configuration web service is a [RESTful service](#) developed on the [WCF](#) technology. The web service is available at following address:

```
[Application Address]/0/rest/[Custom Service Name]/[Custom Service Endpoint]?
[Optional Options]
```

For example:

```
http://mysite.creatio.com/0/rest/UsrCustomConfigurationService/GetContactIdByName?
Name=User1
```

Custom configuration service becomes available after user authentication via the AuthService.svc (see [“Authentication of external requests”](#)).

Starting from version 7.14.1, we have changed the approach to developing web services. The service class must be inheritor of *Terrasoft.Web.Common.BaseService*.

The *UserConnection* and *AppConnection*, properties are already defined in *Terrasoft.Web.Common.BaseService*, so you do not have to receive these objects from *HttpContext.Current*. The latter also defines the *HttpContextAccessor* property, which provides access to context.

HttpContextAccessor grants unified access to *HttpContext* in both frameworks: *ASP.NET Framework* and *ASP.Net Core*. You can receive context in two ways:

- (Not recommended) Using the *HttpContext.Current* static property. To make the migration from *ASP.NET Framework* to *ASP.Net Core* easier, add the *Terrasoft.Web.Http.Abstractions* namespace to the source code of your service (apply the *using* directive). This namespace grants unified access to *HttpContext* that has been implemented using the *HttpContext.Current* static property. When adapting

the old code to the new framework, replace the *System.Web* namespace by *Terrasoft.Web.Http.Abstractions*

- (Recommended) Using *IHttpContextAccessor* registered in *DI (ClassFactory)*. Enables implementing test coverage of the code.

You cannot use specific implementation of access to context from *ASP.NET* (the *System.Web* library) or *ASP.NET Core* (the *Microsoft.AspNetCore.Http* library) in the configuration.

Example of using the *Terrasoft.Web.Common.BaseService* parent class properties.

```
namespace Terrasoft.Configuration.UsrCustomNamespace
{
    using Terrasoft.Web.Common;

    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class UsrCustomConfigurationService: BaseService
    {
        // Service method.
        [OperationContract]
        [WebInvoke(Method = "GET", RequestFormat = WebMessageFormat.Json, BodyStyle =
WebMessageBodyStyle.Wrapped,
            ResponseFormat = WebMessageFormat.Json)]
        public void SomeMethod() {
            ...
            var currentUser = UserConnection.CurrentUser; // UserConnection - the
BaseService property.
            var sdkHelpUrl = AppConnection.SdkHelpUrl; // AppConnection - the
BaseService property.
            var httpContext = HttpContextAccessor.GetInstance(); // 
HttpContextAccessor - the BaseService property.
            ...
        }
    }
}
```

Example of adapting to the ASP.Net Core service

```
namespace Terrasoft.Configuration.UsrCustomNamespace
{
    using Terrasoft.Web.Http.Abstractions; // Use instead of System.Web

    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class UsrCustomConfigurationService
    {
        // Service method.
        [OperationContract]
        [WebInvoke(Method = "GET", RequestFormat = WebMessageFormat.Json, BodyStyle =
WebMessageBodyStyle.Wrapped,
            ResponseFormat = WebMessageFormat.Json)]
        public void SomeMethod() {
            ...
            var httpContext = HttpContext.Current;
            ...
        }
    }
}
```

To create a custom web service in the configuration:

1. Create a schema of the [Source code] type in the development package.
2. Create a class of the service in the schema source code. Use the namespace in the *Terrasoft.Configuration* or any namespace embedded in it. Mark the class with the [\[ServiceContract\]](#) and [\[AspNetCompatibilityRequirements\]](#) attributes with necessary parameters. The service class must be inheritor of *Terrasoft.Web.Common.BaseService*.
3. Add the implementation of methods corresponding to the [service endpoints](#) in the class. Each method of the service should be marked with the [\[OperationContract\]](#) and [\[WebInvoke\]](#) attributes with necessary parameters. If you need to send the data of complex type (object instances, collections, arrays, etc.) you can implement additional classes which instances will receive and return the method of your service. Each class of that type should be marked with the [\[DataContract\]](#) attribute and the fields of the class should be marked with the [\[DataMember\]](#) attribute.
4. Publish the source code schema.

After you publish the schema, the created web service will become available for calling from the source code of the configuration schemas as well as from the external applications (see “**Calling configuration services with ServiceHelper**”).

See also:

- [Creating a user configuration service](#)

Creating anonymous web service

Beginner

Easy

Medium

Advanced

Introduction

Creatio service model implements the base set of web services, which you can use for integration with third-party applications and systems.

Examples of system services:

- **EntityDataService.svc** – data exchange with Creatio via the [OData3](#) protocol.
- **odata** – data exchange with Creatio via the [OData4](#) protocol.
- **ProcessEngineService.svc** – running Creatio business processes using third-party applications.

These services are implemented using [WCF](#) technology and are managed at the IIS level.

Creatio also provides configuration web services designed for calls from the client part of the application. You can implement specific integration tasks using Creatio configuration web services. Learn more about creating user configuration services in the “**Creating a user configuration service**” article.

Most of WCF-services require preliminary user authentication. Some of the services, such as AuthService.svc, permit anonymous usage.

Since configuration services are designed to be managed at the application level rather than the IIS level, you cannot make them anonymous.

To create a WCF-service that will be accessible without user authentication:

1. Create a configuration web service (optional).
2. Register the WCF-service.
3. Configure the WCF-service for the HTTP and HTTPS protocols.
4. Setting up access to the WCF-service for all users.

Creating a configuration service

Learn more about creating configuration services in the “**Creating a user configuration service**” article.

Use *SystemUserConnection* instead of the system connection of a user when creating an anonymous configuration service,

To maintain the availability of business processes when working with *Entity* from an anonymous web service, call

the `SessionHelper.SpecifyWebOperationIdentity` method of the [Terrasoft.Web.Common](#) namespace after retrieving `SystemUserConnection`:

```
Terrasoft.Web.Common.SessionHelper.SpecifyWebOperationIdentity(HttpContextAccessor.GetTInstance(), SystemUserConnection.CurrentUser);
```

This method enables you to specify the user that will be running the HTTP query.

Registering the WCF-service

To register a WCF-service in the `..|Terrasoft.WebApp|ServiceModel` directory, create an `.svc` file and add the following entry:

```
<% @ServiceHost
Service = "Service, ServiceNamespace"
Factory = "Factory, FactoryNamespace"
Debug = "Debug"
Language = "Language"
CodeBehind = "CodeBehind"
%>
```

Specify the full name of the configuration service class in the `Service` attribute. Read more about the `@ServiceHost` WCF directive in [Microsoft documentation](#).

Configuring the WCF-service for the HTTP and HTTPS protocols.

Append the following record to the `services.config` files located at `..|Terrasoft.WebApp|ServiceModel|http` and `..|Terrasoft.WebApp|ServiceModel|https` directories:

```
<services>
  ...
  <service name="Terrasoft.Configuration.[Service name]">
    <endpoint name="[Service name]EndPoint"
      address=""
      binding="webHttpBinding"
      behaviorConfiguration="RestServiceBehavior"
      bindingNamespace="http://Terrasoft.WebApp.ServiceModel"
      contract="Terrasoft.Configuration.[Service name]" />
  </service>
</services>
```

Configure the service here. The `<services>` element must contain a list of configurations of all application services (nested `<service>` elements). The `name` attribute must contain the name of the type (class or interface) that implements the service contract. The `<endpoint>` child element requires an address, binding, and interface, which define the service contract specified in the `name` attribute of the `<service>` element.

You can find a detailed description of the service configuration elements in the [Microsoft documentation](#).

Setting up access to the WCF-service for all users.

To set up access to the WCF-service for all users, make the following adjustments to the `..|Terrasoft.WebApp|Web.config` file:

- Add a `<location>` element defining the relative path and service access permissions
- In the `<appSettings>` element, change the `value` attribute for the “`AllowedLocations`” key by supplying the relative path of the service.

You can find sample changes in the `..|Terrasoft.WebApp|Web.config` file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  ...
  <location path="ServiceModel/[Service name].svc">
    <system.web>
```

```
<authorization>
  <allow users="*" />
</authorization>
</system.web>
</location>
...
<appSettings>
  ...
  <add key="AllowedLocations" value="[Previous values];ServiceModel/[Service
name].svc" />
  ...
</appSettings>
...
</configuration>
```

After reloading the application pool in IIS, the service will become available at:

[Application Address]/0/ServiceModel/[Custom Service Name].svc/[Custom Service Endpoint]?[Optional parameters]

You can access the service from a browser, with or without preliminary login.

You need to change the application configuration files to set up anonymous web-service. When updating the application, all the configuration files are changed by the new ones. Thus, you need to set up the web service again after the application update.

See also

- [Creating anonymous web service](#)

Calling configuration services with ServiceHelper

Beginner

Easy

Medium

Advanced

Introduction

To call a configuration web service from the client JavaScript-code:

1. Add the *ServiceHelper* module as a dependency to the module of the page that was used for calling the service. This module is an interface for executing server queries via the *Terrasoft.AjaxProvider* query provider implemented in the client core.
2. Call the *callService(serviceName, serviceMethodName, callback, serviceData, scope)* method from the *ServiceHelper* module by passing the parameters as listed in table 1.

Table 1. The *callService()* method parameters

Parameter	Details
<i>serviceName</i>	Configuration service name. Corresponds to the name of the C# class that implements the service.
<i>serviceMethodName</i>	Name of the configuration service method being called. The method can accept incoming parameters and return resulting values.
<i>callback(response)</i>	The callback function that processes the service response. The function accepts the <i>response</i> object as a parameter. If the called service method returns any value, you can receive it on a client via the <i>response</i> object property. The name of the property that returns the method output-value is generated according to the following rule: [Service method name] + [Result] For example, if you call the <i>GetSomeValue()</i> method, the returned value will be contained in the <i>response.GetSomeValueResult</i> property.
<i>serviceData</i>	The object with the initialized incoming parameters for the service method.

scope	Execution context.
-------	--------------------

There is an alternative way of calling the `callService(config)` method, where `config` is a configuration object with the following properties:

- `serviceName` – configuration service name
- `methodName` – name of the configuration service method being called
- `callback` – the callback function processing the service response
- `data` – the object with the initialized incoming parameters for the service method
- `scope` – execution context

The `ServiceHelper` module only works with the POST requests. Therefore, the **configuration service** methods must be marked by the [\[WebInvoke\] method](#) with the `Method = "POST"` parameter.

See also:

- **Calling configuration services with ServiceHelper**

Working with database

Contents

- **Introduction**
- **Root schema. Building paths to columns**
- **Multi-row data insert. The Insert class**
- **Multithreading when working with the database. Using the DBExecutor**

Database access is provided by a group of server core classes, listed below. Use this class group to perform all basic CRUD operations, account for the access permissions of the current user, and put the retrieved data to the cache storage.

Select

The `Terrasoft.Core.DB.Select` class is used to build queries for the selection of records from the database tables. As a result of creating and configuring the instance of this class, the SELECT SQL-expression query to the application database will be built. You can add the needed icons, filtering, and restriction conditions to the query. The query results are returned as a class instance that implements the `System.Data.IDataReader` interface or as a scalar value of the needed type.

When working with the `Select` class, the current user permissions are not taken into consideration. All records from the application database are available. The data located in the cache repository are not taken into consideration (see the “**Repositories. Types and recommendations on use**” article). Use the `EntitySchemaQuery` class to access additional permission control options and Creatio cache repository operation.

EntitySchemaQuery

The `Terrasoft.Core.Entities.EntitySchemaQuery` class is used to build queries for selecting records in Creatio database tables. As a result of creating and configuring the instance of this class, the SELECT SQL-expression query to the application database will be built. You can add the needed icons, filtering, and restriction conditions to the query.

`EntitySchemaQuery` implements a mechanism for working with the repository (Creatio cache or user-defined arbitrary repository). When `EntitySchemaQuery` is performed, the data retrieved from the server database are cached. The query cache can be an arbitrary repository that implements the `Terrasoft.Core.Store.ICacheStore` interface. By default, the cache of the `EntitySchemaQuery` query is the session-level Creatio cache (only the data from the current user session are available) with local storage.

For the `EntitySchemaQuery` queries, you can determine additional settings that specify the parameters for the page-by-page output of query results and parameters for building a hierarchical query. The `Terrasoft.Core.Entities.EntitySchemaQueryOptions` class is designed for this purpose.

The result of *EntitySchemaQuery* is the *Terrasoft.Nui.ServiceModel.DataContract.EntityCollection* instance (the *Terrasoft.Core.Entities.Entity* class instance collection). Each *Entity* collection instance represents a string of data returned by the query.

EntitySchemaQuery specifics

1) Support of access permissions

The *EntitySchemaQuery* data selection query is built taking into consideration the current user's permissions. The resulting set will only include the data within the range of the current user permissions. Additionally for *EntitySchemaQuery*, you can manage the conditions for granting permissions to the connected tables available in the query (connected to the query by the JOIN clause). These conditions are defined by the *JoinRightState* property value of the *EntitySchemaQuery* instance.

2) Cache mechanism

EntitySchemaQuery implements a mechanism for working with the repository (Creatio cache or user-defined arbitrary repository). When *EntitySchemaQuery* is performed, the data retrieved from the server database are cached. The query cache can be an arbitrary repository that implements the *ICacheStore* interface. By default, the cache of the *EntitySchemaQuery* query is the session-level Creatio cache (only the data from the current user session are available) with local storage. The query cache is defined by the *Cache* property of the *EntitySchemaQuery* instance. The cache access permission key is set via the *CacheItemName* property (example 4).

3) Additional settings of the query

For the *EntitySchemaQuery* queries, you can determine additional settings that specify the parameters for the page-by-page output of query results and parameters for building a hierarchical query. For this, use the *EntitySchemaQueryOptions* class.

The *EntitySchemaQueryOptions* class properties:

- *HierarchicalColumnName* – name of the column used for building the hierarchical request
- *HierarchicalColumnValue* – initial value of the hierarchical column that will be used to build the hierarchy
- *HierarchicalMaxDepth* – maximum level of nesting the hierarchical request
- *PageableConditionValues* – values of the page-by-page output conditions
- *PageableDirection* – direction of the page-by-page output
- *PageableRowCount* – number of page records of the resulting data set returned by the query

The same *EntitySchemaQueryOptions* instance can be used for receiving results of executing different queries by passing it as a parameter to the *GetEntityCollection()* method of the corresponding query (*EntitySchemaQuery* examples).

Insert

The *Terrasoft.Core.DB.Insert* class is used to build queries for adding records in Creatio database tables. As a result of creating and configuring the instance of this class, the INSERT SQL-expression query to the application database will be built. Execution of the query results in returning the number of involved records.

InsertSelect

The *Terrasoft.Core.DB.InsertSelect* class is used to build queries for adding records in Creatio database tables. The *Terrasoft.Core.DB.Select* class instance is used as the source for adding data (see “**Retrieving information from database. The Select class (on-line documentation)**”). As a result of creating and configuring the instance of *Terrasoft.Core.DB.InsertSelect*, the INSERT INTO SELECT SQL-expression query to the application database will be built.

When working with the *InsertSelect* class, the access permissions are not applied to the added records. No application permissions are applied to such records (including object permissions by operation, records, or columns). The user connection is only used for accessing the database table.

After the *InsertSelect* query is executed, the database will be complemented with all records returned in its *Select* subquery.

Update

The *Terrasoft.Core.DB.Update* class is used to build queries for modifying records in Creatio database tables. As a result of creating and configuring the instance of this class, the UPDATE SQL-expression query to the application database will be built.

Delete

The *Terrasoft.Core.DB.Delete* class is used to build queries for deleting records in Creatio database tables. As a result of creating and configuring the instance of this class, the DELETE SQL-expression query to the application database will be built.

Entity

The *Terrasoft.Core.Entities.Entity* class is designed to provide access to an object that represents a record in the database table. This class can also be used to insert, update, and delete specified records.

See also:

- **Root schema. Building paths to columns**
- **Multi-row data insert. The Insert class**
- **Multithreading when working with the database. Using the DBExecutor**

Root schema. Building paths to columns

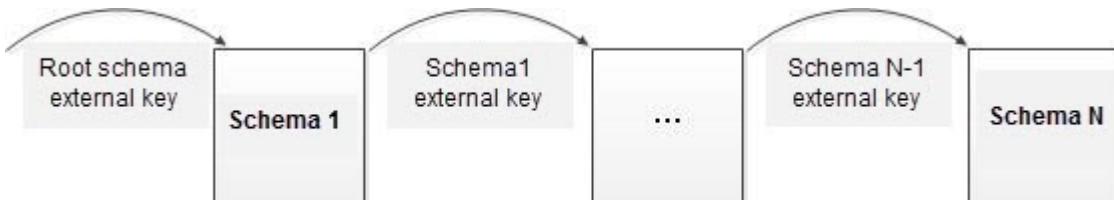
Beginner Easy Medium **Advanced**

Introduction

The root schema is a schema (i.e. a database table) in relation to which the paths are built to all query columns, including the columns of the connected tables.

When building the column paths, the connections are established via the lookup fields. You can build an arbitrary name of a column added to the query as a chain of interconnected items, each of which represents a “context” of a specific schema linked to the previous schema via an external key (fig. 1).

Fig. 1. Schemas interconnected via external keys



In general cases, the format of building an arbitrary column from schema N can be represented as follows:

[Schema 1 context].[...].[Schema N context].[Column_name]

Examples of forming a column name for adding a column to query

The [City] schema acts as a root schema for all the below examples.

Example 1

In this case, the column name is specified as **[Column name in the root schema]**.

- Column containing the city name
- Column name: Name
- Example of creating *EntitySchemaQuery* returning the values of this column:

```
// Creating an instance of the EntitySchemaQuery request with the "City" root schema.  
var esqQuery = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");
```

```
// Adding a column with the city name to the request.  
esqQuery.AddColumn("Name");  
  
// Receiving the text of the resulting sql-request.  
string esqSqlText = esqQuery.GetSelectQuery(UserConnection).GetSqlText();
```

- Resulting sql-query(MS SQL):

```
SELECT  
[City].[Name] [Name]  
FROM  
[dbo].[City] [City]
```

Example 2

The column name is configured as follows: **[Name of lookup column].[Name of lookup schema column]**.

In the resulting query, a “Country” lookup schema will be connected to the City root schema by the JOIN operator (LEFT OUTER JOIN by default). The connection condition (ON condition of the JOIN operator) is formed as follows:

[Name of the connected schema].[Id] = [Name of the root schema].[Name of the lookup column that refers to the connected schema + Id]

- Column containing the name of the country, where the city is located
- Column name: Country.Name
- Example of creating *EntitySchemaQuery* returning the values of this column:

```
// Creating an instance of the EntitySchemaQuery request with the "City" root schema.  
var esqQuery = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");  
  
// Adding a column with the country name to the request.  
esqQuery.AddColumn("Country.Name");  
  
// Receiving the text of the resulting sql-request.  
string esqSqlText = esqQuery.GetSelectQuery(UserConnection).GetSqlText();
```

- Resulting sql-query(MS SQL):

```
SELECT  
[Country].[Name] [Country.Name]  
FROM  
[dbo].[City] [City]  
LEFT OUTER JOIN [dbo].[Country] [Country] ON ([Country].[Id] = [City].[CountryId])
```

- Name of the contact who added the country of a specific city
- Column name: Country.CreatedBy.Name
- Example of creating *EntitySchemaQuery* returning the values of this column:

```
// Creating an instance of the EntitySchemaQuery request with the "City" root schema.  
var esqQuery = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");  
  
// Adding a column with the name of the contact who added the country of a specific  
// city.  
esqQuery.AddColumn("Country.CreatedBy.Name");  
  
// Receiving the text of the resulting sql-request.  
string esqSqlText = esqQuery.GetSelectQuery(UserConnection).GetSqlText();
```

- Resulting sql-query(MS SQL):

```
SELECT  
[CreatedBy].[Name] [CreatedBy.Name]
```

```

FROM
[dbo].[City] [City]
LEFT OUTER JOIN [dbo].[Country] [Country] ON ([Country].[Id] = [City].[CountryId])
LEFT OUTER JOIN [dbo].[Contact] [CreatedBy] ON ([CreatedBy].[Id] =
[Country].[CreatedBy])

```

Example 3

The column name is built as follows: **[Name_of_the_connected_schema:Column_name_for_linking_the_connected_schema:Name_of_the_column_for_linking_the_current_schema]**.

This method of connecting columns suggests applying reverse connections, i.e., the connection of the connected entity lookup column with any of the columns of the primary entity.

If the Id column of the current schema acts as a connecting column, it can be omitted, i.e., the column name will look as follows:

[Name_of_the_connected_schema:Name_of_the_column_for_linking_the_connected_schema].

- Column with the name of the contact, whose page contains the city selected by the query
- Column name: [Contact:City:Id].Name
- Example of creating *EntitySchemaQuery* returning the values of this column:

```

// Creating an instance of the EntitySchemaQuery request with the "City" root schema.
var esqQuery = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

// Adding a column with the name of the contact, whose card contains the specified
// city.
esqQuery.AddColumn("[Contact:City:Id].Name");

// Receiving the text of the resulting sql-request.
string esqSqlText = esqQuery.GetSelectQuery(UserConnection).GetSqlText();

```

- Resulting sql-query(MS SQL):

```

SELECT
[Contact].[Name] [Contact.Name]
FROM
[dbo].[City] [City]
LEFT OUTER JOIN [dbo].[Contact] [Contact] ON ([Contact].[CityId] = [City].[Id])

```

Pay your attention that when you build column names with applying reverse connections, the resulting set of records may contain much more records than the table of the primary entity. In the above example, the database might contain dozens of cities and thousands of contacts, whose pages contain one of the read cities.

In the below example, we consider an alternative variant of building a column name containing the name of the contact that added the country of a specific city (see example 2).

- Column with the name of the contact that added the country of a specific city
- Column name: Country.[Contact:Id:CreatedBy].Name
- Example of creating *EntitySchemaQuery* returning the values of this column:

```

// Creating an instance of the EntitySchemaQuery request with the "City" root schema.
var esqQuery = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");

// Adding a column with the name of the contact who added the country of a specific
// city..
esqQuery.AddColumn("Country.[Contact:Id:CreatedBy].Name");

// Receiving the text of the resulting sql-request.
string esqSqlText = esqQuery.GetSelectQuery(UserConnection).GetSqlText();

```

- Resulting sql-query(MS SQL):

```

SELECT

```

```
[Contact].[Name] [Contact.Name]
FROM
[dbo].[City] [City]
LEFT OUTER JOIN [dbo].[Country] [Country] ON ([Country].[Id] = [City].[CountryId])
LEFT OUTER JOIN [dbo].[Contact] [Contact] ON ([Contact].[Id] =
[Country].[CreatedBy])
```

Multi-row data insert. The Insert class

[Beginner](#)
[Easy](#)
[Medium](#)
[Advanced](#)

Introduction

Starting with version 7.12.4, application supports multi-row data insert. It is available on the *Insert* class level and is determined by the *Values()* method.

After calling the *Values()* method, all subsequent *Set()* calls fall into a new *ColumnsValues* instance. A query with several *Values()* blocks will be built if there is more than one data set in *ColumnsValueCollection*.

Example:

```
new Insert(UserConnection)
    .Into("Table")
    .Values()
        .Set("Column1", Column.Parameter(1))
        .Set("Column2", Column.Parameter(1))
        .Set("Column3", Column.Parameter(1))
    .Values()
        .Set("Column1", Column.Parameter(2))
        .Set("Column2", Column.Parameter(2))
        .Set("Column3", Column.Parameter(2))
    .Values()
        .Set("Column1", Column.Parameter(3))
        .Set("Column2", Column.Parameter(3))
        .Set("Column3", Column.Parameter(3))
.Execute();
```

As the result, the following SQL inquiry is generated:

```
--For MSSQL or PostgreSQL
INSERT INTO [dbo].[Table] (Column1, Column2, Column3)
VALUES (1, 1, 1),
       (2, 2, 2),
       (3, 3, 3)

-- For Oracle
INSERT ALL
    into Table (column1, column2, column3) values (1, 1, 1)
    into Table (column1, column2, column3) values (2, 2, 2)
    into Table (column1, column2, column3) values (3, 3, 3)
SELECT * FROM dual
```

Use specifics

1. MS SQL limits the quantity of parameters to 2100 while using *Column.Parameter* in the *Set()* expression.
2. The *Insert* class cannot independently split a single query into multiple queries if it contains more parameters than necessary. Queries are split by developers themselves. Example:

```
IEnumerable<IEnumerable<ImportEntity>>
GetImportEntitiesChunks(IEnumerable<ImportEntity> entities,
                       IEnumerable<ImportColumn> keyColumns) {
    var entitiesList = entities.ToList();
```

```

    var columnsList = keyColumns.ToList();
    var maxParamsPerChunk = Math.Abs(MaxParametersCountPerQueryChunk /
columnsList.Count + 1);
    var chunksCount = (int)Math.Ceiling(entitiesList.Count /
(double)maxParamsPerChunk);
    return entitiesList.SplitOnParts(chunksCount);
}

var entitiesList = GetImportEntitiesChunks(entities, importColumns);
entitiesList.AsParallel().AsOrdered()
    .ForAll(entitiesBatch => {
        try {
            var insertQuery = GetBufferedImportEntityInsertQuery();
            foreach (var importEntity in entitiesBatch) {
                insertQuery.Values();
                SetBufferedImportEntityInsertColumnValues(importEntity, insertQuery,
                    importColumns);
                insertQuery.Set("ImportSessionId",
Column.Parameter(importSessionId));
            }
            insertQuery.Execute();
        } catch (Exception e) {
            //...
        }
    });
});

```

3. The *Insert()* class does not validate that the number of columns and *Set()* conditions match. For example, if there is a resulting SQL-query:

```
INSERT INTO [dbo].[Table] (Column1, Column2, Column3)
Values (1, 2), (1, 2, 3)
```

This creates an exception on the database level. The *Insert* class is not responsible for a detailed validation. Thus, it only depends on developers.

Multithreading when working with the database. Using the DBExecutor

Beginner

Easy

Medium

Advanced

Introduction

Using a number of streams in working with the database via *UserConnection* may cause to issues in starting synchronization or committing transactions.

The issue occurs during the work with the database even without direct using of the *DBExecutor*. For example, it can be used indirectly, through the *EntitySchemaQuery*.

As unmanaged resources are used to work with the database, you need to wrap the *DBExecutor* in the *using* operator. Another way is to call the *Dispose()* method explicitly to release resources. More information about the *using* operator can be found in the "[C# Guide](#)" documentation.

An example of misuse

A source code fragment with incorrect usage of the the *DBExecutor* is given below. You cannot call the *DBExecutor* instance methods in the parallel threads.

```
// Create a parallel thread.
var task = new Task(() => {
    // Using a DBExecutor instance in a parallel thread.
    using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection()) {
        dbExecutor.StartTransaction();
    }
});
```

```
        dbExecutor.CommitTransaction();
    }
});

// Running an asynchronous task in a parallel thread.
// The execution of the program in the main thread continues on.
task.Start();
//...
var select = (Select) new Select(UserConnection)
    .Column("Id")
    .From("Contact")
    .Where("Name")
    .IsEqual(Column.Parameter("Supervisor"));
// Using an instance of DBExecutor in the main thread will cause an error,
// because the instance DBExecutor is already used in a parallel thread.
using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection()) {
    using (IDataReader dataReader = select.ExecuteReader(dbExecutor)) {
        while (dataReader.Read()) {
            //...
        }
    }
}
```

An example of correct use

A source code fragment with correct usage of the the DBExecutor is given below. The call of the DBExecutor instance methods is preformed consistently in one thread.

```
// The first use of the instance DBExecutor in the main thread.
using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection()) {
    dbExecutor.StartTransaction();
    //...
    dbExecutor.CommitTransaction();
}
//...
var select = (Select) new Select(UserConnection)
    .Column("Id")
    .From("Contact")
    .Where("Name")
    .IsEqual(Column.Parameter("Supervisor"));
// Reusing the DBExecutor instance in the main thread.
using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection()) {
    using (IDataReader dataReader = select.ExecuteReader(dbExecutor)) {
        while (dataReader.Read()) {
            //...
        }
    }
}
```

Localizable configuration resources

Contents

- **Configuration localizable resources**
- **Localizable resource structure and use**
- **Localization tables**
- **Bound data structure**

Configuration localizable resources

Beginner

Easy

Medium

Advanced

Introduction

Configuration resources are localizable strings and images used by the application to display information to the user.

The application resources are places in the packages and bound to the base schema in the schema hierarchy. When resources of a certain schema are queried, all resources are gathered throughout the hierarchy, after which a flat list of the gathered resources is generated.

Resources displayed as a hierarchy

There are two modes of displaying schema resources: design mode (Design-time) and application runtime mode (Run-time).

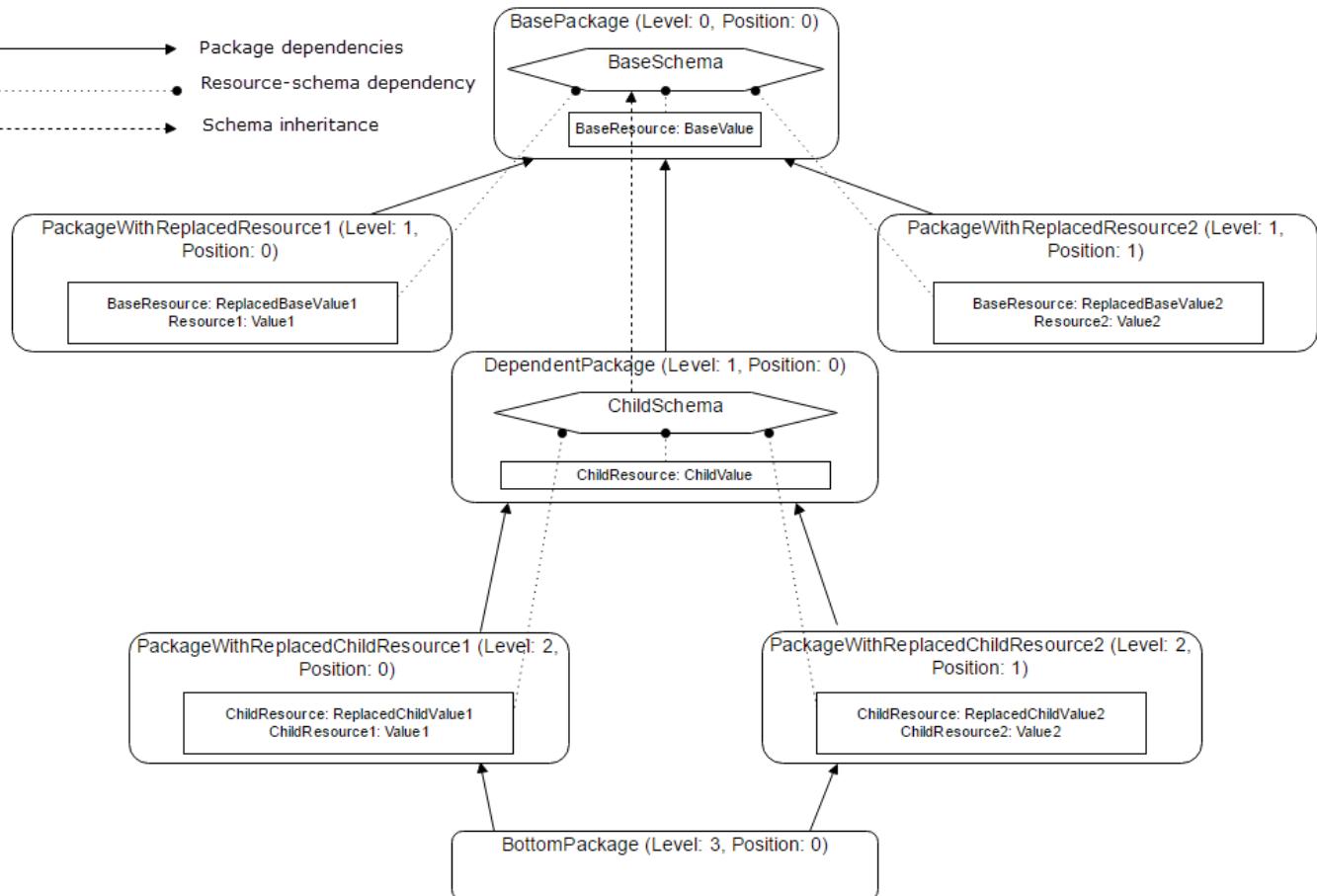
Design-time mode

This mode is used to display resources in designers and wizards. Schema resources are displayed only up to the package that contains the schema. Package resources that are not part of the hierarchy are not taken into account. For example, the *ChildSchema* schema (Fig. 1) will have the following resources:

- *BaseResource: BaseValue*;
- *ChildResource: ChildValue*.

The resources of the *PackageWithReplacedResource1* and *PackageWithReplacedResource2* packages that are not a part of the hierarchy are not taken into account. The resources in the *PackageWithReplacedChildResource1* and *PackageWithReplacedChildResource2* packages that are lower than the requested schema in the hierarchy are not taken into account as well.

Fig. 1 An example of the package hierarchy



If a schema is requested along with the package from which all resources must be obtained, then the resulting set of resources will be generated up to the level of the requested package. For example, *ChildSchema* schema resources up

to the *BottomPackage* level will be as follows:

- *BaseResource: BaseValue;*
- *ChildResource: ReplacedChildValue2;*
- *ChildResource1: Value1;*
- *ChildResource2: Value2.*

The value of the *ChildResource* here has been changed to *ReplacedChildValue2*. This occurred because it has been replaced in the packages one level lower (Level 2). Packages with higher position value take precedence. If the position values are the same, the first package (if sorted by name) will take precedence.

Run-time mode

This mode displays resources in all system sections except for designers. The mechanism for obtaining resources is similar to the mechanism used in the Design-time mode. The main difference is that when a schema is requested, the resulting list will contain resources from packages that are not directly a part of the hierarchy. For example, if the *ChildSchema* resources are requested, the result will be as follows:

- *BaseResource: ReplacedBaseValue2;*
- *Resource1: Value1;*
- *Resource2: Value2;*
- *ChildResource: ReplacedChildValue2;*
- *ChildResource1: Value1;*
- *ChildResource2: Value2.*

Default resource view

If there are no resource values that can be displayed for users for a non-default culture, then the values are “reverted” to the default culture values.

This mechanism is implemented in the *Terrasoft.Common.LocalizableString* (displays a localized string) and *Terrasoft.Common.LocalizableImage* (displays a localized image) classes. These classes contain the following properties and methods for obtaining localized values:

- *Value* – a property that returns the value of a localized object for the current culture or the default culture, if the former is not found.
- *HasValue* – a property that returns the flag indicating that the value of a localized object exists for the current culture or the default culture, if the former is not found.
- *GetCultureValue()* – a method that returns the value of a localized object for the current culture or the default culture, if the former is not found.
- *HasCultureValue()* – a method that returns the flag indicating that the value of a localized object exists for the current culture.

Storing resources

Resources needed for the application operation are stored in the database. Resources can be stored in a version control system for installing on a new application or transferring between applications.

Storing resources in the database

The resources for each string or image are stored in the *SysLocalizableValue* database table in the “key-value” format. The table structure is available in table 1. Each record in the *SysLocalizableValue* table is bound to a package and an Id of the base *Id* schema. The schema itself can be located in a different package.

Table 1. *SysLocalizableValue* table structure

Column name	Description
<i>Id</i>	Table record Id
<i>CreatedOn</i>	Table record creation date
<i>CreatedById</i>	Link to the <i>Contact</i> who created the record.
<i>ModifiedOn</i>	Table record last modification date

<i>ModifiedById</i>	Link to the last <i>Contact</i> who edited the record.
<i>SysPackageId</i>	Link to the package (<i>SysPackage</i>).
<i>SysSchemaId</i>	Link to the base schema (<i>SysSchema</i>). This column is filled in for configuration resources only.
<i>ResourceManager</i>	Name of the resource manager. This column is filled in for core resources only.
<i>SysCultureId</i>	Link to the culture (<i>SysCulture</i>).
<i>ResourceType</i>	The type of resource.
<i>IsChanged</i>	Indicates whether the resource has been modified by the user.
<i>Key</i>	Resource key.
<i>Value</i>	Value of the string resource.
<i>ImageData</i>	Value of the image resource.

Default resource saving

If a resource is created by a user whose culture is not default, a record matching the user's culture will be created for the resource. Newly created resources are automatically duplicated in the default culture. As a result, a similar resource record will be created with a link to the default culture. The new value of the resource will be displayed in all cultures if they don't have a native value for that resource (please see the "Default resource view" section of this article).

Storing resources in the version control system and file system

The resource structure in the version control system and file system is covered in the "Resource storage structure" of the "**Localizable resource structure and use**" article.

Localizable resource structure and use

Beginner

Easy

Medium

Advanced

Introduction

Starting with version 7.8.3, the storage location of localized package resources has changed. In previous versions, the resources were stored in the *SysSchemaResource* table as [BLOB data](#). Now, the localized resources are stored in the *SysLocalizableValues* table in text form.

For each set of schema resources in the package-schema-culture bundle, the checksum is stored in the *SysPackageResourceChecksum* table, which allows you to quickly determine if there are any changes to the package resources when updating the package. The checksum allowed for resources to be separated from schemas, enabling users to create translation packages.

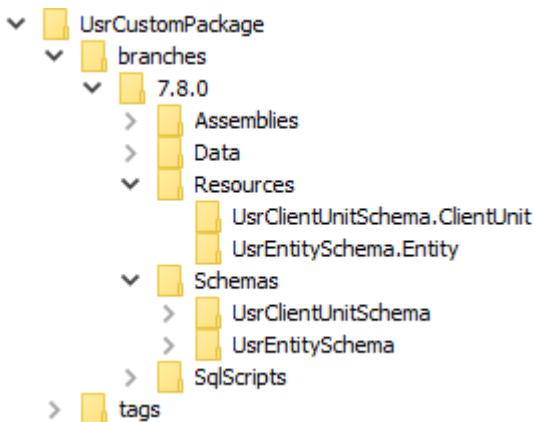
The approach to working with language cultures has changed in version 7.11.1. Now, the application uses only the cultures that have [Active] checkbox selected. This improves performance of different types of tasks, such as logging in, opening a record page, etc. However, all installed language cultures will be used when working with section and detail wizards, process and case designers and the [Translation] section.

Storing resources

Resource storage structure

The resources have been moved from schemas to a package, enabling users to create translation packages. To store schema resources with the same name, but with different managers (e.g. *Entity* and *ClientUnitSchema*), the names of the schema managers with the prefix "SchemaManager" clipping were added to the resource package names.

Fig. 1. Storing resources in a package



A package can contain resources for a schema that is defined in another package. In addition, the package can contain resources without containing schemas ("translation package").

The SysLocalizableValue table

The resources are stored in the *SysLocalizableValue* table for every localizable string or image. Each record is bound to the package and the base schema identifier. The main fields of the *SysLocalizableValue* table are listed in Table 1.

Table 1. The main fields of the SysLocalizableValue table

Column name	Description
<i>Id</i>	Record identifier.
<i>ImageData</i>	Graphic resource value.
<i>IsChanged</i>	A checkbox for specifying if the resource has been changed.
<i>Key</i>	Resource key.
<i>ResourceManager</i>	Resource manager name. Populated only for core resources.
<i>ResourceType</i>	Resource type.
<i>SysCultureId</i>	Culture identifier.
<i>SysPackageId</i>	Package identifier.
<i>SysSchemaId</i>	Base schema identifier. Populated only for configuration resources.
<i>Value</i>	String resource value.

Schema import and export

The format for storing resources in exported schemas has also changed. Now the resources in the exported schemas are stored in XML format.

Working with localizable resources

Updating the package from the repository

By using the new resource storage mechanism, all changes to localized resources are displayed when the package is updated (Fig. 2).

Fig. 2. Displaying resources when updating a package

Name	Type	Status
UsrCustomPackage	Package	Changed
UsrClientUnitSchema.en-US	Schema Resource	Changed

Possible resource states:

- [Modified] – the resource was changed.
- [Added] – a new resource has been added.
- [Deleted] – a resource has been deleted.
- [Conflict] – a resource was modified and fixed in SVN when another developer was working on it.

Committing a package to storage

When the package is committed, all changes to the localized resources are also displayed in the repository (Fig. 3).

Fig. 3. Displaying resources when a package is committed

Description		
Resources changed		
Name	Type	Status
_usrCustomPackage	Package	Modified
_usrClientUnitSchema	Schema	Modified
_usrClientUnitSchema2	Schema	Deleted
_usrClientUnitSchema.en-US	Schema Resource	Modified
_usrClientUnitSchema.es-ES	Schema Resource	Added
_usrClientUnitSchema.fr-FR	Schema Resource	Added
_usrClientUnitSchema.ru-RU	Schema Resource	Modified

Conflicts when a package is committed and updated

It is not currently possible to block resources in the application. If the developer modifies the schema resources, and the package has already been modified and the same resources have been modified in it, they will see a list of those resources with the [Conflict] state. This means that the version and contents of the resources modified by the developer do not match the version and contents committed in SVN. When the developer commits again, their modifications will block the modifications committed in SVN. Such conflicts must be resolved manually in SVN clients (e.g. Tortoise).

Fig. 4. Displaying conflicts when updating a package

Name	Type	Status
_usrCustomPackage	Package	Changed
_usrClientUnitSchema.en-US	Schema Resource	Conflicted

Editing resources in the file system.

Editing resources directly is available starting with version 7.8.3. To do this, you need to upload them to the file system (e.g. with Tortoise), and then make changes and commit to SVN.

For each resource value in the *SysLocalizableValue* table, there is only one record with the corresponding references to *SysPackageId*, *SysSchemaId*, *SysCultureId*, and a specific *Key* value. Therefore, when you commit a resource with the [Conflict] state, the table will write the last value.

Updating resources with a direct SQL query to the database

If you change the *SysLocalizableValue* table value with an SQL query, you must also change the value of the *IsChanged* column in the *SysPackageResourceChecksum* table for the corresponding schema. Otherwise, when the

package is updated, the application will not detect a conflict.

You can not add data to the *SysLocalizableValue* table with a direct SQL query, because there is no information about the added resources in the corresponding schema metadata.

Localization tables

Beginner **Easy** **Medium** **Advanced**

Localization tables are created for objects with at least one localizable column. These tables store localizable data for all languages (cultures) except for the default one.

The approach to working with language cultures has changed in version 7.11.1. Now, the application uses only the cultures that have [Active] checkbox selected. This improves performance of different types of tasks, such as logging in, opening a record page, etc. However, all installed language cultures will be used when working with section and detail wizards, process and case designers and the [Translation] section.

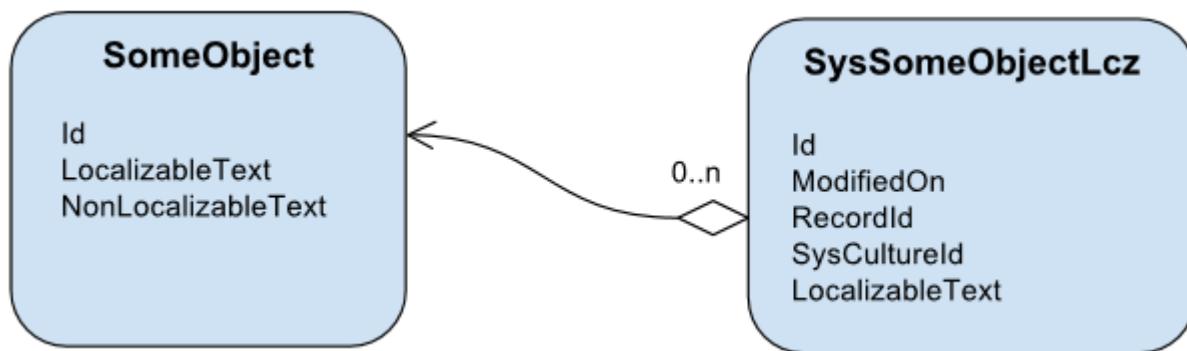
The default localization table name is *Sys[schema_name]Lcz*, where *[schema_name]* is the object schema with the localizable columns. General localization table structure:

Table 1. – General localization table structure

Column name	Description
<i>Id</i>	Record identifier.
<i>ModifiedOn</i>	Modification date.
<i>RecordId</i>	A link to the localized record in the main object table.
<i>SysCultureId</i>	Culture link.
<i>LczColumn1</i>	
<i>LczColumn2</i>	Columns corresponding to the object's localizable columns.
...	
<i>LczColumnN</i>	

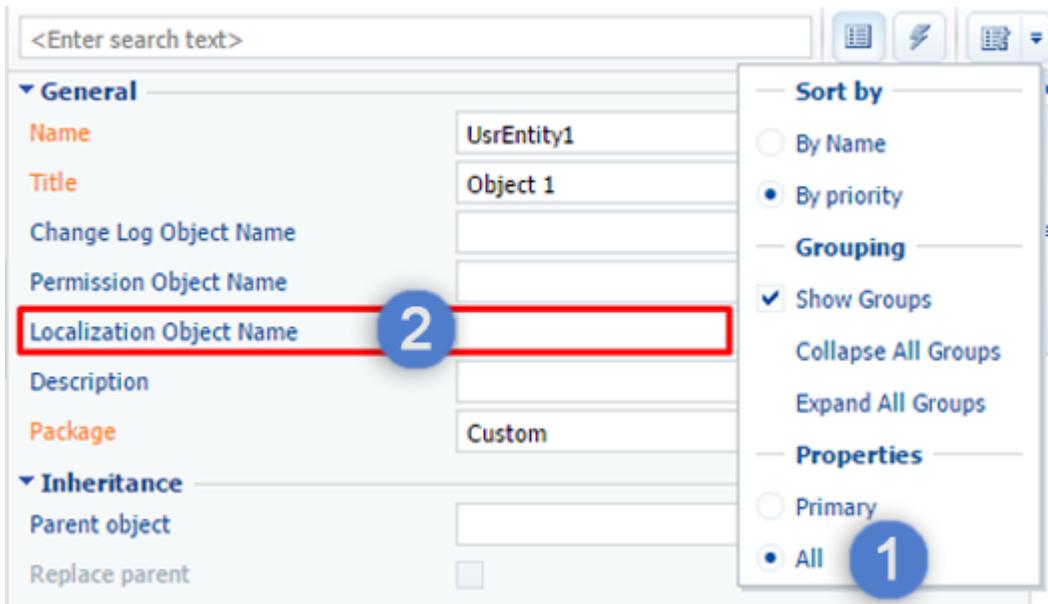
The localization table structure of the *Random* object with the *LocalizableText* column:

Fig. 1. The link between the main table and the localization table.



Use the object designer to specify / change the name of the table. Go to the advanced schema object properties and specify the name of the localization object.

Fig. 2. Localization table name in the object designer



Bound data structure

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

Introduction

Application comes with full multilanguage support since version 7.8. As a result, the storage structure of **resources** and bound data has been reworked.

Data binding specifics (compared to version 7.8):

- A new *SysPackageDataLcz* table now contains localized bindings data.
- A new mechanism for creating and installing bindings.
- A new SVN data storing structure.

The *SysPackageDataLcz* localization table

An additional table is used for storing localized bound data:

Table 1. *SysPackageDataLcz* table columns

Column name	Description
<i>Id</i>	Unique identifier.
<i>SysPackageSchemaDataId</i>	A reference to the unique binding identifier in the <i>SysPackageSchemaData</i> table.
<i>SysCultureId</i>	Unique culture identifier reference.
<i>Data</i>	Localized data.

The binding mechanism interface is identical to that of the previous versions.

Creating a binding

If a schema does not contain localized columns, the bound data for this schema is still stored in the *SysPackageSchemaData* table. Data binding for a schema with localized columns:

- The bound data is still stored in the *SysPackageSchemaData* table.
- The *SysPackageDataLcz* table contains localized bindings data.
- Every record in *SysPackageDataLcz* corresponds with a record in *SysPackageSchemaData*, with a reference to the unique *SysCultureId* identifier. For example, if two cultures (English and Spanish) are

installed in the system, each entry in the *SysPackageSchemaData* table will correspond to entries in the *SysPackageDataLcz* table, with a reference to the corresponding record identifier in the *SysPackageSchemaData* table, and the culture identifier in the *SysCulture* table.

Installing bound data

Installing data for a schema without localized columns is done in the corresponding schema table. If the data includes any localized values (i.e. there are corresponding records in the *SysPackageDataLcz* table), the installation occurs not only in the corresponding schema table, but also in its localized *Sys[schema_name]Lcz* table.

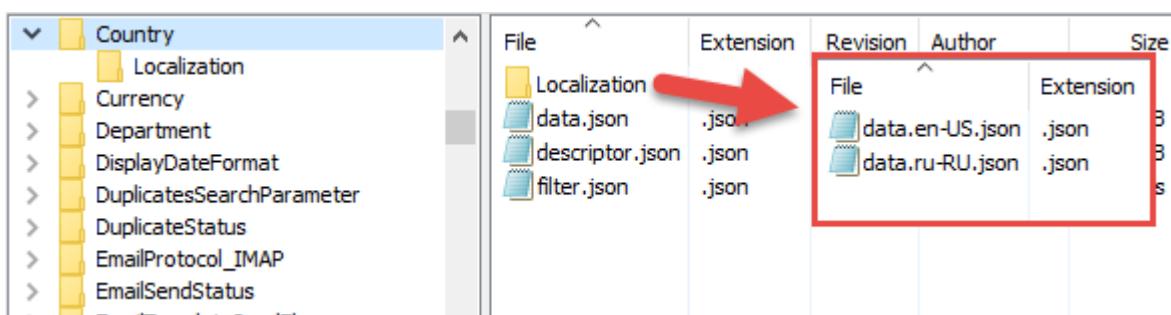
For example, the bound data for the *ContactType* schema is installed. Non-localized data is installed in the *ContactType* table, and the localized data is installed in the *ContactType* table (default culture values), and the *SysContactTypeLcz* table (the values of all other cultures included in the binding and in the system).

If you are working with a system schema (the name begins with the “Sys” prefix), then the “Sys” prefix is not re-added to the localization table. For example, if the schema name is *SysTest1*, the localized data table name will be *SysTest1Lcz*, and not *SysSysTest1Lcz*.

Storing data in SVN

The structure of storing the bound schema data in SVN for application version 7.8 and higher:

Fig. 1. SVN data storing structure



The screenshot shows the SVN data storing structure. On the left, there is a file tree view of the 'Country' schema. It includes a 'Localization' folder which contains 'data.json', 'descriptor.json', and 'filter.json'. To the right of the tree is a detailed file list table. The first column is 'File', the second is 'Extension', and the third is 'Revision'. A red arrow points from the 'Localization' folder in the tree to the 'File' column in the table. The table shows three files: 'data.en-US.json' and 'data.ru-RU.json' both have '.json' extensions. A red box highlights the entire table area.

File	Extension	Revision	Author	Size
File	Extension			
data.en-US.json	.json			3
data.ru-RU.json	.json			3

Non-localized data is stored in the *data.json* file. All localized data is located in the corresponding files in the *Localization* subdirectory. For example, for the *Country* schema of the *Base* package, the data is localized for only two languages and stored in the corresponding files - *data.en-US.json* and *data.es-ES.json* (Fig. 1).

Entity event layer

Beginner Easy Medium **Advanced**

Introduction

In Creatio version 7.12.4, we have enabled the development of object business logic without using event subprocesses.

The *Entity* event layer mechanism is triggered after executing the event subprocesses of an object.

To add the necessary actions to the handler of the needed object event (inheritor of the *Entity* class):

1. Create a class that inherits *BaseEntityEventListener*.
2. Decorate the class with the *EntityEventListener* attribute and specify the name of entity whose event subscription must be executed.
3. Override the handler method of the needed event.

Example:

```
// Event listener of the "Activity" entity.
[EntityEventListener(SchemaName = "Activity")]
public class ActivityEntityEventListener : BaseEntityEventListener
{
    // Overriding the event handler of entity saving event.
    public override void OnSaved(object sender, EntityEventArgs e) {
```

```

    //Calling the parent implementation.
    base.OnSaved(sender, e);
    //...
}
}

```

Classes that form the Entity event layer mechanism:

The BaseEntityEventListener class

The BaseEntityEventListener class provides handler methods for different entity events (table 1).

Table 1. Entity event handler methods

<i>OnDeleted(object sender, EntityAfterEventArgs e)</i>	Event handler after deleting a record.
<i>OnDeleting(object sender, EntityBeforeEventArgs e)</i>	Event handler before deleting a record.
<i>OnInserted(object sender, EntityAfterEventArgs e)</i>	Event handler after adding a record.
<i>OnInserting(object sender, EntityBeforeEventArgs e)</i>	Event handler before adding a record.
<i>OnSaved(object sender, EntityAfterEventArgs e)</i>	Event handler after saving a record.
<i>OnSaving(object sender, EntityBeforeEventArgs e)</i>	Event handler before saving a record.
<i>OnUpdated(object sender, EntityAfterEventArgs e)</i>	Event handler after updating a record.
<i>OnUpdating(object sender, EntityBeforeEventArgs e)</i>	Event handler before updating a record.

Method parameters:

- *sender* – the link to the instance of entity generating the event. Parameter type – *Object*.
- *e* – event arguments. Depending on the execution time of the handler method (after or before the event), the argument type can be either *EntityAfterEventArgs* or *EntityBeforeEventArgs*.

The algorithm of calling event handler methods is provided in table 2.

Table 2. Algorithm of calling event handler methods

Create	Change	Delete
OnSaving()	OnSaving()	OnDeleting()
OnInserting()	OnUpdating()	OnDeleted()
OnInserted()	OnUpdated()	
OnSaved()	OnSaved()	

Obtaining UserConnection

You can obtain the *UserConnection* instance in the event handlers from *sender* parameter:

```

[EntityEventListener(SchemaName = "Activity")]
public class ActivityEntityEventListener : BaseEntityEventListener
{
    public override void OnSaved(object sender, EntityAfterEventArgs e) {
        base.OnSaved(sender, e);
        var entity = (Entity) sender;
        var userConnection = entity.UserConnection;
    }
}

```

The approach to obtaining the *UserConnection* from *HttpContext* instance is not always correct. For example, the *HttpContext* instance may not exist at the time when event handlers are firing in the business processes or in the task scheduler.

The EntityAfterEventArgs class

The class provides properties with arguments of the handler method that is executed after the event occurs.

- *ModifiedColumnValues* – collection of modified columns.
- *PrimaryColumnName* – record identifier.

The EntityBeforeEventArgs class

The class provides properties with arguments of the handler method that is executed before the event occurs.

- *KeyValue* – record identifier.
- *IsCanceled* – enables canceling the further event execution.
- *AdditionalCondition* – enables providing additional description of entity filter conditions before the action.

The EntityEventListener attribute

The *EntityEventListener* attribute (the *EntityEventListenerAttribute* class) is designed for listener registration. The listener can be connected with all objects (*IsGlobal* = *true*) or with a specific object (for example, *SchemaName* = "Contact"). One listener-class can be tagged with many attributes for defining the necessary "set" of "listened-to" entities.

Asynchronous operations in the Entity event layer

It is often a case when additional business logic of an object is time consuming and is executed consistently. Such an approach has its negative impact upon the efficiency of customer part, for example, when saving or modifying entities.

To eliminate such problems, we have developed a method of asynchronous execution of operations, based on the Entity event layer.

For example, when adding a new activity, you need to execute additional logic which can be executed asynchronously. For this, create a class, e.g., *DoSomethingActivityAsyncOperation* with implementation of the *IEntityEventAsyncOperation* interface (see below).

```
//Class implementing the asynchronous calling of operations.
public class DoSomethingActivityAsyncOperation: IEntityEventAsyncOperation
{
    // Start method of the class.
    public void Execute(UserConnection userConnection, EntityEventAsyncOperationArgs
arguments) {
        // ...
    }
}
```

Receive the instance, implementing the *IEntityEventAsyncExecutor* interface in the handler-method of the activity object listener-class after saving the record via class factory, prepare parameters and pass the *DoSomethingActivityAsyncOperation* operation class to execution.

```
[EntityEventListener(SchemaName = "Activity")]
public class ActivityEntityEventListener : BaseEntityEventListener
{
    // Event handler method after saving the entity.
    public override void OnSaved(object sender, EntityEventArgs e) {
        base.OnSaved(sender, e);
        // Class instance for asynchronous execution.
        var asyncExecutor = ClassFactory.Get<IEntityEventAsyncExecutor>();
        // Parameters for asynchronous execution.
```

```
var operationArgs = new EntityEventAsyncOperationArgs((Entity)sender, e);
// Execution in asynchronous mode.
asyncExecutor.ExecuteAsync<DoSomethingActivityAsyncOperation>(operationArgs);
}
}
```

The IEntityEventAsyncExecutor interface

Provides method for asynchronous operation execution.

- *void ExecuteAsync<TOperation>(object parameters)* – the typed method for launching operations with parameters, where *TOperation* – is the configuration class implementing the *IEntityEventAsyncOperation* interface.

The IEntityEventAsyncOperation interface

Provides method for launch of an asynchronous operation.

- *void Execute(UserConnection userConnection, EntityEventAsyncOperationArgs arguments)* – launch method.

We do not recommending describing the logic of changing the primary entity in the class implementing the *IEntityEventAsyncOperation* interface. Such use can lead to incorrect data creation. We do not recommend executing lightweight operations (for example, calculating a field value), since creating a separate flow might take more time than executing the operation itself.

The EntityEventAsyncOperationArgs class

This class instances are used as arguments for passing to the asynchronous operation.

Properties.

- *EntityId* – record identifier.
- *EntitySchemaName* – name of the schema.
- *EntityColumnValues* – the glossary of current column values of an entity.
- *OldEntityColumnValues* – the glossary of old column values of an entity.

Developing the source code in the file content (project package)

Beginner

Easy

Medium

Advanced

The project package model is one of Creatio's standards for accelerating the development of the server schemas for the application.

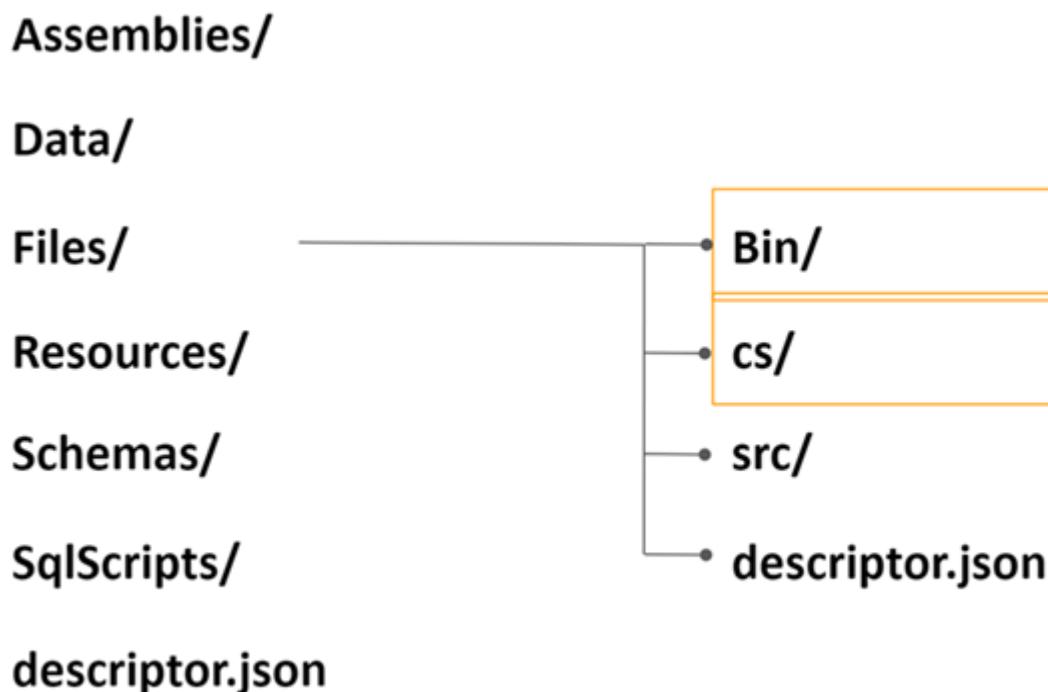
A project package is a package that enables developing new functionality like a regular C# project (fig. 1).

Fig. 1. The difference between a project package and a regular package.



The new functionality is included into the **file content package** (in the Files directory) as a compiled dynamic library and a collection of CS files.

Fig. 2. The folder structure of a project package

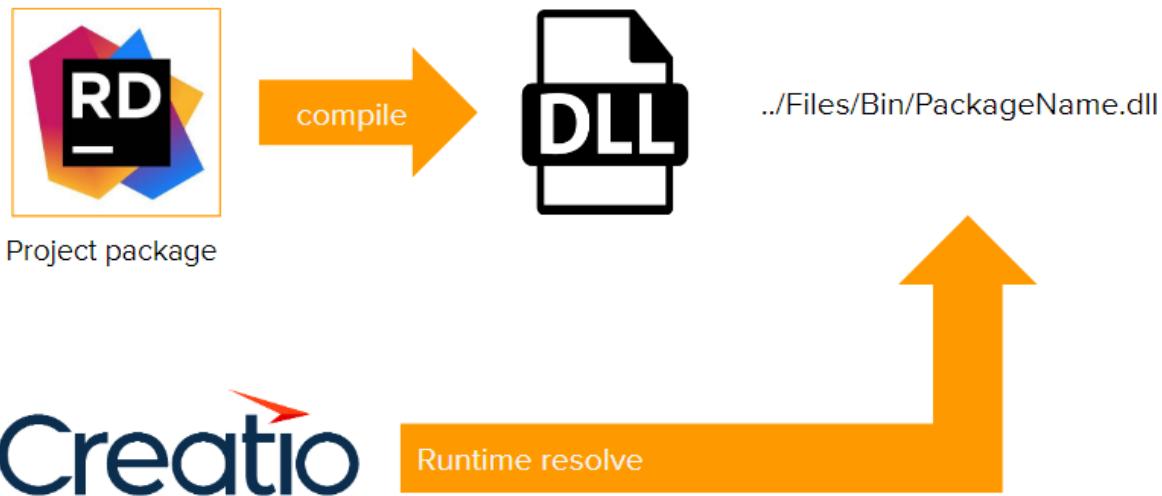


The process for delivering the new functionality to the Creatio application using a project package (fig. 3):

1. The project package is compiled to the library as a separate CS project. The library will have the same name as the package.
2. Compiled files are saved to the ..//Files/Bin/ folder as a DDL file: ..//Files/Bin/[PackageName].dll file.

3. When starting or restarting, the Creatio application will collect information about pre-set libraries and promptly include them.

Fig. 3. The relations of a project package



Advantages of using project packages for developing the server schemas.

1. When the number of schemas in the source code of the packages is significant, compilation takes a lot of time as well. Using project packages makes it possible to reduce the speed of compilation from 30-120 seconds to 1-2 seconds.
2. Project packages enable deploying developed solutions on the production environment without utilizing traditional delivery models. All it takes is compiling the project package, passing a library, and copying it to the folder. After a restart, all the changes will be applied. Compiling the configuration is not required for the delivery of new functionality.
3. Developing the server schemas is significantly easier for cloud applications.
4. Project packages enable easy and thorough dependency tracking for any implementation. This is immensely helpful for understanding the dependency tree and singling out the classes that are to be tested upon updating the functionality.
5. Automated testing of new functionality becomes available.

Recommended tools for project package development

1. [Creatio command-line interface utility \(cli\)](#) – is an open source utility for integration, development, and CI/CD. Enables:
 - creating project packages
 - delivery of a package to an on-premises or cloud environment
 - retrieval of a package from an on-premises or cloud environment
 - restarting the application
 - conversion of existing packages.
2. [CreatioSDK](#) – a NuGet package, which provides a set of development tools for building Creatio applications.

Class Description

Contents

- **The Select class**
- **The EntitySchemaQuery class**
- **The Insert class**

- **The InsertSelect class**
- **The Update class**
- **The Delete class**
- **The Entity class**
- **The EntityMapper class**
- **The QueryFunction class**
- **The EntitySchemaQueryFunction class**

The Select class

Beginner

Easy

Medium

Advanced

Introduction

The *Terrasoft.Core.DB.Select* class is used to build queries for the selection of records from the database tables. As a result of creating and configuring the instance of this class, the SELECT SQL-expression query to the application database will be built. You can add the needed icons, filtering, and restriction conditions to the query. The query results are returned as a class instance that implements the *System.Data.IDataReader* interface or as a scalar value of the needed type.

When working with the *Select* class, the current user permissions are not taken into consideration. All records from the application database are available. The data located in the cache repository are not taken into consideration (see the “**Repositories. Types and recommendations on use**” article). Use the *EntitySchemaQuery* class to access additional permission control options and Creatio cache repository operation.

The “Terrasoft.Core.DB.Select“ class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the *Select* class methods and properties, its parent classes, and the implemented interfaces.

Constructors

`public Select((UserConnection userConnection))`

Creates a class instance with the specified UserConnection parameter.

`public Select((UserConnection userConnection, CancellationToken cancellationToken))`

Creates a class instance with the specified UserConnection parameter and the cancellation token for [managed threads](#).

`public Select((Select source))`

Creates a class instance that is a clone of the instance passed as an argument.

Properties

Table 1. Primary properties of the *Select* class

`UserConnection`

Terrasoft.Core.UserConnection

The current user connection at the moment of executing the query.

`RowCount`

`int`

The number of records returned by the query after execution.

`Parameters`

Terrasoft.Core.DB.QueryParameterCollection

Collection of the query parameters.

HasParameters

bool

Determines whether any query parameters are available.

BuildParametersAsValue

bool

Determines whether the query parameters are to be added as values into the query text.

Joins

Terrasoft.Core.DB.JoinCollection

The collection of *Join* expressions in the query.

HasJoins

bool

Determines whether *Join* expressions are available in the query.

Condition

Terrasoft.Core.DB.QueryCondition

Condition of *Where* expression in the query.

HasCondition

bool

Determines whether *Where* expression is available in the query.

HavingCondition

Terrasoft.Core.DB.QueryCondition

Condition of *Having* expression in the query.

HasHavingCondition

bool

Determines whether *Having* expression is available in the query.

OrderByItems

Terrasoft.Core.DB.OrderByItemCollection

The collection of expressions by which the query results are sorted.

HasOrderByItems

bool

Determines whether the conditions of query result sorting are available.

GroupByItems

Terrasoft.Core.DB.QueryColumnExpressionCollection

The collection of expressions by which the query results are grouped.

HasGroupByItems

bool

Determines whether the conditions of query result grouping are available.

IsDistinct

bool

Determines whether the query should return unique records only.

Columns

Terrasoft.Core.DB.QueryColumnExpressionCollection

Collection of query column expressions.

OffsetFetchPaging
bool

Determines whether the per-page returning of the query result is available.

RowsOffset
int

Number of rows to skip when returning the query result.

QueryKind
Terrasoft.Common.QueryKind

The query type (see “**Separate query pool (on-line documentation)**”).

Methods

Table 2. Primary methods of the *Select* class

string GetSqlText()

Returns the SQL text of the current query.

void BuildSqlText((StringBuilder sb))

Generates the query text via the *StringBuilder* instance.

void ResetCachedSqlText()

Clears the cached text of the query.

QueryParameterCollection GetUsingParameters()

Returns the collection of parameters used by the query.

void ResetParameters()

Clears the collection of the query parameters.

QueryParameterCollection InitializeParameters()

Initiates the collection of the query parameters.

IDataReader ExecuteReader((DBExecutor dbExecutor))

Executes the query using the *DBExecutor* instance. Returns the object implementing the *IDataReader* interface.

IDataReader ExecuteReader((DBExecutor dbExecutor, CommandBehavior behavior))

Executes the query using the *DBExecutor* instance. Returns the object implementing the *IDataReader* interface. The *behavior* parameter describes query results and how they affect the database.

void ExecuteReader((ExecuteReaderReadMethod readMethod))

Executes the query by calling the passed over *ExecuteReaderReadMethod* delegate method for every record of the result set.

TResult ExecuteScalar<TResult>()

Executes the query. Returns the typed first column of the first record of the result set.

TResult ExecuteScalar<TResult>((DBExecutor dbExecutor))

Executes the query using the *DBExecutor* instance. Returns the typed first column of the first record of the result set.

Select Distinct()

Adds the DISTINCT keyword to the SQL-query. Excludes record duplicates in the result set. Returns the query instance.

Select Top((int rowCount))

Sets the number of records returned in the result set. The *RowCount* property value is applied (see table 1). Returns the query instance.

`Select As((string alias))`

Adds the alias for the last query expression specified in the argument. Returns the query instance.

```
Select Column((string sourceColumnAlias))
Select Column((string sourceAlias, string sourceColumnAlias))
Select Column((Select subSelect))
Select Column((Query subSelectQuery))
Select Column((QueryCase queryCase))
Select Column((QueryParameter queryParameter))
Select Column((QueryColumnExpression columnExpression))
```

Adds an expression, a subquery or a parameter to the query column expression collection. Returns the query instance.

Parameters:

- *sourceColumnAlias* – an alias of the column for which the expression is added;
- *sourceAlias* – an alias of the source that the column expression is added from;
- *subSelect* – the added subquery for data selection;
- *subSelectQuery* – the added subquery;
- *queryCase* – the added expression for the *Case* operator;
- *queryParameter* – the added parameter query;
- *columnExpression* – the expression, for whose results the condition is added.

```
Select From((string schemaName))
Select From((Select subSelect))
Select From((Query subSelectQuery))
Select From((QuerySourceExpression sourceExpression))
```

Adds the data source to the query. Returns the query instance.

Parameters:

- *schemaName* – the schema name;
- *subSelect* – the selection subquery, whose results become the data source for the current query;
- *subSelectQuery* – the subquery, whose results become the data source for the current query;
- *sourceExpression* – the expression of the query data source.

```
Join Join((JoinType joinType, string schemaName))
Join Join((JoinType joinType, Select subSelect))
Join Join((JoinType joinType, Query subSelectQuery))
Join Join((JoinType joinType, QuerySourceExpression sourceExpression))
```

Joins a schema, a subquery or an expression to the current query.

Parameters:

- *joinType* – join type (see table 3);
- *schemaName* – the joined schema name;
- *subSelect* – the joined subquery for data selection;
- *subSelectQuery* – the joined subquery;
- *sourceExpression* – the joined expression.

```
QueryCondition Where()
QueryCondition Where((string sourceColumnAlias))
QueryCondition Where((string sourceAlias, string sourceColumnAlias))
QueryCondition Where((Select subSelect))
QueryCondition Where((Query subSelectQuery))
QueryCondition Where((QueryColumnExpression columnExpression))
Query Where((QueryCondition condition))
```

Adds the initial condition to the current query.

Parameters:

- *sourceColumnAlias* – an alias of the column for which the condition is added;
- *sourceAlias* – the alias of the source;
- *subSelect* – a subquery of the data selection, for whose results the condition is added;
- *subSelectQuery* – the subquery, for whose results the condition is added;
- *columnExpression* – the expression, for whose results the condition is added;
- *condition* – the query condition.

```
QueryCondition And()
QueryCondition And((string sourceColumnAlias))
QueryCondition And((string sourceAlias, string sourceColumnAlias))
QueryCondition And((Select subSelect))
QueryCondition And((Query subSelectQuery))
QueryCondition And((QueryParameter parameter))
QueryCondition And((QueryColumnExpression columnExpression))
Query And((QueryCondition condition))
```

Adds the condition (predicate) to the current query condition using the AND logical operation.

Parameters:

- *sourceColumnAlias* – an alias of the column, for which the predicate is added;
- *sourceAlias* – the alias of the source;
- *subSelect* – the data selection subquery used as a predicate;
- *subSelectQuery* – the subquery used as a predicate;
- *parameter* – parameter that the predicate is added to;
- *columnExpression* – the expression used as a predicate;
- *condition* – the query condition.

```
QueryCondition Or()
QueryCondition Or((string sourceColumnAlias))
QueryCondition Or((string sourceAlias, string sourceColumnAlias))
QueryCondition Or((Select subSelect))
QueryCondition Or((Query subSelectQuery))
QueryCondition Or((QueryParameter parameter))
QueryCondition Or((QueryColumnExpression columnExpression))
Query Or((QueryCondition condition))
```

Adds the condition (predicate) to the current query condition using the OR logical operation.

Parameters:

- *sourceColumnAlias* – an alias of the column, for which the predicate is added;
- *sourceAlias* – the alias of the source;
- *subSelect* – the data selection subquery used as a predicate;
- *subSelectQuery* – the subquery used as a predicate;
- *parameter* – parameter that the predicate is added to;
- *columnExpression* – the expression used as a predicate;
- *condition* – the query condition.

```
Query OrderBy((OrderDirectionStrict direction, string sourceColumnAlias))
Query OrderBy((OrderDirectionStrict direction, string sourceAlias, string sourceColumnAlias))
Query OrderBy((OrderDirectionStrict direction, QueryFunction queryFunction))
Query OrderBy((OrderDirectionStrict direction, Select subSelect))
Query OrderBy((OrderDirectionStrict direction, Query subSelectQuery))
Query OrderBy((OrderDirectionStrict direction, QueryColumnExpression columnExpression))
```

Executes sorting of query results. Returns the query instance.

Parameters:

- *direction* – sorting order;
- *sourceColumnAlias* – the alias of the column by which the sorting is performed;

- *sourceAlias* – the alias of the source;
- *queryFunction* – the function whose value is used as the sort key;
- *subSelect* – the data selection subquery, whose results are used as the sort key;
- *subSelectQuery* – the subquery, whose results are used as the sort key;
- *columnExpression* – the expression, whose results are used as the sort key.

```
Query OrderByAsc((string sourceColumnAlias))
Query OrderByAsc((string sourceAlias, string sourceColumnAlias))
Query OrderByAsc((Select subSelect))
Query OrderByAsc((Query subSelectQuery))
Query OrderByAsc((QueryColumnExpression columnExpression))
```

Sorts query results in the ascending order. Returns the query instance.

Parameters:

- *sourceColumnAlias* – the alias of the column by which the sorting is performed;
- *sourceAlias* – the alias of the source;
- *subSelect* – the data selection subquery, whose results are used as the sort key;
- *subSelectQuery* – the subquery, whose results are used as the sort key;
- *columnExpression* – the expression, whose results are used as the sort key.

```
Query OrderByDesc((string sourceColumnAlias))
Query OrderByDesc((string sourceAlias, string sourceColumnAlias))
Query OrderByDesc((Select subSelect))
Query OrderByDesc((Query subSelectQuery))
Query OrderByDesc((QueryColumnExpression columnExpression))
```

Sorts query results in the descending order. Returns the query instance.

Parameters:

- *sourceColumnAlias* – the alias of the column by which the sorting is performed;
- *sourceAlias* – the alias of the source;
- *subSelect* – the selection subquery, whose results are used as the sort key;
- *subSelectQuery* – the subquery, whose results are used as the sort key;
- *columnExpression* – the expression, whose results are used as the sort key.

```
Query GroupBy((string sourceColumnAlias))
Query GroupBy((string sourceAlias, string sourceColumnAlias))
Query GroupBy((QueryColumnExpression columnExpression))
```

Executes grouping of query results. Returns the query instance.

Parameters:

- *sourceColumnAlias* – the alias of the column by which the grouping is performed;
- *sourceAlias* – the alias of the source;
- *columnExpression* – the expression, whose results are used as the group key.

```
QueryCondition Having(())
QueryCondition Having((string sourceColumnAlias))
QueryCondition Having((string sourceAlias, string sourceColumnAlias))
QueryCondition Having((Select subSelect))
QueryCondition Having((Query subSelectQuery))
QueryCondition Having((QueryParameter parameter))
QueryCondition Having((QueryColumnExpression columnExpression))
```

Adds the grouping condition to the current query. Returns the *Terrasoft.Core.DB.QueryCondition* instance that represents the grouping condition for the query parameter.

Parameters:

- *sourceColumnAlias* – the alias of the column by which the grouping condition is added;
- *sourceAlias* – the alias of the source;

- *subSelect* – the selection subquery, for whose results the grouping condition is added;
- *subSelectQuery* – the subquery, for whose results the grouping condition is added;
- *parameter* – the query parameter, for which the grouping condition is added;
- *columnExpression* – the expression used as a predicate.

Query Union((Select unionSelect))

Query Union((Query unionSelectQuery))

Combines the results of the passed query with the results of the current query excluding the duplicates from the result set.

Parameters:

- *subSelect* – selection subquery;
- *subSelectQuery* – subquery.

Query UnionAll((Select unionSelect))

Query UnionAll((Query unionSelectQuery))

Combines the results of the passes query with the results of the current query. The duplicates are not excluded from the result set.

Parameters:

- *subSelect* – selection subquery;
- *subSelectQuery* – subquery.

Table 3. Types of joins (the “Terrasoft.Core.DB.JoinType” enumeration)

Inner

Inner join.

LeftOuter

Left outer join.

RightOuter

Right outer join.

FullOuter

Full join.

Cross

Cross join.

See also

- **Retrieving data from the database**

The EntitySchemaQuery class

Beginner

Easy

Medium

Advanced

Introduction

The *Terrasoft.Core.Entities.EntitySchemaQuery* class is used to build queries for selecting records in Creatio database tables. As a result of creating and configuring the instance of this class, the SELECT SQL-expression query to the application database will be built. You can add the needed icons, filtering, and restriction conditions to the query.

EntitySchemaQuery implements a mechanism for working with the repository (Creatio cache or user-defined arbitrary repository). When *EntitySchemaQuery* is performed, the data retrieved from the server database are cached. The query cache can be an arbitrary repository that implements the *Terrasoft.Core.Store.ICacheStore* interface. By default, the cache of the *EntitySchemaQuery* query is the session-level Creatio cache (only the data from the current user session are available) with local storage.

For the *EntitySchemaQuery* queries, you can determine additional settings that specify the parameters for the page-by-page output of query results and parameters for building a hierarchical query. The *Terrasoft.Core.Entities.EntitySchemaQueryOptions* class is designed for this purpose.

The result of *EntitySchemaQuery* is the *Terrasoft.Nui.ServiceModel.DataContract.EntityCollection* instance (the *Terrasoft.Core.Entities.Entity* class instance collection). Each *Entity* collection instance represents a string of data returned by the query.

The “**Terrasoft.Core.Entities.EntitySchemaQuery**“ class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaQuery* class.

Constructors

`public EntitySchemaQuery((EntitySchema rootSchema))`

Creates a class instance, where the *EntitySchema* passed instance is set as a root schema. The manager of the passed-in instance of the root schema is set as the schema manager.

`public EntitySchemaQuery((EntitySchemaManager entitySchemaManager, string sourceSchemaName))`

Creates a class instance with the specified *EntitySchemaManager* and the root schema passed as an argument.

`public EntitySchemaQuery((EntitySchemaQuery source))`

Creates a class instance that is a clone of the instance passed as an argument.

Properties

Table 1. Primary properties of the *EntitySchemaQuery* class

Cache

Terrasoft.Core.Store.ICacheStore

The query cache.

CacheItemName

string

Name of the cache item.

CanReadUncommittedData

bool

Determines whether the query results will include the data for which the transaction is not completed.

Caption

Terrasoft.Common.LocalizableString

Header.

ChunkSize

int

The number of strings in one chunk.

Columns:

Terrasoft.Core.Entities.EntitySchemaQueryColumnCollection

Collection of columns of the current entity schema query.

DataValueTypeManager

DataValueTypeManager

Manager of the data type values.

EntitySchemaManager

Terrasoft.Core.Entities.EntitySchemaManager

Entity schema manager.

Filters

Terrasoft.Core.Entities.EntitySchemaQueryFilterCollection

Collection of filters of the current entity schema query.

HideSecurityValue

bool

Determines whether the values of the encrypted columns will be hidden.

IgnoreDisplayValues

bool

Determines whether the displayed column values will be used in the query.

IsDistinct

bool

Indicates whether duplicates in the resulting data set should be removed.

IsInherited

bool

Indicates whether the query is inherited.

JoinRightState

QueryJoinRightLevel

Determines the conditions for applying permissions when using related tables if the schema is managed by records.

Manager

Terrasoft.Core.IManager

Schema manager.

ManagerItem

Terrasoft.Core.IManagerItem

Manager element.

Name

string

Name.

ParentCollection

Terrasoft.Core.Entities.EntitySchemaQueryCollection

A collection of queries to which the current request to the object schema belongs.

ParentEntitySchema

Terrasoft.Core.Entities.EntitySchema

Parent schema of the query.

PrimaryQueryColumn

Terrasoft.Core.Entities.EntitySchemaQueryColumn

The column created from the primary column of the root schema Initialized during the first access.

QueryOptimize

bool

Allows using query optimization.

RootSchema

Terrasoft.Core.Entities.EntitySchema

The root schema.

RowCount

int

Number of rows that are returned by the query.

SchemaAliasPrefix

string

The prefix used to create schema alias.

SkipRowCount

int

Number of rows to skip when returning the query result.

UseAdminRights

bool

The parameter that defines whether permissions will be taken into account when constructing a data acquisition request.

UseLocalization

bool

Determines whether localizable data will be used.

UseOffsetFetchPaging

bool

Determines whether the per-page returning of the query result is available.

UseRecordDeactivation

bool

Determines whether data will be excluded from filtering.

Methods

Table 2. Primary methods of the *EntitySchemaQuery* class

void AddAllSchemaColumns((bool skipSystemColumns))

The object schema adds all the columns of the root schema in the column collection of the current query.

```
EntitySchemaQueryColumn AddColumn((string columnPath, AggregationTypeStrict aggregationType, out EntitySchemaQuery subQuery))
void AddColumn((EntitySchemaQueryColumn queryColumn))
EntitySchemaQueryColumn AddColumn((string columnPath))
EntitySchemaQueryColumn AddColumn((EntitySchemaQueryFunction function))
EntitySchemaQueryColumn AddColumn((object parameterValue, DataValueType parameterDataType parameterValueType))
EntitySchemaQueryColumn AddColumn((EntitySchemaQuery subQuery))
```

Creates and inserts a column in the current entity schema query.

Parameters:

- *columnPath* – path to the schema column in relation to the root schema;
- *aggregationType* – the type of aggregating function. The enumeration type values of the *Terrasoft.Common.AggregationTypeStrict* aggregate function are passed as a parameter;
- *subQuery* – reference to the created subquery placed in the column;

- *queryColumn* – the *EntitySchemaQueryColumn* instance to be added to the column collection of the current query;
- *Function* – the *EntitySchemaQueryFunction* function instance;
- *parameterValue* – the value of the parameter added to the query as a column;
- *parameterValueType* – the type of parameter value added to the query as a column.

```
void ClearCache()
```

Clears the cache of the current query.

```
static void ClearDefCache((string cacheItemName))
```

Removes the item with the specified *cacheItemName* name from the cache of the query.

```
object Clone()
```

Creates a clone of the current *EntitySchemaQuery* instance.

```
EntitySchemaQueryExpression CreateAggregationEntitySchemaExpression((string leftExprColumnPath,  
AggregationTypeStrict leftExprAggregationType))
```

Returns expression of the aggregated function with the specified aggregation type from the *Terrasoft.Common.AggregationTypeStrict* enumeration for the column, located at the *leftExprColumnPath* specified path.

```
static EntitySchemaQueryExpression CreateParameterExpression((object parameterValue))
```

```
static EntitySchemaQueryExpression CreateParameterExpression((object parameterValue, DataValueType  
valueType))
```

```
static EntitySchemaQueryExpression CreateParameterExpression((object parameterValue, string displayValue,  
DataValueType valueType))
```

Creates an expression for the query parameter.

Parameters:

- *parameterValue* – the type of the parameter;
- *valueType* – the type of the parameter value;
- *displayValue* – the parameter displayed value.

```
static IEnumerable CreateParameterExpressions((DataValueType valueType, params object[] parameterValues))
```

```
static IEnumerable CreateParameterExpressions((DataValueType valueType, IEnumerable<object>  
parameterValues))
```

Creates an expression collection for the query parameters with a certain *DataValueType* type of data.

```
static EntitySchemaQueryExpression CreateSchemaColumnExpression((EntitySchemaQuery parentQuery,  
EntitySchema rootSchema, string columnPath, bool useCoalesceFunctionForMultiLookup, bool useDisplayValue))
```

```
static EntitySchemaQueryExpression CreateSchemaColumnExpression((EntitySchema rootSchema, string  
columnPath, bool useCoalesceFunctionForMultiLookup))
```

```
EntitySchemaQueryExpression CreateSchemaColumnExpression((string columnPath, bool  
useCoalesceFunctionForMultiLookup))
```

Returns the expression of the entity schema column.

Parameters:

- *parentQuery* – the entity schema query for which the column expression is created;
- *rootSchema* – the root schema;
- *columnPath* – path to the schema column in relation to the root schema;
- *useCoalesceFunctionForMultiLookup* – indicates whether to use the COALESCE function for the column of the lookup type. Optional parameter, set to *false* by default.
- *useDisplayValue* – indicates whether to use the displayed value for the column. Optional parameter, set to *false* by default.

```
Enumerable CreateSchemaColumnExpressions((params string[] columnPaths))
```

```
IEnumerable CreateSchemaColumnExpressions((IEnumerable columnPaths, bool  
useCoalesceFunctionForMultiLookup))
```

Returns the collection of column expressions of the entity schema query by the specified *columnPaths*. collection of

paths to columns.

`IEnumerable CreateSchemaColumnExpressionsWithoutCoalesce((params string[] columnPaths))`

Returns the collection of column expressions of the entity schema query by the specified array of paths to columns. If it is a column of the multilookup type, the COALESCE function does not apply to its values.

`static EntitySchemaQueryExpression CreateSchemaColumnQueryExpression((string columnPath, EntitySchema rootSchema, EntitySchemaColumn schemaColumn, bool useDisplayValue))`

`static EntitySchemaQueryExpression CreateSchemaColumnQueryExpression((string columnPath, EntitySchema rootSchema, bool useDisplayValue))`

Returns the expression of the entity schema query by the specified path to column, root schema and schema column instance. You can define which column value type to use in the expression – either the stored value or the displayed value.

`EntitySchemaQueryExpression CreateSubEntitySchemaExpression((string leftExprColumnPath))`

Returns the expression of entity schema subquery for the column located at the specified `leftExprColumnPath` path.

`EntitySchemaAggregationQueryFunction CreateAggregationFunction((AggregationTypeStrict aggregationType, string columnPath))`

Returns the `EntitySchemaAggregationQueryFunction` aggregation function instance with the specified type of aggregation from the `Terrasoft.Common.AggregationTypeStrict` enumeration for the column at the specified `columnPath` path in relation to the root schema.

`EntitySchemaCaseNotNullQueryFunction CreateCaseNotNullFunction((params EntitySchemaCaseNotNullQueryFunctionWhenItem[] whenItems))`

Returns the instance of the CASE `EntitySchemaCaseNotNullQueryFunction` function for the specified `EntitySchemaCaseNotNullQueryFunctionWhenItem` array of condition expressions.

`EntitySchemaCaseNotNullQueryFunctionWhenItem CreateCaseNotNullQueryFunctionWhenItem((string whenColumnPath, object thenParameterValue))`

Returns an expression instance for the sql construct of the WHEN <Expression_1> IS NOT NULL THEN <Expression_2> view, where:

- `whenColumnPath` – path to the column that contains the expression of the WHEN clause;
- `thenParameterValue` – path to the column that contains the expression of the THEN clause.

`EntitySchemaCastQueryFunction CreateCastFunction((string columnPath, DBValueType castType))`

Returns an instance of the CAST `EntitySchemaCastQueryFunction` function for the column expression located at the specified `columnPath` path relative to the root schema and the specified `DBValueType` target data type.

`EntitySchemaCoalesceQueryFunction CreateCoalesceFunction((params string[] columnPaths))`

`static EntitySchemaCoalesceQueryFunction CreateCoalesceFunction((EntitySchemaQuery parentQuery, EntitySchema rootSchema, params string[] columnPaths))`

`static EntitySchemaCoalesceQueryFunction CreateCoalesceFunction((EntitySchema rootSchema, params string[] columnPaths))`

Returns the function instance returning the first expression that is other than `null` from the list of arguments for the specified columns.

Parameters:

- `columnPaths` – array of paths to columns in relation to the root schema;
- `parentQuery` – query to the entity schema, for which the function instance is created;
- `rootSchema` – the root schema.

`EntitySchemaConcatQueryFunction CreateConcatFunction((params EntitySchemaQueryExpression[] expressions))`

Returns a function instance for generating a string that is the result of combining the string values of function arguments for the specified `EntitySchemaQueryExpression` array of expressions.

`EntitySchemaDatePartQueryFunction CreateDatePartFunction((EntitySchemaDatePartQueryFunctionInterval interval, string columnPath))`

Returns the DATEPART instance of the `EntitySchemaDatePartQueryFunction` function that determines the date

interval specified by the *EntitySchemaDatePartQueryFunctionInterval* enumeration (month, day, hour, year, week day...) for the value of column located at the specified path in relation to the root schema.

EntitySchemaDatePartQueryFunction CreateDayFunction((string columnPath))

Returns an instance of the *EntitySchemaDatePartQueryFunction* function that determines the [Day] date range for a column value located at the specified path in relation to the root schema.

EntitySchemaDatePartQueryFunction CreateHourFunction((string columnPath))

Returns an instance of the *EntitySchemaDatePartQueryFunction* function that returns a part of the [Hour] date for a column value located at the specified path in relation to the root schema.

EntitySchemaDatePartQueryFunction CreateHourMinuteFunction((string columnPath))

Returns an instance of the *EntitySchemaDatePartQueryFunction* function that returns a part of the [Minute] date for a column value located at the specified path in relation to the root schema.

EntitySchemaDatePartQueryFunction CreateMonthFunction((string columnPath))

Returns an instance of the *EntitySchemaDatePartQueryFunction* function that returns a part of the [Month] date for a column value located at the specified path in relation to the root schema.

EntitySchemaDatePartQueryFunction CreateWeekdayFunction((string columnPath))

Returns an instance of the *EntitySchemaDatePartQueryFunction* function that returns a part of the [Week day] date for a column value located at the specified path in relation to the root schema.

EntitySchemaDatePartQueryFunction CreateWeekFunction((string columnPath))

Returns an instance of the *EntitySchemaDatePartQueryFunction* function that returns a part of the [Week] date for a column value located at the specified path in relation to the root schema.

EntitySchemaDatePartQueryFunction CreateYearFunction((string columnPath))

Returns an instance of the *EntitySchemaDatePartQueryFunction* function that returns a part of the [Year] date for a column value located at the specified path in relation to the root schema.

EntitySchemaIsNullQueryFunction CreateIsNullFunction((string checkColumnPath, string replacementColumnPath))

Returns the instance of the *EntitySchemaIsNullQueryFunction* function for columns with values to check and substitute located at specified paths in relation to the root schema.

EntitySchemaLengthQueryFunction CreateLengthFunction((string columnPath))

EntitySchemaLengthQueryFunction CreateLengthFunction((params EntitySchemaQueryExpression[] expressions))

Creating an instance of the LEN function (the function for returning the length of expression) for the column expression at the specified path in relation to the root schema or for the specified expression array.

EntitySchemaTrimQueryFunction CreateTrimFunction((string columnPath))

EntitySchemaTrimQueryFunction CreateTrimFunction((params EntitySchemaQueryExpression[] expressions))

Creating an instance of the TRIM function (the function for removing all leading and trailing whitespace from expression) for the column expression at the specified path in relation to the root schema or for the specified expression array.

EntitySchemaUpperQueryFunction CreateUpperFunction((string columnPath))

Returns an instance of the *EntitySchemaUpperQueryFunction* function that converts all the argument expression characters into upper case. The column expression located at the specified path relative to the root schema is specified as the argument.

EntitySchemaCurrentdateQueryFunction CreateCurrentdateFunction()

Returns an instance of the *EntitySchemaCurrentDateQueryFunction* function that defines the current date.

EntitySchemaCurrentdateTimeQueryFunction CreateCurrentdateTimeFunction()

Returns an instance of the *EntitySchemaCurrentDateTimeQueryFunction* function that returns the current date and time.

EntitySchemaCurrentTimeQueryFunction CreateCurrentTimeFunction()

Returns the instance of the *EntitySchemaCurrentTimeQueryFunction* function that defines the current time.

EntitySchemaCurrentUserAccountQueryFunction CreateCurrentUserAccountFunction()

Returns the instance of the *EntitySchemaCurrentUserAccountQueryFunction* function that defines the account Id of the current user.

EntitySchemaCurrentUserContactQueryFunction CreateCurrentUserContactFunction()

Returns the instance of the *EntitySchemaCurrentUserContactQueryFunction* function that defines the contact Id of the current user.

EntitySchemaCurrentUserQueryFunction CreateCurrentUserFunction()

Returns the instance of the *EntitySchemaCurrentUserQueryFunction* function that defines the current user.

EntitySchemaQueryFilter CreateExistsFilter((string rightExpressionColumnPath))

Creates the [Exists by the specified condition] type comparison filter and sets the column expression located at the *rightExpressionColumnPath* specified path as the testing value.

```
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType, string leftExpressionColumnPath, params string[] rightExpressionColumnPaths))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType, string leftExpressionColumnPath, EntitySchemaQueryExpression rightExpression))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType, string leftExpressionColumnPath, EntitySchemaQueryFunction rightExpressionValue))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType,
EntitySchemaQueryExpression leftExpression, EntitySchemaQueryMacrosType macrosType, int rightValue))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType,
EntitySchemaQueryExpression leftExpression, EntitySchemaQueryMacrosType macrosType, DateTime
rightValue))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType,
EntitySchemaQueryExpression leftExpression, EntitySchemaQueryMacrosType macrosType, DayOfWeek
rightValue))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType, string leftExpressionColumnPath, EntitySchemaQueryMacrosType macrosType, int rightValue))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType, string leftExpressionColumnPath, EntitySchemaQueryMacrosType macrosType, DateTime rightValue))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType, string leftExpressionColumnPath, EntitySchemaQueryMacrosType macrosType, DayOfWeek rightValue))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType, string leftExpressionColumnPath, EntitySchemaQuery rightExpressionValue))
EntitySchemaQueryFilter CreateFilter((FilterComparisonType comparisonType, string leftExprColumnPath,
AggregationTypeStrict leftExprAggregationType, int rightExprParameterValue))
EntitySchemaQueryFilter CreateFilter((FilterComparisonType comparisonType, string leftExprColumnPath,
AggregationTypeStrict leftExprAggregationType, double rightExprParameterValue))
EntitySchemaQueryFilter CreateFilter((FilterComparisonType comparisonType, string leftExprColumnPath,
AggregationTypeStrict leftExprAggregationType, DateTime rightExprParameterValue))
EntitySchemaQueryFilter CreateFilter((FilterComparisonType comparisonType, string leftExprColumnPath,
AggregationTypeStrict leftExprAggregationType, string rightExprParameterValue))
EntitySchemaQueryFilter CreateFilter((FilterComparisonType comparisonType, string leftExprColumnPath,
AggregationTypeStrict leftExprAggregationType, object rightExprParameterValue, out EntitySchemaQuery
leftExprSubQuery))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType, string
leftExprColumnPath, AggregationTypeStrict leftExprAggregationType, EntitySchemaQueryMacrosType
macrosType, int daysCount))
IEntitySchemaQueryFilterItem CreateFilter((FilterComparisonType comparisonType, string
leftExprColumnPath, AggregationTypeStrict leftExprAggregationType, EntitySchemaQueryMacrosType
macrosType, out EntitySchemaQuery leftExprSubQuery, int daysCount))
```

Creates a query filter for selecting records according to specified conditions.

Parameters:

- *comparisonType* – comparison type from the *Terrasoft.Core.Entities.FilterComparisonType* enumeration;

- *leftExpressionColumnPath* – path to the column that contains the left side expression of the filter;
- *leftExpression* – expression on the left side of the filter;
- *leftExprAggregationType* – the type of aggregating function;
- *leftExprSubQuery* – parameter that returns a subquery for the left side expression of the filter (if it is not equal to *null*) or a subquery for the first expression on the right side of the filter (if the left-side expression of the filter is equal to *null*);
- *rightExpressionColumnPaths* – array of column paths that contain the right side expressions of the filter;
- *rightExpression* – expression on the right side of the filter;
- *rightExpressionValue* – instance of the expression function on the right side of the filter (the *EntitySchemaQueryFunction* parameter type) or a subquery expression on the right side of the filter (the *EntitySchemaQuery* parameter type);
- *rightValue* – value processed by the macro in the right side of the filter;
- *rightExprParameterValue* – parameter value, to which the aggregate function on the right side of the filter is applied;
- *macrosType* – a type of macro from the *Terrasoft.Core.Entities.EntitySchemaQueryMacrosType* enumeration;
- *daysCount* – value to which the macro from the right side of the filter is applied. Optional parameter, default value is 0.

```
IEntitySchemaQueryFilterItem CreateFilterWithParameters((FilterComparisonType comparisonType, bool
useDisplayValue, string leftExpressionColumnPath, params object[] rightExpressionParameterValues))
IEntitySchemaQueryFilterItem CreateFilterWithParameters((FilterComparisonType comparisonType, string
leftExpressionColumnPath, params object[] rightExpressionParameterValues))
IEntitySchemaQueryFilterItem CreateFilterWithParameters((FilterComparisonType comparisonType, string
leftExpressionColumnPath, IEnumerable<object> rightExpressionParameterValues, bool useDisplayValue))
static IEntitySchemaQueryFilterItem CreateFilterWithParameters((EntitySchemaQuery parentQuery,
EntitySchema rootSchema, FilterComparisonType comparisonType, bool useDisplayValue, string
leftExpressionColumnPath, params object[] rightExpressionParameterValues))
static IEntitySchemaQueryFilterItem CreateFilterWithParameters((EntitySchema rootSchema,
FilterComparisonType comparisonType, bool useDisplayValue, string leftExpressionColumnPath, params object[]
rightExpressionParameterValues))
```

Creates a parameterized filter for selecting records matching certain conditions.

Parameters:

- *parentQuery* – parent query for which the filter is being created;
- *rootSchema* – the root schema;
- *comparisonType* – comparison type from the *Terrasoft.Core.Entities.FilterComparisonType* enumeration;
- *useDisplayValue* – defines the column value type that is used in the filter: *true* – for the displayed value; *false* – for the stored value;
- *leftExpressionColumnPath* – path to the column that contains the left side expression of the filter;
- *rightExpressionParameterValues* – collection of parameter expressions on the right side of the filter.

```
IEntitySchemaQueryFilterItem CreateIsNotNullFilter((string leftExpressionColumnPath))
```

Creates the comparison filter of the [Is not “null” in database] type and, as the tested value, sets the column expression located at the path specified in the *leftExpressionColumnPath* parameter.

```
IEntitySchemaQueryFilterItem CreateIsNullFilter((string leftExpressionColumnPath))
```

Creates the comparison filter of the [Is “null” in database] type and, as the testing condition, sets the column expression located at the path specified in the *leftExpressionColumnPath* parameter.

```
EntitySchemaQueryFilter CreateNotExistsFilter((string rightExpressionColumnPath))
```

Creates the [Does not exist by the specified condition] type comparison filter and sets the column expression located at the *rightExpressionColumnPath* specified path as the tested value.

```
DataTable GetdataTable((UserConnection userConnection))
```

Returns the result of the current query execution to the object schema as a data table in memory using *UserConnection*.

`static int GeddayOfWeekNumber((UserConnection userConnection, DayOfWeek dayOfWeek))`

Returns the sequence number of the week day for the `System.DayOfWeek` entity taking into account local settings.

`Entity GetEntity((UserConnection userConnection, object primaryColumnName))`

Returns an Entity instance by the `primaryColumnName` primary key using `UserConnection`.

`EntityCollection GetEntityCollection((UserConnection userConnection, EntitySchemaQueryOptions options))`

`EntityCollection GetEntityCollection((UserConnection userConnection))`

Returns the `Entity` instance collection representing the results of executing the current query using `UserConnection` and the specified additional `EntitySchemaQueryOptions` query settings.

`EntitySchema GetSchema()`

Returns the `EntitySchema` entity schema instance of the current `EntitySchemaQuery` instance.

`Select GetSelectQuery((UserConnection userConnection))`

`Select GetSelectQuery((UserConnection userConnection, EntitySchemaQueryOptions options))`

Returns a data selection query instance using `UserConnection` and the specified additional settings of the `EntitySchemaQueryOptions` query.

`EntitySchemaQueryColumnCollection GetSummaryColumns()`

`EntitySchemaQueryColumnCollection GetSummaryColumns((IEnumerable<string> columnNames))`

Returns the collection of expressions of the query columns for which total values are calculated.

`Entity GetSummaryEntity((UserConnection userConnection, EntitySchemaQueryColumnCollection summaryColumns))`

`Entity GetSummaryEntity((UserConnection userConnection))`

`Entity GetSummaryEntity((UserConnection userConnection, IEnumerable<string> columnNames))`

`Entity GetSummaryEntity((UserConnection userConnection, params string[] columnNames))`

Returns an `Entity` instance for the result returned by the total value selection query.

Parameters:

- `userConnection` – system user connection;
- `summaryColumns` – collection of the query columns, whose totals are selected;
- `columnNames` – column name collection.

`Select GetSummarySelectQuery((UserConnection userConnection, EntitySchemaQueryColumnCollection summaryColumns))`

`Select GetSummarySelectQuery((UserConnection userConnection))`

`Select GetSummarySelectQuery((UserConnection userConnection, IEnumerable<string> columnNames))`

`Select GetSummarySelectQuery((UserConnection userConnection, params string[] columnNames))`

Builds a query for the selection of total values for a specified column collection of the `EntitySchemaQuery` current instance.

Parameters:

- `userConnection` – system user connection;
- `summaryColumns` – collection of the query columns, whose totals are selected;
- `columnNames` – column name collection.

`T GetTypedColumnValue((Entity entity, string columnName))`

Returns the typed column value with `columnName` from the `Entity` passed instance.

`void LoadDataTableData((UserConnection userConnection, DataTable dataTable))`

`void LoadDataTableData((UserConnection userConnection, DataTable dataTable, EntitySchemaQueryOptions options))`

Loads the result of executing the current query to the entity schema into the `System.Data.DataTable` entity using `UserConnection` and the `EntitySchemaQueryOptions` specified additional query settings.

`void RemoveColumn((string columnName))`

Removes the `columnName` column from the current query column collection.

`void ResetSchema()`

Clears the schema of the `EntitySchemaQuery` current instance.

`void ResetSelectQuery()`

Clears the select query for the current entity schema query.

`void SetLocalizationCultureId((System.Guid cultureId))`

Sets the identifier of the local culture.

See also

- **Examples of using the EntitySchemaQuery class to retrieve data with the access permissions of the user**

The Insert class

Beginner

Easy

Medium

Advanced

Introduction

The `Terrasoft.Core.DB.Insert` class is used to build queries for adding records in Creatio database tables. As a result of creating and configuring the instance of this class, the INSERT SQL-expression query to the application database will be built. Execution of the query results in returning the number of involved records.

When working with the `Insert` class, the access permissions are not applied to the added records. The user connection is only used for accessing the database table.

The “Terrasoft.Core.DB.Insert“ class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the `Insert` class methods and properties, its parent classes, and the implemented interfaces.

Constructors

`public Entity((UserConnection userConnection))`

Creates a new `Entity` class instance for the set `UserConnection`.

`public Insert((UserConnection userConnection))`

Creates a class instance with the specified `UserConnection` parameter.

`public Insert((Insert source))`

Creates a class instance that is a clone of the instance passed as an argument.

Properties

Table 1. Primary properties of the `Insert` class

`UserConnection`

`Terrasoft.Core.UserConnection`

The current user connection at the moment of executing the query.

`Source`

`Terrasoft.Core.DB.ModifyQuerySource`

Data source.

`Parameters`

`Terrasoft.Core.DB.QueryParameterCollection`

Collection of the query parameters.

HasParameters

bool

Determines whether the query has any parameters.

BuildParametersAsValue

bool

Determines whether the query parameters are to be added as values into the query text.

ColumnValues

Terrasoft.Core.DB.ModifyQueryColumnValueCollection

Collection of values of the query columns.

ColumnValuesCollection

List<ModifyQueryColumnValueCollection>

Collection of column values for adding multiple records (see “**Multi-row data insert. The Insert class**”).

Methods

Table 2. Primary methods of the *Insert* class

string GetSqlText()

Returns the SQL text of the current query.

void BuildSqlText((StringBuilder sb))

Generates the query text via the *StringBuilder* instance.

void ResetCachedSqlText()

Clears the cached text of the query.

QueryParameterCollection GetUsingParameters()

Returns the collection of parameters used by the query.

void ResetParameters()

Clears the collection of the query parameters.

void SetParameterValue((string name, object value))

Sets the value of the query parameter.

Parameters:

- *name* – parameter name;
- *value* – the value.

void InitializeParameters()

Initiates the collection of the query parameters.

int Execute()

Executes the query. Returns the number of records involved by the query.

int Execute((DBExecutor dbExecutor))

Executes the query using the *DBExecutor* instance. Returns the number of records involved by the query.

Insert Into((string schemaName))

Insert Into((ModifyQuerySource source))

Adds the data source to the current query.

Parameters:

- *schemaName* – the schema name;
- *source* – the data source.

`Insert Set((string sourceColumnAlias, Select subSelect))`

`Insert Set((string sourceColumnAlias, Query subSelectQuery))`

`Insert Set((string sourceColumnAlias, QueryColumnExpression columnExpression))`

`Insert Set((string sourceColumnAlias, QueryParameter parameter))`

Adds a SET clause to the current query for assigning the passed expression or parameter to the column. Returns the current *Insert* instance.

Parameters:

- *sourceColumnAlias* – column alias;
- *subSelect* – selection subquery;
- *subSelectQuery* – subquery;
- *columnExpression* – the expression of the column;
- *parameter* – parameter of the query.

`Insert Values()`

Initiates values for adding multiple columns (see “**Multi-row data insert. The “Insert” class**”).

See also

- **Examples of using the Insert class to build queries for data insertion**

The InsertSelect class

[Beginner](#)

[Easy](#)

[Medium](#)

[Advanced](#)

Introduction

The *Terrasoft.Core.DB.InsertSelect* class is used to build queries for adding records in Creatio database tables. The *Terrasoft.Core.DB.Select* class instance is used as the source for adding data (see “**Retrieving information from database. The “Select” class ('Retrieving information from database. The Select class' in the on-line documentation)**”). As a result of creating and configuring the instance of *Terrasoft.Core.DB.InsertSelect*, the INSERT INTO SELECT SQL-expression query to the application database will be built.

When working with the *InsertSelect* class, the access permissions are not applied to the added records. No application permissions are applied to such records (including object permissions by operation, records, or columns). The user connection is only used for accessing the database table.

After the *InsertSelect* query is executed, the database will be complemented with all records returned in its *Select* subquery.

The “Terrasoft.Core.DB.InsertSelect” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *InsertSelect* class.

Constructors

`public InsertSelect((UserConnection userConnection))`

Creates a class instance with the specified *UserConnection* parameter.

`public InsertSelect((InsertSelect source))`

Creates a class instance that is a clone of the instance passed as an argument.

Properties

Table 1. Primary properties of the class

UserConnection

Terrasoft.Core.UserConnection

The current user connection at the moment of executing the query.

Source

Terrasoft.Core.DB.ModifyQuerySource

Data source.

Parameters

Terrasoft.Core.DB.QueryParameterCollection

Collection of the query parameters.

HasParameters

bool

Determines whether the query has any parameters.

BuildParametersAsValue

bool

Determines whether the query parameters are to be added as values into the query text.

Columns:

Terrasoft.Core.DB.ModifyQueryColumnValueCollection

Collection of values of the query columns.

Select.

Terrasoft.Core.DB.Select

The *Terrasoft.Core.DB.Select* instance used in the query.

Methods

Table 2. Primary methods of the class

string GetSqlText()

Returns the SQL text of the current query.

void BuildSqlText((SdlingBuilder sb))

Generates the query text via the *StringBuilder* instance.

void ResetCachedSqlText()

Clears the cached text of the query.

QueryParameterCollection GetUsingParameters()

Returns the collection of parameters used by the query.

void ResetParameters()

Clears the collection of the query parameters.

void SetParameterValue((sdling name, object value))

Sets the value of the query parameter.

Parameters:

- *name* – parameter name;
- *value* – the value.

void InitializeParameters()

Initiates the collection of the query parameters.

int Execute()

Executes the query. Returns the number of records involved by the query.

```
int Execute((DBExecutor dbExecutor))
```

Executes the query using the *DBExecutor* instance. Returns the number of records involved by the query.

```
InsertSelect Into((sdling schemaName))
```

```
InsertSelect Into((ModifyQuerySource source))
```

Adds the data source to the current query.

Parameters:

- *schemaName* – the schema name;
- *source* – the data source.

```
InsertSelect Set((IEnumerable<sdling> sourceColumnAliases))
```

```
InsertSelect Set((params sdling[] sourceColumnAliases))
```

```
InsertSelect Set((IEnumerable<ModifyQueryColumn> columns))
```

```
InsertSelect Set((params ModifyQueryColumn[] columns))
```

Adds a set of columns to the current query, where the values will be added via the subquery. Returns the current *InsertSelect* instance.

Parameters:

- *sourceColumnAliases* – method parameter collection or array containing the column aliases;
- *columns* – method parameter collection or array containing the column instances.

```
InsertSelect FromSelect((Select subSelect))
```

```
InsertSelect FromSelect((Query subSelectQuery))
```

Adds the SELECT clause to the current query.

Parameters:

- *subSelect* – selection subquery;
- *subSelectQuery* – subquery.

See also

- [Examples of adding data via subqueries](#)

The Update class

Beginner

Easy

Medium

Advanced

Introduction

The *Terrasoft.Core.DB.Update* class is used to build queries for modifying records in Creatio database tables. As a result of creating and configuring the instance of this class, the UPDATE SQL-expression query to the application database will be built.

The “Terrasoft.Core.DB.Update” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods and properties, parent classes, and implemented interfaces of the *Update* class.

Constructors

```
public Update((UserConnection userConnection))
```

Creates a class instance using *UserConnection*.

```
public Update((UserConnection userConnection, string schemaName))
```

Creates a class instance for the specified schema using *UserConnection*.

```
public Update((UserConnection userConnection, ModifyQuerySource source))
```

Creates a class instance for the specified data source using *UserConnection*.

`public Update((Insert source))`

Creates a class instance that is a clone of the instance passed as an argument.

Properties

Table 1. Primary properties of the class

`UserConnection`

`Terrasoft.Core.UserConnection`

The current user connection at the moment of executing the query.

`Condition`

`Terrasoft.Core.DB.QueryCondition`

Condition of *Where* expression in the query.

`HasCondition`

`bool`

Determines whether *Where* expression is available in the query.

`Source.`

`Terrasoft.Core.DB.ModifyQuerySource`

The query data source.

`ColumnValues`

`Terrasoft.Core.DB.ModifyQueryColumnValueCollection`

Collection of values of the query columns.

Methods

Table 2. Primary methods of the class

`string GetSqlText()`

Returns the SQL text of the current query.

`void BuildSqlText((StringBuilder sb))`

Generates the query text via the *StringBuilder* instance.

`void ResetCachedSqlText()`

Clears the cached text of the query.

`QueryParameterCollection GetUsingParameters()`

Returns the collection of parameters used by the query.

`int Execute()`

Executes the query. Returns the number of records involved by the query.

`int Execute(DBExecutor dbExecutor)`

Executes the query using the *DBExecutor* instance. Returns the number of records involved by the query.

`QueryCondition Where()`

`QueryCondition Where(string sourceColumnAlias)`

`QueryCondition Where(string sourceAlias, string sourceColumnAlias)`

`QueryCondition Where(Select subSelect)`

`QueryCondition Where(Query subSelectQuery)`

`QueryCondition Where(QueryColumnExpression columnExpression)`

`Query Where(QueryCondition condition)`

Adds the initial condition to the current query.

Parameters:

- *sourceColumnAlias* – an alias of the column for which the condition is added;
- *sourceAlias* – the alias of the source;
- *subSelect* – a subquery of the data selection, for whose results the condition is added;
- *subSelectQuery* – the subquery, for whose results the condition is added;
- *columnExpression* – the expression, for whose results the condition is added;
- *condition* – the query condition.

```
QueryCondition And()
QueryCondition And((string sourceColumnAlias))
QueryCondition And((string sourceAlias, string sourceColumnAlias))
QueryCondition And((Select subSelect))
QueryCondition And((Query subSelectQuery))
QueryCondition And((QueryParameter parameter))
QueryCondition And((QueryColumnExpression columnExpression))
Query And((QueryCondition condition))
```

Adds the condition (predicate) to the current query condition using the AND logical operation.

Parameters:

- *sourceColumnAlias* – an alias of the column, for which the predicate is added;
- *sourceAlias* – the alias of the source;
- *subSelect* – the data selection subquery used as a predicate;
- *subSelectQuery* – the subquery used as a predicate;
- *parameter* – parameter that the predicate is added to;
- *columnExpression* – the expression used as a predicate;
- *condition* – the query condition.

```
QueryCondition Or()
QueryCondition Or((string sourceColumnAlias))
QueryCondition Or((string sourceAlias, string sourceColumnAlias))
QueryCondition Or((Select subSelect))
QueryCondition Or((Query subSelectQuery))
QueryCondition Or((QueryParameter parameter))
QueryCondition Or((QueryColumnExpression columnExpression))
Query Or((QueryCondition condition))
```

Adds the condition (predicate) to the current query condition using the OR logical operation.

Parameters:

- *sourceColumnAlias* – an alias of the column, for which the predicate is added;
- *sourceAlias* – the alias of the source;
- *subSelect* – the data selection subquery used as a predicate;
- *subSelectQuery* – the subquery used as a predicate;
- *parameter* – parameter that the predicate is added to;
- *columnExpression* – the expression used as a predicate;
- *condition* – the query condition.

```
Update Set((string sourceColumnAlias, Select subSelect))
Update Set((string sourceColumnAlias, Query subSelectQuery))
Update Set((string sourceColumnAlias, QueryColumnExpression columnExpression))
Update Set((string sourceColumnAlias, QueryParameter parameter))
```

Adds a SET clause to the current query for assigning the passed expression or parameter to the column. Returns the current *Update* instance.

Parameters:

- *sourceColumnAlias* – column alias;
- *subSelect* – selection subquery;
- *subSelectQuery* – subquery;
- *columnExpression* – expression of the column;

- *parameter* – parameter of the query.

See also

- **Example of using the Update class to modify data**

The Delete class

Beginner

Easy

Medium

Advanced

Introduction

The *Terrasoft.Core.DB.Delete* class is used to build queries for deleting records in Creatio database tables. As a result of creating and configuring the instance of this class, the DELETE SQL-expression query to the application database will be built.

The “Terrasoft.Core.DB.Delete” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *Delete* class.

Constructors

`public Delete((UserConnection userConnection))`

Creates a class instance using *UserConnection*.

`public Delete((Delete source))`

Creates a class instance that is a clone of the instance passed as an argument.

Properties

Table 1. Primary properties of the class

`UserConnection`

Terrasoft.Core.UserConnection

The current user connection at the moment of executing the query.

`Condition`

Terrasoft.Core.DB.QueryCondition

Condition of *Where* expression in the query.

`HasCondition`

`bool`

Determines whether *Where* expression is available in the query.

`Source.`

Terrasoft.Core.DB.ModifyQuerySource

The query data source.

Methods

Table 2. Primary methods of the class

`string GetSqlText()`

Returns the SQL text of the current query.

`void BuildSqlText((StringBuilder sb))`

Generates the query text via the *StringBuilder* instance.

```
void ResetCachedSqlText()
```

Clears the cached text of the query.

```
QueryParameterCollection GetUsingParameters()
```

Returns the collection of parameters used by the query.

```
int Execute()
```

Executes the query. Returns the number of records involved by the query.

```
int Execute(DBExecutor dbExecutor)
```

Executes the query using the *DBExecutor* instance. Returns the number of records involved by the query.

```
QueryCondition Where()
```

```
QueryCondition Where((string sourceColumnAlias))
```

```
QueryCondition Where((string sourceAlias, string sourceColumnAlias))
```

```
QueryCondition Where((Select subSelect))
```

```
QueryCondition Where((Query subSelectQuery))
```

```
QueryCondition Where((QueryColumnExpression columnExpression))
```

```
Query Where((QueryCondition condition))
```

Adds the initial condition to the current query.

Parameters:

- *sourceColumnAlias* – an alias of the column for which the condition is added;
- *sourceAlias* – the alias of the source;
- *subSelect* – a subquery of the data selection, for whose results the condition is added;
- *subSelectQuery* – the subquery, for whose results the condition is added;
- *columnExpression* – the expression, for whose results the condition is added;
- *condition* – the query condition.

```
QueryCondition And()
```

```
QueryCondition And((string sourceColumnAlias))
```

```
QueryCondition And((string sourceAlias, string sourceColumnAlias))
```

```
QueryCondition And((Select subSelect))
```

```
QueryCondition And((Query subSelectQuery))
```

```
QueryCondition And((QueryParameter parameter))
```

```
QueryCondition And((QueryColumnExpression columnExpression))
```

```
Query And((QueryCondition condition))
```

Adds the condition (predicate) to the current query condition using the AND logical operation.

Parameters:

- *sourceColumnAlias* – an alias of the column, for which the predicate is added;
- *sourceAlias* – the alias of the source;
- *subSelect* – the data selection subquery used as a predicate;
- *subSelectQuery* – the subquery used as a predicate;
- *parameter* – parameter that the predicate is added to;
- *columnExpression* – the expression used as a predicate;
- *condition* – the query condition.

```
QueryCondition Or()
```

```
QueryCondition Or((string sourceColumnAlias))
```

```
QueryCondition Or((string sourceAlias, string sourceColumnAlias))
```

```
QueryCondition Or((Select subSelect))
```

```
QueryCondition Or((Query subSelectQuery))
```

```
QueryCondition Or((QueryParameter parameter))
```

```
QueryCondition Or((QueryColumnExpression columnExpression))
```

```
Query Or((QueryCondition condition))
```

Adds the condition (predicate) to the current query condition using the OR logical operation.

Parameters:

- *sourceColumnAlias* – an alias of the column, for which the predicate is added;
- *sourceAlias* – the alias of the source;
- *subSelect* – the data selection subquery used as a predicate;
- *subSelectQuery* – the subquery used as a predicate;
- *parameter* – parameter that the predicate is added to;
- *columnExpression* – the expression used as a predicate;
- *condition* – the query condition.

```
Delete From((string schemaName))
```

```
Delete From(((ModifyQuerySource source)))
```

Adds the data source to the current query. Returns the current *Delete* instance.

Parameters:

- *schemaName* – schema name (tables, views);
- *source* – the data source.

See also

- **Examples of using the Delete class to erase data**

The Entity class

Beginner

Easy

Medium

Advanced

Introduction

The *Terrasoft.Core.Entities.Entity* class is designed to provide access to an object that represents a record in the database table.

The “Terrasoft.Core.Entities.Entity“ class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *Entity* class.

Constructors

```
public Entity((UserConnection userConnection))
```

Creates a new *Entity* class instance for the set *UserConnection*.

```
public Entity((UserConnection userConnection, Guid schemaUid))
```

Creates a new *Entity* class instance for the set *UserConnection* and the schema set by the *schemaUid* identifier.

```
public Entity((Entity source))
```

Creates a class instance that is a clone of the instance passed as an argument.

Properties

Table 1. Primary properties of the *Entity* class

ChangeType

EntityChangeType

The type of changing the entity status (added, modified, deleted, unchanged).

```
EntitySchemaManager
```

EntitySchemaManager

An instance of the entity schema manager.

EntitySchemaManagerName
string

Name of the entity schema manager.

HasColumnValues
bool

Determines whether an object has at least one column.

HierarchyColumnName
Guid

Value of column of the parent record relationship for hierarchical objects.

InstanceUID
Guid

Object instance identifier.

IsDeletedFromDB
bool

Determines whether the object is deleted from the database.

IsInColumnValueChanged
bool

Determines whether the handling of the *ColumnValueChanged* event is performed.

IsInColumnValueChanging
bool

Determines whether the handling of the *ColumnValueChanging* event is performed.

IsInDefColumnValuesSet
bool

Determines whether the handling of the *DefColumnValuesSet* event is performed.

IsInDeleted
bool

Determines whether the handling of the *Deleted* event is performed.

IsInDeleting
bool

Determines whether the handling of the *Deleting* event is performed.

IsInInserted
bool

Determines whether the handling of the *Inserted* event is performed.

IsInInserting
bool

Determines whether the handling of the *Inserting* event is performed.

IsInLoaded
bool

Determines whether the handling of the *Loaded* event is performed.

IsInLoading
bool

Determines whether the handling of the *Loading* event is performed.

IsInSaved
bool

Determines whether the handling of the *Saved* event is performed.

IsInSaveError
bool

Determines whether the handling of the *SaveError* event is performed.

IsInSaving
bool

Determines whether the handling of the *Saving* event is performed.

IsInUpdated
bool

Determines whether the handling of the *Updated* event is performed.

IsInUpdating
bool

Determines whether the handling of the *Updating* event is performed.

IsInValidating
bool

Determines whether the handling of the *Validating* event is performed.

IsSchemaInitialized
bool

Determines whether the entity schema is initialized.

LicOperationPrefix
string

The prefix of the operation that is being licensed.

LoadState
EntityLoadState

State of the object loading.

PrimaryColumnName
Guid

Initial column identifier.

PrimaryDisplayColumnName
string

Initial column value for displaying.

Process
Process

The embedded process of an object.

Schema
EntitySchema

An instance of the entity schema.

SchemaName
string

Object schema name.

StoringState
StoringObjectState

Object status (modified, added, deleted, unchanged).

UseAdminRights
bool

Determines whether permissions will be taken into account when inserting, updating, deleting and receiving data.

UseDefRights
bool

Determines whether the default object permissions should be used.

UseLazyLoad
bool

Determines whether to use the initial lazy loading of object data.

UserConnection
UserConnection

User connection.

ValidationMessages
EntityValidationMessageCollection

Collection of the messages output when validating an object.

ValueListSchemaManager
ValueListSchemaManager

An instance of the object enumerations manager.

ValueListSchemaManagerName
string

Name of the manager for object enumerations.

Methods

Table 2. Primary methods of the *Entity* class

void AddDefRights()
void AddDefRights((Guid primaryColumnValue))
void AddDefRights((IEnumerable<Guid> primaryColumnValues))
Sets the default permissions for the given object.

virtual object Clone()
Creates a clone of the current *Entity* instance.

Insert CreateInsert((bool skipLookupColumnValues))
Creates a query to insert data.

Update CreateUpdate((bool skipLookupColumnValues))
Creates a query to update data.

virtual bool Delete()
virtual bool Delete((object keyValue))

Deletes the object record from the database. The *keyValue* parameter determines the initial key of a record.

`bool DeleteWithCancelProcess()`

Deletes the object record from the database and cancels the launched process.

`static Entity DeserializeFromJson((UserConnection userConnection, string jsonValue))`

Creates an *Entity* type object using *userConnection* and populates the field values from the specified string of the JSON *jsonValue* format.

`bool ExistInDB((EntitySchemaColumn conditionColumn, object conditionValue))`

`bool ExistInDB((string conditionColumnName, object conditionValue))`

`bool ExistInDB((object keyValue))`

`bool ExistInDB((Dictionary<string,object> conditions))`

Determines whether a record matching the given condition of the *conditionValue* query to the *conditionColumn* object schema column or with the specified *keyValue* initial key exists in the database.

`bool FetchFromDB((EntitySchemaColumn conditionColumn, object conditionValue, bool useDisplayValues))`

`bool FetchFromDB((string conditionColumnName, object conditionValue, bool useDisplayValues))`

`bool FetchFromDB((object keyValue, bool useDisplayValues))`

`bool FetchFromDB((Dictionary<string,object> conditions, bool useDisplayValues))`

`bool FetchFromDB((EntitySchemaColumn conditionColumn, object conditionValue,`

`IEnumerable<EntitySchemaColumn> columnsToFetch, bool useDisplayValues))`

`bool FetchFromDB((string conditionColumnName, object conditionValue,`

`IEnumerable<string> columnNamesToFetch, bool useDisplayValues))`

By the specified condition, loads the object from the database.

Parameters:

- *conditionColumn* – the column, for which the selection condition is specified;
- *conditionColumnName* – the name of the column, for which the selection condition is specified;
- *conditionValue* – the value of the condition column for the selected data;
- *columnsToFetch* – the list of columns to be selected;
- *columnNamesToFetch* – the list of column names to be selected;
- *conditions* – set of conditions for filtering the selection of object records;
- *keyValue* – the key field value;
- *useDisplayValues* – indicates that the query returns the primary display values. If the parameter is *true*, the query will return the primary display values.

`bool FetchPrimaryColumnFromDB((object keyValue))`

By the set condition, *keyValue* loads the object with the initial column from the database.

`bool FetchPrimaryInfoFromDB((EntitySchemaColumn conditionColumn, object conditionValue))`

`bool FetchPrimaryInfoFromDB((string conditionColumnName, object conditionValue))`

By the set condition, loads an object with initial columns including the initial display column from the database.

`byte[] GetBytesValue((string valueName))`

Returns the value of the specified object column as a byte array.

`IEnumerable<EntityColumnValue> GetChangedColumnValues()`

Returns the name collection of the object properties that have been modified.

`string GetColumnDisplayValue((EntitySchemaColumn column))`

Returns the value for display of the object property that matches the specified column of the entity schema.

`object GetColumnOldValue((string valueName))`

`object GetColumnOldValue((EntitySchemaColumn column))`

Returns the previous value of the specified object property.

`virtual object GetColumnValue((string valueName))`

`virtual object GetColumnValue((EntitySchemaColumn column))`

Returns the value of the object column with the specified name that matches the passed column of the object schema.

`IEnumerable<string> GetColumnValueNames()`

Returns the collection of object column names.

`virtual bool GetIsColumnValueLoaded((string columnName))`

`bool GetIsColumnValueLoaded((EntitySchemaColumn column))`

Returns whether the specified property of an object is loaded.

`virtual MemoryStream GetStreamValue((string columnName))`

Returns the value of the passed object schema column converted into the `System.IO.MemoryStream` type instance.

`virtual TResult GetTypedColumnValue<TResult>((string columnName))`

`TResult GetTypedColumnValue<TResult>((EntitySchemaColumn column))`

Returns the typed value of the object property that matches the specified column of the entity schema.

`TResult GetTypedOldColumnValue<TResult>((string columnName))`

`TResult GetTypedOldColumnValue<TResult>((EntitySchemaColumn column))`

Returns the typed previous value of the entity property that matches the specified column of the entity schema.

`virtual bool InsertToDB((bool skipLookupColumnValues, bool validateRequired))`

Adds an entry of the current object to the database taking into account the passed parameters:

- `skipLookupColumnValues` – parameter that determines whether the columns of the lookup type are to be added to the database. If the parameter is `true`, the columns of the lookup type will not be added to the base. Default value – `false`;
- `validateRequired` – parameter that determines the necessity of validating the required values. Default value – `true`.

`bool IsColumnValueLoaded((string columnName))`

`bool IsColumnValueLoaded((EntitySchemaColumn column))`

Determines whether the value of the object property with the specified name is loaded.

`virtual bool Load((DataRow dataRow))`

`virtual bool Load((DataRow dataRow, Dictionary<string, string> columnMap))`

`virtual bool Load((IDataReader dataReader))`

`virtual bool Load((IDataReader dataReader, IDictionary<string, string> columnMap))`

`virtual bool Load((object dataSource))`

`virtual bool Load((object dataSource, IDictionary<string, string> columnMap))`

Populates the object with the passed data.

Parameters:

- `dataRow` – The `System.Data.DataRow` instance from which the data is loaded to the object;
- `dataRow` – The `System.Data.IDataReader` instance from which the data is loaded to the object;
- `dataSource` – The `System.Object` instance from which the data is loaded to the object;
- `columnMap` – object properties populated with data.

`void LoadColumnValue((string columnName, IDataReader dataReader, int fieldIndex, int binaryPackageSize))`

`void LoadColumnValue((string columnName, IDataReader dataReader, int fieldIndex))`

`void LoadColumnValue((string columnName, object value))`

`void LoadColumnValue((EntitySchemaColumn column, object value))`

Loads the value from the passed instance for the property with the specified name.

Parameters:

- `columnName` – name of the object property;
- `column` – object schema column;
- `dataReader` – the `System.Data.IDataReader` instance from which the property value is loaded;
- `fieldIndex` – index of the field loaded from `System.Data.IDataReader`;

- *binaryPackageSize* – size of the loaded value;
- *Value* – value of the property that is being loaded.

static Entity Read((UserConnection userConnection, DataReader dataReader))

Returns the current property value of the *Entity* type from the output stream.

void ReadData((DataReader reader))

void ReadData((DataReader reader, EntitySchema schema))

Reads data from the object schema and saves them in the specified object of the *System.Data.IDataReader* type.

void ResetColumnValues()

Cancels changes for all object properties.

void ResetOldColumnValues()

Reverts all object properties to previous values.

bool Save((bool validateRequired))

Saves the object to the database. The *validateRequired* parameter determines the necessity of validating the required values. Default value – *true*.

static string SerializeToJson((Entity entity))

Converts the *entity* object into a *JSON* format string.

virtual void SetBytesValue((string valueName, byte[] streamBytes))

Sets the passed value of the *System.Byte* type for the specified object property.

bool SetColumnBothValues((EntitySchemaColumn column, object value, string displayValue))

bool SetColumnBothValues((string columnName, object value, string displayName, string displayValue))

Sets the passed *value* and *displayValue* to the object property matching the specified schema column.

bool SetColumnValue((string valueName, object value))

bool SetColumnValue((EntitySchemaColumn column, object value))

Sets the passed *value* to the specified schema column.

void SeddefColumnValue((string columnName, object defValue))

void SeddefColumnValue((string columnName))

Sets the property with the specified name to the default value.

void SeddefColumnValues()

Sets default values for all object properties.

bool SetStreamValue((string valueName, Stream value))

Sets the passed value of the *System.IO.Stream* type for the specified object property.

virtual bool UpdateInDB((bool validateRequired))

Updates the object record in the database. The *validateRequired* parameter determines the necessity of validating the required values. Default value – *true*.

bool Validate()

Verifies if the required fields are populated.

static void Write((DataWriter dataWriter, Entity entity, string propertyName))

static void Write((DataWriter dataWriter, Entity entity, string propertyName, bool couldConvertForXml))

Records the value of the *Entity* type to the output stream with the specified name.

Parameters:

- *dataWriter* – instance of the *Terrasoft.Common.DataWriter* class that provides methods for sequential recording of values to the output stream;
- *entity* – value for record of the *Entity* type;

- *propertyName* – name of the object;
- *couldConvertForXml* – allows converting for XML-serialization.

`void Write((DataWriter dataWriter, string propertyName))`

Records the value of the *Entity* type to the output stream with the specified name.

`void WriteData((DataWriter writer))`

`void WriteData((DataWriter writer, EntitySchema schema))`

Records to the output stream for the specified object schema.

Events

Table 3. The *Entity* class events

`event EventHandler<EntityColumnEventArgs> ColumnValueChanged`

Event handler used after modifying the value of an entity column.

The event handler receives an argument of the *EntityColumnEventArgs* type.

The *EntityColumnEventArgs* properties that provide information referring to the event:

- *ColumnNameValueName*;
- *DisplayColumnNameValueName*.

`event EventHandler<EntityColumnBeforeEventArgs> ColumnValueChanging`

Event handler used before modifying the value of an entity column.

The event handler receives an argument of the *EntityColumnBeforeEventArgs* type.

The *EntityColumnBeforeEventArgs* properties that provide information referring to the event:

- *ColumnStreamValue*;
- *ColumnValues*;
- *ColumnNameValueName*;
- *DisplayColumnNameValue*;
- *DisplayColumnNameValueName*.

`event EventHandler<EventArgs> DefColumnValuesSet`

Event handler used after setting default values for the object fields.

`event EventHandler<EntityAfterEventArgs> Deleted`

Event handler used after deleting an object.

The event handler receives an argument of the *EntityAfterEventArgs* type.

The *EntityAfterEventArgs* properties that provide information referring to the event:

- *ModifiedColumnValues*;
- *PrimaryColumnNameValue*.

`event EventHandler<EntityBeforeEventArgs> Deleting`

Event handler used before deleting an object.

The event handler receives an argument of the *EntityBeforeEventArgs* type.

The *EntityBeforeEventArgs* properties that provide information referring to the event:

- *AdditionalCondition*;
- *IsCanceled*;
- *KeyValue*.

`event EventHandler<EntityAfterEventArgs> Inserted`

Event handler used after inserting an object.

The event handler receives an argument of the *EntityAfterEventArgs* type.

The *EntityAfterEventArgs* properties that provide information referring to the event:

- *ModifiedColumnValues*;
- *PrimaryColumnName*.

event EventHandler<EntityBeforeEventArgs> Inserting

Event handler used before inserting an object.

The event handler receives an argument of the *EntityBeforeEventArgs* type.

The *EntityBeforeEventArgs* properties that provide information referring to the event:

- *AdditionalCondition*;
- *IsCanceled*;
- *KeyValue*.

event EventHandler<EntityAfterLoadEventArgs> Loaded

Event handler used after loading an object.

The event handler receives an argument of the *EntityAfterLoadEventArgs* type.

The *EntityAfterLoadEventArgs* properties that provide information referring to the event:

- *ColumnMap*;
- *DataSource*.

event EventHandler<EntityBeforeLoadEventArgs> Loading

Event handler used before loading an object.

The event handler receives an argument of the *EntityBeforeLoadEventArgs* type.

The *EntityBeforeLoadEventArgs* properties that provide information referring to the event:

- *ColumnMap*;
- *DataSource*;
- *IsCanceled*.

event EventHandler<EntityAfterEventArgs> Saved

Event handler used after saving an object.

The event handler receives an argument of the *EntityAfterEventArgs* type.

The *EntityAfterEventArgs* properties that provide information referring to the event:

- *ModifiedColumnValues*;
- *PrimaryColumnName*.

event EventHandler<EntitySaveErrorEventArgs> SaveError

Event handler used when an error occurs while saving an object.

The event handler receives an argument of the *EntitySaveErrorEventArgs* type.

The *EntitySaveErrorEventArgs* properties that provide information referring to the event:

- *Exception*;
- *IsHandled*.

event EventHandler<EntityBeforeEventArgs> Saving

Event handler used before saving an object.

The event handler receives an argument of the *EntityBeforeEventArgs* type.

The *EntityBeforeEventArgs* properties that provide information referring to the event:

- *AdditionalCondition*;
- *IsCanceled*;
- *KeyValue*.

event EventHandler<EntityAfterEventArgs> Updated

Event handler used after updating an object.

The event handler receives an argument of the *EntityAfterEventArgs* type.

The *EntityAfterEventArgs* properties that provide information referring to the event:

- *ModifiedColumnValues*;
- *PrimaryColumnName*.

event EventHandler<EntityBeforeEventArgs> Updating

Event handler used before updating an object.

The event handler receives an argument of the *EntityBeforeEventArgs* type.

The *EntityBeforeEventArgs* properties that provide information referring to the event:

- *AdditionalCondition*;
- *IsCanceled*;
- *KeyValue*.

event EventHandler<EntityValidationEventArgs> Validating

Event handler used when validating an object.

The event handler receives an argument of the *EntityValidationEventArgs* type.

The *EntityValidationEventArgs* properties that provide information referring to the event:

- *Messages*.

See also

- **Examples of using the Delete class to erase data**

The EntityMapper class

Beginner

Easy

Medium

Advanced

Introduction

The *Terrasoft.Configuration.EntityMapper* class is a utility configuration class stored in the **[FinAppLending]** package of the Lending product. *EntityMapper* allows to map data of one *Entity* with another using rules defined in the configuration file. Using the approach of mapping the data of different entities avoids the appearance of a monotonous code.

The idea of mapping the data of different entities is implemented in the following classes:

- *EntityMapper* – implements the mapping logic.
- *EntityResult* – defines the resulting type of the mapped entity.
- *MapConfig* – a set of mapping rules.
- *DetailMapConfig* – used to set up a list of mapping rules of the details and entities connected with them.
- *RelationEntityMapConfig* – contains rules for mapping connected entities.
- *EntityFilterMap* – a filter for database query.

The “*Terrasoft.Configuration.EntityMapper*“ class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the *EntityMapper* class methods, its parent classes and the implemented interfaces.

Methods

Table 1. Primary methods of the *EntityMapper* class

virtual EntityResult GetMappedEntity((Guid recId, MapConfig config))

Returns mapped data for two *Entity* objects.

Parameters:

- *recId* – GUID records in the database;
- *config* – an instance of the *MapConfig* class, which is a set of mapping rules.

```
virtual Dictionary<string, object> GetColumnsValues((Guid recordId, MapConfig config, Dictionary<string, object> result))
```

Gets the main entity from the database and matches its columns and values according to the rules specified in the *config* object.

Parameters:

- *recordId* – GUID records in the database;
- *config* – an instance of the *MapConfig* class, which is a set of mapping rules;
- *result* – a dictionary of columns and their values of the mapped entity.

```
virtual Dictionary<string, object> GetRelationEntityColumnsValues((List<RelationEntityMapConfig> relations, Dictionary<string, object> dictionaryToMerge, string columnName, Terrasoft.Nui.ServiceModel.DataContract.LookupColumnValue entitylookup))
```

Gets the related entities from the database and matches them to the main entities.

Parameters:

- *relations* – a list of rules for obtaining related records;
- *dictionaryToMerge* – a dictionary with columns and theirs values;
- *columnName* – name of the parent column;
- *entitylookup* – an object that contains name and Id of the record in the database.

The “Terrasoft.Configuration.EntityResult“ class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the *EntityResult* class properties, its parent classes and the implemented interfaces.

Properties

Table 2. Primary methods of the *EntityResult* class

Columns

```
Dictionary<string, object>
```

A dictionary of main entity column names and their values.

Details

```
Dictionary<string, List<Dictionary<string, object>>>
```

A dictionary of the detail names with the list of their columns and values.

The “Terrasoft.Configuration.MapConfig“ class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the *MapConfig* class properties, its parent classes and the implemented interfaces.

Properties

Table 3. Primary methods of the *MapConfig* class

SourceEntityName

```
string
```

Entity name in the database.

Columns

Dictionary<string, object>

A dictionary with the names of columns of one entity and compared columns of another entity.

DetailsConfig

List<DetailMapConfig>

A list of configuration objects with rules for details.

CleanDetails

List<string>

A list of detail names for cleaning their values.

RelationEntities

List<RelationEntityMapConfig>

List of configuration objects with rules for mapping related records with the main entity.

The “Terrasoft.Configuration.DetailMapConfig“ class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the *DetailMapConfig* class properties, its parent classes and the implemented interfaces.

Properties

Table 4. Primary methods of the *DetailMapConfig* class

DetailName

string

Detail name (The ensure the uniqueness of detail instance).

SourceEntityName

string

Entity name in the database.

Columns

Dictionary<string, object>

A dictionary with the names of columns of one entity and compared columns of another entity.

Filters

List<EntityFilterMap>

A list of configuration objects with filtration rules for more accurate selections from the database.

RelationEntities

List<RelationEntityMapConfig>

List of configuration objects with rules for mapping related records with the main entity.

The “Terrasoft.Configuration.RelationEntityMapConfig“ class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the *RelationEntityMapConfig* class properties, its parent classes and the implemented interfaces.

Properties

Table 5. Primary methods of the *RelationEntityMapConfig* class

ParentColumnName

string

The name of the parent column, which, when found, will trigger the logic for obtaining and mapping the entity data.

SourceEntityName
string

Entity name in the database.

Columns
Dictionary<string, object>

A dictionary with the names of columns of one entity and compared columns of another entity.

Filters
List<EntityFilterMap>

A list of configuration objects with filtration rules to refine selections from the database.

RelationEntities
List<RelationEntityMapConfig>

List of configuration objects with rules for mapping related records with the main entity.

The “Terrasoft.Configuration.EntityFilterMap“ class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the *EntityFilterMap* class properties, its parent classes and the implemented interfaces.

Properties

Table 6. Primary methods of the *EntityFilterMap* class

ColumnName
string

The name of the column, which when found, will start the filtering logic.

Value
object

The value to compare to.

The QueryFunction class

Beginner **Easy** **Medium** **Advanced**

Introduction

The *Terrasoft.Core.DB.QueryFunction* class implements the expression function.

The expression function is implemented in the following classes:

- *QueryFunction* – a base class of the expression function.
- *AggregationQueryFunction* – implements the aggregate function of the expression.
- *IsNullQueryFunction* – replaces *null* values with the replacement experession.
- *CreateGuidQueryFunction* – implements the function for generating a new identifier.
- *CurrentDateTimeQueryFunction* – implements the function for current date and time expression.
- *CoalesceQueryFunction* – returns the first not *null* expression from the list of arguments.
- *DatePartQueryFunction* – implements the function for the date part of the Date/Time type.
- *DateAddQueryFunction* – implements the function for the date expression calculated by adding the specified period to the specified date.
- *DateDiffQueryFunction* – implements the function for calculating the delta between two specified dates.

- *CastQueryFunction* – casts the argument expression to the specified data type.
- *UpperQueryFunction* – converts the argument expression characters to uppercase.
- *CustomQueryFunction* – implements a custom function.
- *DataLengthQueryFunction* – calculates the number of bytes used to represent the expression.
- *TrimQueryFunction* – removes whitespaces from both ends of the expression.
- *LengthQueryFunction* – returns the length of the expression.
- *SubstringQueryFunction* – returns a part of the string.
- *ConcatQueryFunction* – returns a string resulting from merging the string arguments of the function.
- *WindowQueryFunction* – implements an SQL window function.

The “Terrasoft.Core.DB.QueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, parent classes, and implemented interfaces of the *QueryFunction* class.

Methods

Table 1. Primary methods of the *QueryFunction* class

static QueryColumnExpression Negate(QueryFunction operand)

Gets the expression that negates the value of the passed-in function.

Parameters:

- *operand* – the function of the expression.

static QueryColumnExpression operator -(QueryFunction operand)

Overloads the operator of negating the passed-in expression function.

Parameters:

- *operand* – the function of the expression.

static QueryColumnExpression Add(QueryFunction leftOperand, QueryFunction rightOperand)

Gets the arithmetic addition expression of the passed-in expression functions.

Parameters:

- *leftOperand* – the left operand in the addition operation;
- *rightOperand* – the right operand in the addition operation.

static QueryColumnExpression operator /(QueryFunction leftOperand, QueryFunction rightOperand)

Overloads the operator of adding two expression functions.

Parameters:

- *leftOperand* – the left operand in the addition operation;
- *rightOperand* – the right operand in the addition operation.

static QueryColumnExpression Subtract(QueryFunction leftOperand, QueryFunction rightOperand)

Returns the expression of subtraction of the passed-in right expression function from the passed-in left expression function.

Parameters:

- *leftOperand* – the left operand in the subtraction operation;
- *rightOperand* – the right operand in the subtraction operation.

static QueryColumnExpression operator /(QueryFunction leftOperand, QueryFunction rightOperand)

Reloads the operator of subtraction of the passed-in right expression function from the passed-in left expression function.

Parameters:

- *leftOperand* – the left operand in the subtraction operation;

- *rightOperand* – the right operand in the subtraction operation.

static QueryColumnExpression Multiply(QueryFunction leftOperand, QueryFunction rightOperand)
Gets the expression of multiplying the passed-in expression functions.

Parameters:

- *leftOperand* – the left operand in the multiplication operation;
- *rightOperand* – the right operand in the multiplication operation.

static QueryColumnExpression operator /(QueryFunction leftOperand, QueryFunction rightOperand)
Overloads the operator of multiplying two expression functions.

Parameters:

- *leftOperand* – the left operand in the multiplication operation;
- *rightOperand* – the right operand in the multiplication operation.

static QueryColumnExpression Divide(QueryFunction leftOperand, QueryFunction rightOperand)
Returns the expression of dividing the passed-in left expression function by the passed-in right expression function.

Parameters:

- *leftOperand* – the left operand in the division operation;
- *rightOperand* – the right operand in the division operation.

static QueryColumnExpression operator /(QueryFunction leftOperand, QueryFunction rightOperand)
Overloads the operator of dividing two expression functions.

Parameters:

- *leftOperand* – the left operand in the division operation;
- *rightOperand* – the right operand in the division operation.

abstract object Clone()

Creates a copy of the current *QueryFunction* instance.

abstract void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the passed *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder

virtual void AddUsingParameters(QueryParameterCollection resultParameters)

Adds the passed-in collection of parameters in the function arguments.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

QueryColumnExpressionCollection GetQueryColumnExpressions()

Gets the collection of the query column expression for the current query function.

QueryColumnExpression GetQueryColumnExpression()

Gets the query column expression for the current query function.

The “Terrasoft.Core.DB.AggregationQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *AggregationQueryFunction* class.

Constructors

AggregationQueryFunction()

Initializes a new instance of the *AggregationQueryFunction* class.

AggregationQueryFunction(AggregationTypeStrict aggregationType, QueryColumnExpression expression)

Initializes a new *AggregationQueryFunction* instance with the specified type of the aggregate function for the specified column expression.

Parameters:

- *aggregationType* – the type of aggregating function;
- *expression* – the column expression to which the aggregate function is applied.

AggregationQueryFunction(AggregationTypeStrict aggregationType, IQueryColumnExpressionConvertible expression)

Initializes a new *AggregationQueryFunction* instance with the specified type of the aggregate function for the specified column expression.

Parameters:

- *aggregationType* – the type of aggregating function;
- *expression* – the column expression to which the aggregate function is applied.

AggregationQueryFunction((AggregationQueryFunction source))

Initializes a new *AggregationQueryFunction* instance that is a clone of the passed aggregate function of the expression.

Parameters:

- *source* – the *AggregationQueryFunction* expression aggregate function whose clone is being created.

Properties

Table 2. Primary properties of the *AggregationQueryFunction* class

AggregationType

AggregationTypeStrict

The type of the aggregate function.

AggregationEvalType

AggregationEvalType

The scope of the aggregate function.

Expression

QueryColumnExpression

The expression of the function argument.

Methods

Table 3. Primary methods of the *AggregationQueryFunction* class

override void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder

override void AddUsingParameters(QueryParameterCollection resultParameters)

Adds the passed-in collection of parameters in the function arguments.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *AggregationQueryFunction* instance.

AggregationQueryFunction All()

Sets the [To All Values] scope for the current aggregate function.

AggregationQueryFunction Distinct()

Sets the [To Unique Values] scope for the current aggregate function.

The “Terrasoft.Core.DB.IsNotNullQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *.IsNotNullQueryFunction* class.

Constructors

.IsNotNullQueryFunction()

Initializes a new *.IsNotNullQueryFunction* instance.

.IsNotNullQueryFunction(QueryColumnExpression checkExpression, QueryColumnExpression replacementExpression)

.IsNotNullQueryFunction(IQueryColumnExpressionConvertible checkExpression, IQueryColumnExpressionConvertible replacementExpression)

Initializes a new *.IsNotNullQueryFunction* instance for the specified validated expression and substitute expression.

Parameters:

- *checkExpression* – the expression to check the *null* value;
- *replacementExpression* – the expression returned by the function if *checkExpression* is *null*.

.IsNotNullQueryFunction((IsNotNullQueryFunction source))

Initializes a new *.IsNotNullQueryFunction* instance that is a clone of the passed expression function.

Parameters:

- *source* – the *.IsNotNullQueryFunction* expression aggregate function whose clone is being created.

Properties

Table 4. Primary properties of the *.IsNotNullQueryFunction* class

CheckExpression

QueryColumnExpression

Expression of the function argument to check the *null* value.

ReplacementExpression

QueryColumnExpression

Expression of the function argument that is returned if the check expression is equal to *null*.

Methods

Table 5. Primary methods of the *.IsNotNullQueryFunction* class

override void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;

- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

Adds the passed-in collection of parameters in the function arguments.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *IsNullQueryFunction* instance.

The “Terrasoft.Core.DB.CreateGuidQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, parent classes, and implemented interfaces of the *CreateGuidQueryFunction* class.

Constructors

CreateGuidQueryFunction()

Initializes a new *CreateGuidQueryFunction* instance.

CreateGuidQueryFunction((CreateGuidQueryFunction source))

Initializes a new *CreateGuidQueryFunction* instance. The instance is a clone of the passed function.

Parameters:

- *source* – the *CreateGuidQueryFunction* whose clone is being created.

Methods

Table 6. Primary methods of the *CreateGuidQueryFunction* class

override void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override object Clone()

Clones the current *CreateGuidQueryFunction* instance.

The “Terrasoft.Core.DB.CurrentDateTimeQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, parent classes, and implemented interfaces of the *CurrentDateTimeQueryFunction* class.

Constructors

CurrentDateTimeQueryFunction()

Initializes a new *CurrentDateTimeQueryFunction* instance.

CurrentDateTimeQueryFunction((CurrentDateTimeQueryFunction source))

Initializes a new *CurrentDateTimeQueryFunction* instance. The instance is a clone of the passed function.

Parameters:

- *source* – the *CurrentDateTimeQueryFunction* whose clone is being created.

Methods

Table 7. Primary methods of the *CurrentDateTimeQueryFunction* class

override void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override object Clone()

Creates a clone of the current *CurrentDateTimeQueryFunction* instance.

The “Terrasoft.Core.DB.CoalesceQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *CoalesceQueryFunction* class.

Constructors

CoalesceQueryFunction()

Initializes a new *CoalesceQueryFunction* instance.

CoalesceQueryFunction((CoalesceQueryFunction source))

Initializes a new *CoalesceQueryFunction* instance. The instance is a clone of the passed function.

Parameters:

- *source* – the *CoalesceQueryFunction* whose clone is being created.

CoalesceQueryFunction((QueryColumnExpressionCollection expressions))

Initializes a new *CoalesceQueryFunction* instance for the passed collection of column expressions.

Parameters:

- *expressions* – a collection of query column expressions.

CoalesceQueryFunction((QueryColumnExpression[] expressions))

CoalesceQueryFunction((IQueryColumnExpressionConvertible[] expressions))

Initializes a new *CoalesceQueryFunction* instance for the passed array of column expressions.

Parameters:

- *expressions* – an array of expressions of query columns.

Properties

Table 8. Primary properties of the *CoalesceQueryFunction* class

Expressions

QueryColumnExpressionCollection

Collection of expressions of function arguments.

Methods

Table 9. Primary methods of the *CoalesceQueryFunction* class

override void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override object Clone()

Creates a clone of the current *CoalesceQueryFunction* instance.

override void AddUsingParameters(QueryParameterCollection resultParameters)

Adds specified parameters to a collection.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

The “Terrasoft.Core.DB.DatePartQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *DatePartQueryFunction* class.

Constructors

DatePartQueryFunction()

Initializes a new instance of the *DatePartQueryFunction* class.

DatePartQueryFunction(DatePartQueryFunctionInterval interval, QueryColumnExpression expression)

DatePartQueryFunction(DatePartQueryFunctionInterval interval, IQueryColumnExpressionConvertible expression)

Initializes a new *DatePartQueryFunction* instance with the specified expression of the column of the Date/Time type and the specified date part.

Parameters:

- *interval* – the date part;
- *expression* – the expression of the column of the Date/Time type.

DatePartQueryFunction((DatePartQueryFunction source))

Initializes a new *DatePartQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *DatePartQueryFunction* whose clone is being created.

Properties

Table 10. Primary properties of the *DatePartQueryFunction* class

Expression

QueryColumnExpression

The expression of the function argument.

Interval

DatePartQueryFunctionInterval

The date part returned by the function.

UseUtcOffset

bool

Using the coordinated universal time (UTC) offset from the specified local time.

UtcOffset

int?

Using the coordinated universal time (UTC).

Methods

Table 11. Primary methods of the *DatePartQueryFunction* class

override void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

Adds specified parameters to a collection.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *DatePartQueryFunction* instance.

The “Terrasoft.Core.DB.DateAddQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *DatePartQueryFunction* class.

Constructors

DateAddQueryFunction()

Initializes a new instance of the *DateAddQueryFunction* class.

DateAddQueryFunction(DatePartQueryFunctionInterval interval, int number, QueryColumnExpression expression)

DateAddQueryFunction(DatePartQueryFunctionInterval interval, IQueryColumnExpressionConvertible numberExpression, IQueryColumnExpressionConvertible expression)

DateAddQueryFunction(DatePartQueryFunctionInterval interval, int number, IQueryColumnExpressionConvertible expression)

Initializes the *DateAddQueryFunction* instance with the specified parameters.

Parameters:

- *interval* – the date part to which the time span is added;
- *number* – the value added to *interval*;
- *expression* – the expression of the column with the original date.

DateAddQueryFunction((DateAddQueryFunction source))

Initializes the *DateAddQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – an instance of the *CoalesceQueryFunction* whose clone is being created.

Properties

Table 12. Primary properties of the *DateAddQueryFunction* class

Expression

QueryColumnExpression

Expression of the column containing the original date.

Interval

DatePartQueryFunctionInterval

Datepart to which the time span is added.

Number
int

The time span to add.

NumberExpression
QueryColumnExpression

Expression containing the time period to be added.

Methods

Table 13. Primary methods of the *DateAddQueryFunction* class

override void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

Adds specified parameters to a collection.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *DateAddQueryFunction* instance.

The “Terrasoft.Core.DB.DateDiffQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *DatePartQueryFunction* class.

Constructors

DateDiffQueryFunction(DateDiffQueryFunctionInterval interval, QueryColumnExpression startDateExpression, QueryColumnExpression endDateExpression)

DateDiffQueryFunction(DateDiffQueryFunctionInterval interval, IQueryColumnExpressionConvertible startDateExpression, IQueryColumnExpressionConvertible endDateExpression)

Initializes the *DateDiffQueryFunction* instance with the specified parameters.

Parameters:

- *interval* – the unit for measuring the date difference;
- *startDateExpression* – an expression of the column with the start date;
- *endDateExpression* – an expression of the column with the end date.

DateDiffQueryFunction((DateDiffQueryFunction source))

Initializes the *DateDiffQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – an instance of the *CoalesceQueryFunction* whose clone is being created.

Properties

Table 14. Primary properties of the *DateDiffQueryFunction* class

StartDateExpression

QueryColumnExpression

An expression of the column containing the start date.

EndDateExpression

QueryColumnExpression

An expression of the column containing the end date.

Interval

DateDiffQueryFunctionInterval

Unit of measurement of the date difference returned by function.

Methods

Table 15. Primary methods of the *DateDiffQueryFunction* class

override void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

Adds specified parameters to a collection.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *DateDiffQueryFunction* instance.

The “Terrasoft.Core.DB.CastQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *CastQueryFunction* class.

Constructors

CastQueryFunction(QueryColumnExpression expression, DBDataType castType)

CastQueryFunction(IQueryColumnExpressionConvertible expression, DBDataType castType)

Initializes a new *CastQueryFunction* instance with the specified column expression and target data type.

Parameters:

- *expression* – a query column expression;
- *castType* – the target data type.

CastQueryFunction((CastQueryFunction source))

Initializes a new *CastQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *CastQueryFunction* whose clone is being created.

Properties

Table 16. Primary properties of the *CastQueryFunction* class

Expression

QueryColumnExpression

The expression of the function argument.

CastType

DBDataValueType

The target data type.

Methods

Table 17. Primary methods of the *CastQueryFunction* class

override void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

Adds specified parameters to a collection.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *CastQueryFunction* instance.

The “Terrasoft.Core.DB.UpperQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *UpperQueryFunction* class.

Constructors

UpperQueryFunction()

Initializes a new instance of the *UpperQueryFunction* class.

UpperQueryFunction((QueryColumnExpression expression))

UpperQueryFunction((IQueryColumnExpressionConvertible expression))

Initializes a new *UpperQueryFunction* instance for the specified column expression.

Parameters:

- *expression* – a query column expression.

UpperQueryFunction((UpperQueryFunction source))

Initializes a new *UpperQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *UpperQueryFunction* whose clone is being created.

Properties

Table 18. Primary properties of the *UpperQueryFunction* class

Expression

QueryColumnExpression

The expression of the function argument.

Methods

Table 19. Primary methods of the *UpperQueryFunction* class

override void BuildSqlText(StringBuilder sb, DBEngine dbEngine)

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

Adds specified parameters to a collection.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *UpperQueryFunction* instance.

The “Terrasoft.Core.DB.CustomQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *CastQueryFunction* class.

Constructors

CustomQueryFunction()

Initializes a new instance of the *CustomQueryFunction* class.

CustomQueryFunction((string functionName, QueryColumnExpressionCollection expressions))

Initializes a new *CustomQueryFunction* instance for the specified function and passed collection of column expressions.

Parameters:

- *functionName* – the function name;
- *expressions* – a collection of query column expressions.

CustomQueryFunction((string functionName, QueryColumnExpression[] expressions))

CustomQueryFunction((string functionName, IQueryColumnExpressionConvertible[] expressions))

Initializes a new *CustomQueryFunction* instance for the specified function and passed array of column expressions.

Parameters:

- *functionName* – the function name;
- *expressions* – an array of expressions of query columns.

CustomQueryFunction((CustomQueryFunction source))

Initializes a new *CustomQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *CustomQueryFunction* whose clone is being created.

Properties

Table 20. Primary properties of the *CustomQueryFunction* class

Expressions

QueryColumnExpressionCollection

Collection of expressions of function arguments.

FunctionName

string

The name of the function.

Methods

Table 21. Primary methods of the *CustomQueryFunction* class

override void BuildSqlText((StringBuilder sb, DBEngine dbEngine))

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
 - *dbEngine* – the instance of the database query builder.
-

override void AddUsingParameters(QueryParameterCollection resultParameters)

Adds specified parameters to a collection.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.
-

override object Clone()

Creates a clone of the current *CustomQueryFunction* instance.

The “Terrasoft.Core.DB.DataLengthQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *DataLengthQueryFunction* class.

Constructors

DataLengthQueryFunction()

Initializes a new instance of the *DataLengthQueryFunction* class.

DataLengthQueryFunction((QueryColumnExpression expression))

Initializes a new *DataLengthQueryFunction* instance for the specified column expression.

Parameters:

- *expression* – a query column expression.
-

DataLengthQueryFunction((IQueryColumnExpressionConvertible columnNameExpression))

Initializes a new *DataLengthQueryFunction* instance for the specified column expression.

Parameters:

- *columnNameExpression* – a query column expression.
-

DataLengthQueryFunction((DataLengthQueryFunction source))

Initializes a new *DataLengthQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *DataLengthQueryFunction* whose clone is being created.
-

Properties

Table 22. Primary properties of the *DataLengthQueryFunction* class

Expression

QueryColumnExpression

The expression of the function argument.

Methods

Table 23. Primary methods of the *DataLengthQueryFunction* class

override void BuildSqlText((StringBuilder sb, DBEngine dbEngine))

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

In the function arguments adds the passed-in collection of parameters.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *DataLengthQueryFunction* instance.

The “Terrasoft.Core.DB.TrimQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *CastQueryFunction* class.

Constructors

TrimQueryFunction((QueryColumnExpression expression))

TrimQueryFunction((IQueryColumnExpressionConvertible expression))

Initializes a new *TrimQueryFunction* instance for the specified column expression.

Parameters:

- *expression* – a query column expression.

TrimQueryFunction((TrimQueryFunction source))

Initializes a new *TrimQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *TrimQueryFunction* whose clone is being created.

Properties

Table 24. Primary properties of the *TrimQueryFunction* class

Expression

QueryColumnExpression

The expression of the function argument.

Methods

Table 25. Primary methods of the *TrimQueryFunction* class

override void BuildSqlText((StringBuilder sb, DBEngine dbEngine))

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

In the function arguments adds the passed-in collection of parameters.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *TrimQueryFunction* instance.

The “Terrasoft.Core.DB.LengthQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *DataLengthQueryFunction* class.

Constructors

LengthQueryFunction()

Initializes a new instance of the *LengthQueryFunction* class.

LengthQueryFunction((QueryColumnExpression expression))

LengthQueryFunction((IQueryColumnExpressionConvertible expression))

Initializes a new *LengthQueryFunction* instance for the specified column expression.

Parameters:

- *expression* – a query column expression.

LengthQueryFunction((LengthQueryFunction source))

Initializes a new *LengthQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *LengthQueryFunction* whose clone is being created.

Properties

Table 26. Primary properties of the *LengthQueryFunction* class

Expression

QueryColumnExpression

The expression of the function argument.

Methods

Table 27. Primary methods of the *LengthQueryFunction* class

override void BuildSqlText((StringBuilder sb, DBEngine dbEngine))

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

In the function arguments adds the passed-in collection of parameters.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *LengthQueryFunction* instance.

The “Terrasoft.Core.DB.SubstringQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *SubstringQueryFunction* class.

Constructors

SubstringQueryFunction((QueryColumnExpression expression, int start, int length))

SubstringQueryFunction((IQueryColumnExpressionConvertible expression, int start, int length))

Initializes a new *SubstringQueryFunction* instance for the specified column expression, starting position and the length of the substring.

Parameters:

- *expression* – a query column expression;
- *start* – the start index of the substring;
- *length* – the length of the substring.

SubstringQueryFunction((SubstringQueryFunction source))

Initializes a new *SubstringQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *SubstringQueryFunction* whose clone is being created.

Properties

Table 28. Primary properties of the *SubstringQueryFunction* class

Expression

QueryColumnExpression

The expression of the function argument.

StartExpression

QueryColumnExpression

The start index of the substring.

LengthExpression

QueryColumnExpression

The length of the substring.

Methods

Table 29. Primary methods of the *SubstringQueryFunction* class

override void BuildSqlText((StringBuilder sb, DBEngine dbEngine))

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

In the function arguments adds the passed-in collection of parameters.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *SubstringQueryFunction* instance.

The “Terrasoft.Core.DB.ConcatQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *ConcatQueryFunction* class.

Constructors

ConcatQueryFunction((QueryColumnExpressionCollection expressions))

Initializes a new *ConcatQueryFunction* instance for the passed expression collection.

Parameters:

- *expressions* – a collection of query column expressions.

ConcatQueryFunction((ConcatQueryFunction source))

Initializes a new *ConcatQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *ConcatQueryFunction* whose clone is being created.

Properties

Table 30. Primary properties of the *ConcatQueryFunction* class

Expressions

QueryColumnExpressionCollection

Collection of expressions of function arguments.

Methods

Table 31. Primary methods of the *ConcatQueryFunction* class

override void BuildSqlText((StringBuilder sb, DBEngine dbEngine))

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

override void AddUsingParameters(QueryParameterCollection resultParameters)

In the function arguments adds the passed-in collection of parameters.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

override object Clone()

Creates a clone of the current *ConcatQueryFunction* instance.

The “Terrasoft.Core.DB.WindowQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *WindowQueryFunction* class.

Constructors

WindowQueryFunction((QueryFunction innerFunction))

Implements an SQL window function.

Parameters:

- *innerFunction* – nested function.

```
WindowQueryFunction((QueryFunction innerFunction, QueryColumnExpression partitionByExpression = null,  
QueryColumnExpression orderByExpression = null) : this(innerFunction))
```

Implements an SQL window function.

Parameters:

- *innerFunction* – nested function;
- *partitionByExpression* – expression for separating the query;
- *orderByExpression* – expression for sorting the query.

```
WindowQueryFunction((WindowQueryFunction source) : this( source.InnerFunction,  
source.PartitionByExpression, source.OrderByExpression))
```

Initializes a new *WindowQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *WindowQueryFunction* whose clone is being created.

Properties

Table 32. Primary properties of the *WindowQueryFunction* class

InnerFunction

QueryFunction

The function to apply.

PartitionByExpression

QueryColumnExpression

Split by expression.

OrderByExpression

QueryColumnExpression

Sort by expression.

Methods

Table 33. Primary methods of the *WindowQueryFunction* class

```
override void BuildSqlText((StringBuilder sb, DBEngine dbEngine))
```

Generates the query text, using the specified *StringBuilder* instance and the *DBEngine* query builder.

Parameters:

- *sb* – *StringBuilder* instance used to create query text;
- *dbEngine* – the instance of the database query builder.

```
override void AddUsingParameters(QueryParameterCollection resultParameters)
```

In the function arguments adds the passed-in collection of parameters.

Parameters:

- *resultParameters* – the collection of query parameters that are added in the function arguments.

```
override object Clone()
```

Creates a clone of the current *WindowQueryFunction* instance.

The EntitySchemaQueryFunction class

Beginner**Easy****Medium****Advanced**

Introduction

The `Terrasoft.Core.Entities.EntitySchemaQueryFunction` class implements the expression function.

The expression function is implemented in the following classes:

- `EntitySchemaQueryFunction` – the base class of expression of the entity schema query.
- `EntitySchemaAggregationQueryFunction` – implements the aggregate function of the expression.
- `EntitySchemaIsNullQueryFunction` – replaces `null` values with the replacement expression.
- `EntitySchemaCoalesceQueryFunction` – returns the first not `null` expression from the list of arguments.
- `EntitySchemaCaseNotNullQueryFunctionWhenItem` – class that describes the condition expression of the CASE SQL operator.
- `EntitySchemaCaseNotNullQueryFunctionWhenItems` – collection of condition expressions of the CASE SQL operator.
- `EntitySchemaCaseNotNullQueryFunction` – returns one value from the set of possible values depending on the specified conditions.
- `EntitySchemaSystemValueQueryFunction` – returns the expression of the system value.
- `EntitySchemaCurrentDateTimeQueryFunction` – implements the function for current date and time expression.
- `EntitySchemaBaseCurrentDateQueryFunction` – base class of the expression function for the base date.
- `EntitySchemaCurrentDateQueryFunction` – implements the function for current date and time expression.
- `EntitySchemaDateToCurrentYearQueryFunction` – implements the function that converts the date expression to the same date of the current year.
- `EntitySchemaBaseCurrentDateTimeQueryFunction` – implements the function for current date and time expression.
- `EntitySchemaDatePartQueryFunction` – implements the function for the datepart of the Date/Time type.
- `EntitySchemaCastQueryFunction` – casts the argument expression to the specified data type.
- `EntitySchemaUpperQueryFunction` – converts the argument expression characters to uppercase.
- `EntitySchemaTrimQueryFunction` – removes whitespaces from both ends of the expression.
- `EntitySchemaLengthQueryFunction` – returns the length of the expression.
- `EntitySchemaConcatQueryFunction` – returns a string resulting from merging the string arguments of the function.
- `EntitySchemaWindowQueryFunction` – implements an SQL window function.

The “`Terrasoft.Core.Entities.EntitySchemaQueryFunction`” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, parent classes, and implemented interfaces of the `EntitySchemaQueryFunction` class.

Methods

Table 1. Primary methods of the `EntitySchemaQueryFunction` class

`public abstract QueryColumnExpression CreateQueryColumnExpression((DBSecurityEngine dbSecurityEngine))`
For the current function, gets the query column expression that is generated taking into account the specified access rights.

Parameters:

- `dbSecurityEngine` – a `Terrasoft.Core.DB.DBSecurityEngine` object that defines access rights.

`public abstract DataValueType GetResultDataValueType((DataValueTypeManager dataValueTypeManager))`
Gets the data type of the output returned by the function, using the passed-in data type manager.

Parameters:

- `dataValueTypeManager` – data type manager.

`public abstract bool GetIsSupportedDataType((DataValueType dataType))`
Indicates whether the output of the function has the specified data type.

Parameters:

- *dataType* – the type of data.

`public abstract string GetCaption()`

Gets the caption of the expression function.

`public virtual EntitySchemaQueryExpressionCollection GetArguments()`

Gets the collection of expressions of function arguments.

`public void CheckIsSupportedDataType((DataValueType dataType))`

Verifies that the output of the function has the specified data type. Otherwise, an exception is thrown.

Parameters:

- *dataType* – the type of data.

The “*Terrasoft.Core.Entities.EntitySchemaAggregationQueryFunction*” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaAggregationQueryFunction* class.

Constructors

`EntitySchemaAggregationQueryFunction((EntitySchemaQuery parentQuery))`

Initializes the *EntitySchemaAggregationQueryFunction* instance of the specified aggregate function type for the specified query to the object schema.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

`EntitySchemaAggregationQueryFunction((AggregationTypeStrict aggregationType, EntitySchemaQuery parentQuery))`

Initializes the *EntitySchemaAggregationQueryFunction* instance of the specified aggregate function type for the specified query to the object schema.

Parameters:

- *aggregationType* – the type of aggregating function;
- *parentQuery* – query against the schema of the entity that contains the function.

`EntitySchemaAggregationQueryFunction((AggregationTypeStrict aggregationType, EntitySchemaQueryExpression expression, EntitySchemaQuery parentQuery))`

Initializes a new *EntitySchemaAggregationQueryFunction* instance for the specified type of aggregate /// function, expression, and query to the object schema.

Parameters:

- *aggregationType* – the type of aggregating function;
- *expression* – a query expression;
- *parentQuery* – query against the schema of the entity that contains the function.

`EntitySchemaAggregationQueryFunction((EntitySchemaAggregationQueryFunction source))`

Initializes a new *EntitySchemaAggregationQueryFunction* instance that is a clone of the passed aggregate expression function instance.

Parameters:

- *source* – instance of the expression aggregate function whose clone is being created.

Properties

Table 2. Primary properties of the *EntitySchemaAggregationQueryFunction* class

QueryAlias
string

The alias of the function in the SQL query.

AggregationType
AggregationTypeStrict

The type of aggregating function.

AggregationEvalType
AggregationEvalType

The scope of the aggregate function.

Expression
EntitySchemaQueryExpression

The expression of the aggregate function argument.

Methods

Table 3. Primary methods of the *EntitySchemaAggregationQueryFunction* class

override void WriteMetaData((DataWriter writer))

Serializes the aggregate function, using the specified *Terrasoft.Common.DataWriter* instance.

Parameters:

- *writer* – the *Terrasoft.Common.DataWriter* instance used for serialization.

override QueryColumnExpression CreateQueryColumnExpression((DBSecurityEngine dbSecurityEngine))

For the aggregate function, gets the query column expression that is generated taking into account the specified access rights.

Parameters:

- *dbSecurityEngine* – a *Terrasoft.Core.DB.DBSecurityEngine* object that defines access rights.

override EntitySchemaQueryExpressionCollection GetArguments()

Gets the collection of expressions of the aggregate function arguments.

override DataValueType GetResultDataValueType((DataValueTypeManager dataValueTypeManager))

Gets the data type of the output returned by the aggregate function, using the specified data type manager.

Parameters:

- *dataValueTypeManager* – data type manager.

override bool GetIsSupportedDataValueType((DataValueType dataType))

Indicates whether the output of the aggregate function has the specified data type.

Parameters:

- *dataType* – the type of data.

override string GetCaption()

Gets the caption of the expression function.

override object Clone()

Creates a clone of the current *EntitySchemaAggregationQueryFunction* instance.

EntitySchemaAggregationQueryFunction All()

Sets the [To All Values] scope for the current aggregate function.

`EntitySchemaAggregationQueryFunction Distinct()`

Sets the [To Unique Values] scope for the current aggregate function.

The “Terrasoft.Core.Entities.EntitySchemaIsNullOrEmptyQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the `EntitySchemaIsNullOrEmptyQueryFunction` class.

Constructors

`EntitySchemaIsNullOrEmptyQueryFunction((EntitySchemaQuery parentQuery))`

Initializes the `EntitySchemaIsNullOrEmptyQueryFunction` instance for the specified entity schema query.

Parameters:

- `parentQuery` – query against the schema of the entity that contains the function.

`EntitySchemaIsNullOrEmptyQueryFunction((EntitySchemaQuery parentQuery, EntitySchemaQueryExpression checkExpression, EntitySchemaQueryExpression replacementExpression))`

Initializes a new `EntitySchemaIsNullOrEmptyQueryFunction` instance for the specified query to the object schema, validated expression and substitute expression.

Parameters:

- `parentQuery` – query against the schema of the entity that contains the function;
- `checkExpression` – the expression to check for being `null`;
- `replacementExpression` – the expression returned by the function if `checkExpression` is `null`.

`EntitySchemaIsNullOrEmptyQueryFunction((EntitySchemaIsNullOrEmptyQueryFunction source))`

Initializes a new `EntitySchemaIsNullOrEmptyQueryFunction` instance that is a clone of the passed expression function.

Parameters:

- `source` – an instance of the `EntitySchemaIsNullOrEmptyQueryFunction` whose clone is being created.

Properties

Table 4 Primary properties of the `EntitySchemaIsNullOrEmptyQueryFunction` class

`QueryAlias`

`string`

The alias of the function in the SQL query.

`CheckExpression`

`EntitySchemaQueryExpression`

Expression of the function argument to check for being equal to the `null` value.

`ReplacementExpression`

`EntitySchemaQueryExpression`

Expression of the function argument that is returned if the check expression is equal to `null`.

Methods

Table 5. Primary methods of the `EntitySchemaIsNullOrEmptyQueryFunction` class

`override void WriteMetaData((DataWriter writer))`

Serializes the expression function, using the passed `DataWriter` instance.

Parameters:

- *writer* – the *DataWriter* instance used for serializing the expression function.

override QueryColumnExpression CreateQueryColumnExpression((DBSecurityEngine dbSecurityEngine))

For the current function, gets the query column expression that is generated taking into account the specified access rights.

Parameters:

- *dbSecurityEngine* – a *Terrasoft.Core.DB.DBSecurityEngine* object that defines the access rights.

override EntitySchemaQueryExpressionCollection GetArguments()

Gets the collection of expressions of function arguments.

override DataValueType GetResultDataValueType((DataValueTypeManager dataValueTypeManager))

Gets the data type of the output returned by the function, using the passed-in data type manager.

Parameters:

- *dataValueTypeManager* – data type manager.

The “**Terrasoft.Core.Entities.EntitySchemaCoalesceQueryFunction**” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaCoalesceQueryFunction* class.

Constructors

EntitySchemaCoalesceQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaCoalesceQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaCoalesceQueryFunction((EntitySchemaCoalesceQueryFunction source))

Initializes a new *EntitySchemaCoalesceQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaCoalesceQueryFunction* whose clone is being created.

Properties

Table 6. Primary properties of the *EntitySchemaCoalesceQueryFunction* class

QueryAlias

string

The alias of the function in the SQL query.

Expressions

EntitySchemaQueryExpressionCollection

Collection of expressions of function arguments.

HasExpressions

bool

Indicates whether at least one item exists in the collection of expressions of the function arguments.

Methods

Table 7. Primary methods of the *EntitySchemaCoalesceQueryFunction* class

override bool GetIsSupportedDataValueType((DataValueType dataType))

Indicates whether the output of the function has the specified data type.

Parameters:

- *dataValueType* – the type of data.

The “Terrasoft.Core.Entities.EntitySchemaCaseNotNullQueryFunctionWhenItem” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, parent classes, and implemented interfaces of the *EntitySchemaCaseNotNullQueryFunctionWhenItem* class.

Constructors

EntitySchemaCaseNotNullQueryFunctionWhenItem()

Initializes a new instance of the *EntitySchemaCaseNotNullQueryFunctionWhenItem* class.

EntitySchemaCaseNotNullQueryFunctionWhenItem((EntitySchemaQueryExpression whenExpression, EntitySchemaQueryExpression thenExpression))

Initializes the *EntitySchemaCaseNotNullQueryFunctionWhenItem* instance for the specified expressions of the WHEN and THEN clauses.

Parameters:

- *whenExpression* – expression of the WHEN condition clause;
- *thenExpression* – expression of the THEN condition clause.

EntitySchemaCaseNotNullQueryFunctionWhenItem((EntitySchemaCaseNotNullQueryFunctionWhenItem source))

Initializes the *EntitySchemaCaseNotNullQueryFunctionWhenItem* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaCaseNotNullQueryFunctionWhenItem* whose clone is being created.

Properties

Table 8. Primary properties of the *EntitySchemaCaseNotNullQueryFunctionWhenItem* class

WhenExpression

EntitySchemaQueryExpression

Expression of the WHEN clause.

ThenExpression

EntitySchemaQueryExpression

Expression of the THEN clause.

The “Terrasoft.Core.Entities.EntitySchemaCaseNotNullQueryFunctionWhenItems” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, parent classes, and implemented interfaces of the *EntitySchemaCaseNotNullQueryFunctionWhenItems* class.

Constructors

EntitySchemaCaseNotNullQueryFunctionWhenItems()

Initializes an *EntitySchemaCaseNotNullQueryFunctionWhenItems* instance.

EntitySchemaCaseNotNullQueryFunctionWhenItems((EntitySchemaCaseNotNullQueryFunctionWhenItems source))

Initializes a new *EntitySchemaCaseNotNullQueryFunctionWhenItems* instance that is a clone of the passed

collection of conditions.

Parameters:

- *source* – the collection of conditions whose clone is being created.

The “Terrasoft.Core.Entities.EntitySchemaCaseNotNullQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaCaseNotNullQueryFunction* class.

Constructors

EntitySchemaCaseNotNullQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaCaseNotNullQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaCaseNotNullQueryFunction((EntitySchemaCaseNotNullQueryFunction source))

Initializes a new *EntitySchemaCaseNotNullQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaCaseNotNullQueryFunction* whose clone is being created.

Properties

Table 9. Primary properties of the *EntitySchemaCaseNotNullQueryFunction* class

QueryAlias

string

The alias of the function in the SQL query.

WhenItems

EntitySchemaCaseNotNullQueryFunctionWhenItems

Collection of conditions of the expression function.

HasWhenItems

bool

Indicates whether the function has at least one condition.

ElseExpression

EntitySchemaQueryExpression

Expression of the ELSE clause.

Methods

Table 10. Primary methods of the *EntitySchemaCaseNotNullQueryFunction* class

void SpecifyQueryAlias((string queryAlias))

For the current expression function, defines the specified alias in the resulting SQL query.

Parameters:

- *queryAlias* – alias to define for the current function.

The “Terrasoft.Core.Entities.EntitySchemaSystemValueQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties,

parent classes, and implemented interfaces of the *EntitySchemaSystemValueQueryFunction* class.

Properties

Table 11. Primary properties of the *EntitySchemaSystemValueQueryFunction* class

QueryAlias

string

The alias of the function in the SQL query.

SystemValueName

string

Name of the system value.

The “Terrasoft.Core.Entities.EntitySchemaBaseCurrentDateQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaBaseCurrentDateQueryFunction* class.

Properties

Table 12. Primary properties of the *EntitySchemaBaseCurrentDateQueryFunction* class

SystemValueName

string

Name of the system value.

Offset

int

The offset.

The “Terrasoft.Core.Entities.EntitySchemaDateToCurrentYearQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaDateToCurrentYearQueryFunction* class.

Constructors

EntitySchemaDateToCurrentYearQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaDateToCurrentYearQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaDateToCurrentYearQueryFunction((EntitySchemaQuery parentQuery, EntitySchemaQueryExpression expression))

Initializes the new *EntitySchemaDateToCurrentYearQueryFunction* instance for the specified entity schema query and passed date expression.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function;
- *expression* – a query expression.

EntitySchemaDateToCurrentYearQueryFunction((EntitySchemaDateToCurrentYearQueryFunction source))

Initializes a new *EntitySchemaDateToCurrentYearQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaDateToCurrentYearQueryFunction* whose clone is being created.

Properties

Table 13. Primary properties of the *EntitySchemaDateToCurrentYearQueryFunction* class

QueryAlias
string

The alias of the function in the SQL query.

Expression
EntitySchemaQueryExpression

The expression of the function arguments.

The “Terrasoft.Core.Entities.EntitySchemaCurrentTimeQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaCurrentTimeQueryFunction* class.

Constructors

EntitySchemaCurrentTimeQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaCurrentTimeQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaCurrentTimeQueryFunction((EntitySchemaCurrentTimeQueryFunction source))

Initializes a new *EntitySchemaCurrentTimeQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaCurrentTimeQueryFunction* whose clone is being created.

Properties

Table 14. Primary properties of the *EntitySchemaCurrentTimeQueryFunction* class

SystemValueName
string

Name of the system value.

The “Terrasoft.Core.Entities.EntitySchemaCurrentUserQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaCurrentUserQueryFunction* class.

Constructors

EntitySchemaCurrentUserQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaCurrentUserQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaCurrentUserQueryFunction((EntitySchemaCurrentUserQueryFunction source))

Initializes a new *EntitySchemaCurrentUserQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaCurrentUserQueryFunction* whose clone is being created.

Properties

Table 15. Primary properties of the *EntitySchemaCurrentUserQueryFunction* class

SystemValueName
string

Name of the system value.

The “Terrasoft.Core.Entities.EntitySchemaCurrentUserAccountQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaCurrentUserAccountQueryFunction* class.

Constructors

EntitySchemaCurrentUserAccountQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaCurrentUserAccountQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaCurrentUserAccountQueryFunction((EntitySchemaCurrentUserAccountQueryFunction source))

Initializes a new *EntitySchemaCurrentUserAccountQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaCurrentUserAccountQueryFunction* whose clone is being created.

Properties

Table 16. Primary properties of the *EntitySchemaCurrentUserAccountQueryFunction* class

SystemValueName
string

Name of the system value.

The “Terrasoft.Core.Entities.EntitySchemaUpperQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaUpperQueryFunction* class.

Constructors

EntitySchemaUpperQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaUpperQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaUpperQueryFunction((EntitySchemaQuery parentQuery, EntitySchemaQueryExpression expression))

Initializes the new *EntitySchemaUpperQueryFunction* instance for the specified entity schema query and passed date expression.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function;
- *expression* – a query expression.

EntitySchemaUpperQueryFunction((EntitySchemaUpperQueryFunction source))

Initializes a new *EntitySchemaUpperQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaUpperQueryFunction* whose clone is being created.

Properties

Table 17. Primary properties of the *EntitySchemaUpperQueryFunction* class

QueryAlias

string

The alias of the function in the SQL query.

Expression

EntitySchemaQueryExpression

The expression of the function arguments.

The “Terrasoft.Core.Entities.EntitySchemaTrimQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaTrimQueryFunction* class.

Constructors

EntitySchemaTrimQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaTrimQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaTrimQueryFunction((EntitySchemaQuery parentQuery, EntitySchemaQueryExpression expression))

Initializes the new *EntitySchemaTrimQueryFunction* instance for the specified entity schema query and passed date expression.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function;
- *expression* – a query expression.

EntitySchemaTrimQueryFunction((EntitySchemaTrimQueryFunction source))

Initializes a new *EntitySchemaTrimQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaTrimQueryFunction* whose clone is being created.

Properties

Table 18. Primary properties of the *EntitySchemaTrimQueryFunction* class

QueryAlias

string

The alias of the function in the SQL query.

Expression

EntitySchemaQueryExpression

The expression of the function arguments.

The “Terrasoft.Core.Entities.EntitySchemaLengthQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaLengthQueryFunction* class.

Constructors

EntitySchemaLengthQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaLengthQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaLengthQueryFunction((EntitySchemaQuery parentQuery, EntitySchemaQueryExpression expression))

Initializes the new *EntitySchemaLengthQueryFunction* instance for the specified entity schema query and passed date expression.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function;
- *expression* – a query expression.

EntitySchemaLengthQueryFunction((EntitySchemaLengthQueryFunction source))

Initializes a new *EntitySchemaLengthQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaLengthQueryFunction* whose clone is being created.

Properties

Table 19. Primary properties of the *EntitySchemaLengthQueryFunction* class

QueryAlias
string

The alias of the function in the SQL query.

Expression
EntitySchemaQueryExpression

The expression of the function arguments.

The “Terrasoft.Core.Entities.EntitySchemaCastQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaCastQueryFunction* class.

Constructors

EntitySchemaCastQueryFunction((EntitySchemaQuery parentQuery, DBDataValueType castType))

Initializes a new *EntitySchemaCastQueryFunction* instance for the specified query to the schema of the object with the specified target data type.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function;
- *castType* – the target data type.

EntitySchemaCastQueryFunction((EntitySchemaQuery parentQuery, EntitySchemaQueryExpression expression, DBDataValueType castType))

Initializes a new *EntitySchemaCastQueryFunction* instance with the specified expression and target data type.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function;
- *expression* – a query expression;
- *castType* – the target data type.

EntitySchemaCastQueryFunction((EntitySchemaCastQueryFunction source))

Initializes a new *EntitySchemaCastQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaCastQueryFunction* whose clone is being created.

Properties

Table 20. Primary properties of the *EntitySchemaCoalesceQueryFunction* class

QueryAlias

string

The alias of the function in the SQL query.

Expression

EntitySchemaQueryExpression

The expression of the function argument.

CastType

DBDataValueType

The target data type.

The “Terrasoft.Core.Entities.EntitySchemaConcatQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaConcatQueryFunction* class.

Constructors

EntitySchemaConcatQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaConcatQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaConcatQueryFunction((EntitySchemaQuery parentQuery, EntitySchemaQueryExpression[] expressions))

Initializes a new *EntitySchemaConcatQueryFunction* instance for the specified array of expressions and entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function;
- *expressions* – array of expressions.

EntitySchemaConcatQueryFunction((EntitySchemaConcatQueryFunction source))

Initializes a new *EntitySchemaConcatQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaConcatQueryFunction* whose clone is being created.

Properties

Table 21. Primary properties of the *EntitySchemaConcatQueryFunction* class

QueryAlias
string

The alias of the function in the SQL query.

Expressions
EntitySchemaQueryExpressionCollection

Collection of expressions of function arguments.

HasExpressions
bool

Indicates whether at least one item exists in the collection of expressions of the function arguments.

The “Terrasoft.Core.Entities.EntitySchemaWindowQueryFunction” class

Use the “[.NET class libraries of platform core](#)” documentation to access the full list of the methods, properties, parent classes, and implemented interfaces of the *EntitySchemaWindowQueryFunction* class.

Constructors

EntitySchemaWindowQueryFunction((EntitySchemaQuery parentQuery))

Initializes the new *EntitySchemaWindowQueryFunction* instance for the specified entity schema query.

Parameters:

- *parentQuery* – query against the schema of the entity that contains the function.

EntitySchemaWindowQueryFunction((EntitySchemaQueryExpression function, EntitySchemaQuery esq))

Initializes the new *EntitySchemaWindowQueryFunction* instance for the specified entity schema query.

Parameters:

- *function* – nested query function;
- *esq* – the expression of the entity schema query.

EntitySchemaWindowQueryFunction((EntitySchemaQueryExpression function, EntitySchemaQuery esq, EntitySchemaQueryExpression partitionBy = null, EntitySchemaQueryExpression orderBy = null))

Initializes the new *EntitySchemaWindowQueryFunction* instance for the specified entity schema query.

Parameters:

- *function* – nested query function;
- *parentQuery* – query against the schema of the entity that contains the function;
- *partitionBy* – the expression for splitting the query;
- *orderBy* – the expression for sorting the query.

EntitySchemaWindowQueryFunction((EntitySchemaQueryFunction source))

Initializes a new *EntitySchemaWindowQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaQueryFunction* whose clone is being created.

EntitySchemaWindowQueryFunction((EntitySchemaWindowQueryFunction source))

Initializes a new *EntitySchemaWindowQueryFunction* instance that is a clone of the passed function.

Parameters:

- *source* – the *EntitySchemaWindowQueryFunction* whose clone is being created.

Properties

Table 22. Primary properties of the *EntitySchemaConcatQueryFunction* class

<i>QueryAlias</i>
<i>string</i>
The alias of the function in the SQL query.
<i>InnerFunction</i>
<i>EntitySchemaQueryExpression</i>
The function to apply.
<i>PartitionByExpression</i>
<i>EntitySchemaQueryExpression</i>
Split by expression.
<i>OrderByExpression</i>
<i>EntitySchemaQueryExpression</i>
Sort by expression.

Examples

Contents

- **Creating replacement classes in packages**
- **Configuration web-services**
- **CRUD operations**
- **Work with localizable resources**

Creating replacement classes in packages

Beginner Easy Medium **Advanced**

Specifics of creating replacement classes

Working with replacement classes has the following specifics:

1. You must set a dependency on the replaceable class for a replacement class.
2. Developing replaceable and replacement classes must follow C# inheritance principles. In the hierarchy, the replaceable class is a parent, and the replacement class a descendant.
3. All properties and methods of a class to be replaced in other schemas must be virtual. In the replacement classes, these properties and methods are overloaded with the “override” keyword.
4. When instantiating replacement classes, comply with the general rules for creating and applying constructors.
5. Before the compilation, all properties and methods declared in the replacement class and not explicitly declared as virtual in the replaceable class will not be available. Binding and injection of type dependencies are implemented by the [Ninject](#) open-source framework for dependency injection only during execution.
6. When creating replaceable classes with parameterized constructors, make sure to comply with the rules for naming data types of argument instances that are passed as parameters to the *Get<T>* method.

Case description

Create a replaceable class and configuration service in a custom package. Create a replacement class in another custom package. Demonstrate the operation of the configuration service with and without class replacement.

Learn more about developing replacement classes in the “**Replacement class object factory**” article.

Source code

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

Case implementation algorithm

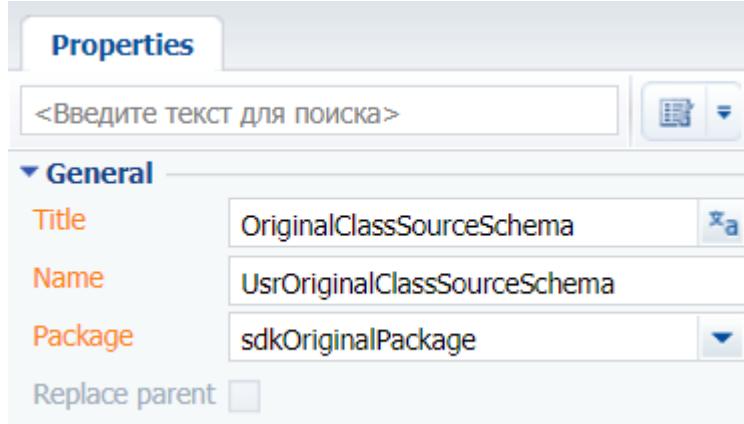
1. Implement a replaceable class

Go to the [Advanced settings] section -> [Configuration] -> Custom package -> the [Schemas] tab. Click [Add] -> [Source code]. Learn more about creating a schema of the [Source Code] type in the “**Creating the [Source code] schema**” article.

Specify the following parameters for the created object schema (Fig. 1):

- [Title] – “OriginalClassSourceSchema”.
- [Name] – "UsrOriginalClassSourceSchema".

Fig. 1. – Setting up the [Source Code] type object schema



Implement a replaceable class *OriginalClass* that contains the *GetAmount(int, int)* method for adding two values passed as parameters. The complete source code of the module is available below:

```
namespace Terrasoft.Configuration
{
    public class OriginalClass
    {
        // The GetAmount() method is virtual. It can be reloaded by descendants and
        // implemented.
        public virtual int GetAmount(int originalValue1, int originalValue2)
        {
            return originalValue1 + originalValue2;
        }
    }
}
```

After making changes, save and publish the schema.

2. Create a configuration service

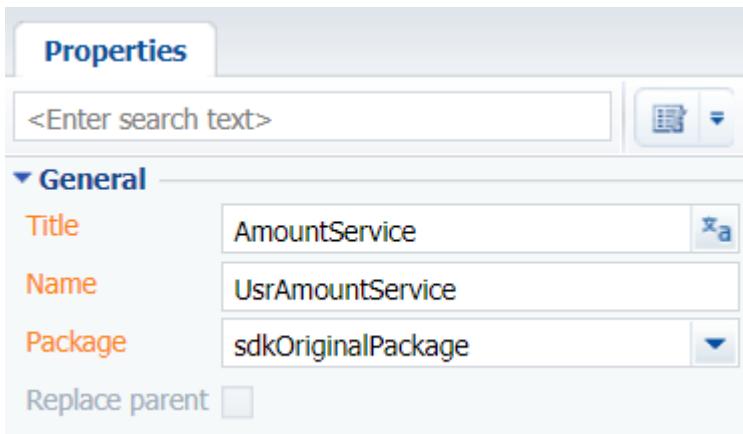
To demonstrate the operation of the *GetAmount(int, int)* method of the *OriginalClass* replaceable class, create a configuration service. Learn more about creating a custom configuration service in the “**Configuration service development (on-line documentation)**” article.

Go to the [Advanced settings] section -> [Configuration] -> Custom package -> the [Schemas] tab. Click [Add] -> [Source code]. Learn more about creating a schema of the [Source Code] type in the “**Creating the [Source code] schema**” article.

Specify the following parameters for the created object schema (Fig. 2):

- [Title] – “AmountService”.
- [Name] – "UsrAmountService".

Fig. 2. – Setting up the [Source Code] type object schema



Implement a replacement class *AmountService* that contains the *GetAmount(int, int)*, method for adding two values passed as parameters. The full source code with the implementation of the service is available below.

```
namespace Terrasoft.Configuration
{
    using System.ServiceModel;
    using System.ServiceModel.Activation;
    using System.ServiceModel.Web;
    using Terrasoft.Core;
    using Terrasoft.Web.Common;

    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Required)]
    public class AmountService : BaseService
    {

        [OperationContract]
        [WebGet(RequestFormat = WebMessageFormat.Json, BodyStyle =
        WebMessageBodyStyle.Wrapped, ResponseFormat = WebMessageFormat.Json)]
        public string GetAmount(int value1, int value2)
        {
            /*
             * Instantiating the original class using the class factory.
             */
            var originalObject =
                Terrasoft.Core.Factories.ClassFactory.Get<OriginalClass>();

            /*
             * Getting the result value of the GetAmount() method. Values from page
             * fields are passed as the parameters.
             */
            int result = originalObject.GetAmount(value1, value2);

            /*
             * Returning the result
             */
            return string.Format("The result value, retrieved after calling the
replacement class method : {0}", result.ToString());
        }

        /*
         * Instantiating the replacement class using the replacement class
factory.
         */
        // The factory is given an instance of the constructor class argument as
an input parameter.
        var substObject =
            Terrasoft.Core.Factories.ClassFactory.Get<OriginalClass>(new
            Terrasoft.Core.Factories.ConstructorArgument("rateValue", 2));

        /*
         * Getting the result value of the GetAmount() method. Values from page
         */
    }
}
```

```
fields are passed as the parameters.
    int result = substObject.GetAmount(value1, value2);

    // Displaying the result on the page
    return string.Format("The result value, retrieved after calling the
replaceable class method: {0}", result.ToString());
    */

    // Instantiating the replacement class using new() operator.
    var substObjectByNew = new OriginalClass();

    // Instantiating the replacement class using the replacement class
factory.
    var substObjectByFactory =
Terrasoft.Core.Factories.ClassFactory.Get<OriginalClass>(new
Terrasoft.Core.Factories.ConstructorArgument("rateValue", 2));

    // Getting the result value of the GetAmount() method. The OriginalClass
method will be called without replacement.
    int resultByNew = substObjectByNew.GetAmount(value1, value2);

    // Getting the result value of the GetAmount() method. The
SubstituteClass method will be called. SubstituteClass is a replacement for
OriginalClass.
    int resultByFactory = substObjectByFactory.GetAmount(value1, value2);

    // Displaying the result on the page
    return string.Format("Result without class replacement: {0}; Result with
class replacement: {1}", resultByNew.ToString(), resultByFactory.ToString());
}
}
}
```

After making changes, save and publish the schema.

3. Implement a replacement class

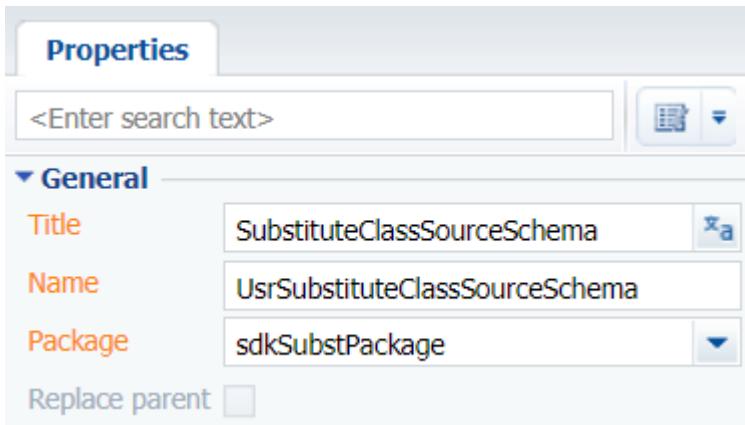
Add the custom package with the replaceable class as a dependency for the custom package with the replacement class.

Go to the [Advanced settings] section -> [Configuration] -> Custom package -> the [Schemas] tab. Click [Add] -> [Source code]. Learn more about creating a schema of the [Source Code] type in the “[Creating the \[Source code\] schema](#)” article.

Specify the following parameters for the created object schema (Fig. 8):

- [Title] – “SubstituteClassSourceSchema”.
- [Name] – "UsrSubstituteClassSourceSchema".

Fig. 3. – Setting up the [Source Code] type object schema



Implement a replacement class *SubstituteClass* that contains the *GetAmount(int, int)* method. The overloaded method adds the two values passed as parameters and multiplies the resulting value by the value passed in the *Rate* property. The initialization of the *Rate* property will be performed in the constructor of the replacement class. The complete source code of the module is available below:

```
namespace Terrasoft.Configuration
{
    [Terrasoft.Core.Factories.Override]
    public class SubstituteClass : OriginalClass
    {
        // Rate. The value of the property is assigned in the class.
        public int Rate { get; private set; }

        // The constructor runs a preliminary initialization of the Rate property
        // with a passed rate value.
        public SubstituteClass(int rateValue)
        {
            Rate = rateValue;
        }

        // Replacing a parent class with a custom implementation.
        public override int GetAmount(int substValue1, int substValue2)
        {
            return (substValue1 + substValue2) * Rate;
        }
    }
}
```

As a result, the new configuration service *AmountService* with a *GetAmount* endpoint will be available in the Creatio application. When accessing the endpoint of the service, such as using a browser, pass values *value1* and *value2*.

To demonstrate the operation of the configuration service with and without class replacement, insert the following string in the address bar:

<https://mycreatio.com/0/rest/AmountService/GetAmount?value1=25&value2=125>

The result of the *GetAmount(int, int)* method is displayed below.

```
{"GetAmountResult":"Result without class replacement: 150; Result with class
replacement: 300"}
```

Configuration web-services

Contents

- **Creating a user configuration service**
- **Creating anonymous web service**

- Calling configuration services with ServiceHelper
- Calling configuration services using Postman

Creating a user configuration service

Beginner

Easy

Medium

Advanced

Case description

Create a custom configuration service that returns Id of the contact of the given name. If there are several contacts found, return the Id of the first contact only. If no contacts are found, the service should return an empty string.

Source code

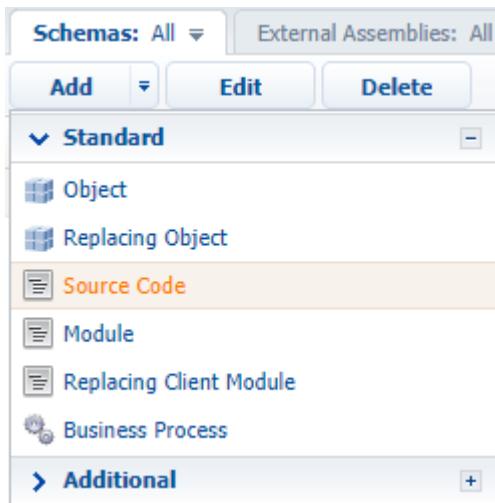
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Creating a [Source code] schema

Perform the [Add] – [Source code] action on the [Schemas] tab of the [Configuration] section.

Fig. 1. Adding the [Source Code] schema



Sett following properties for the schema:

- [Name] – UsrCustomConfigurationService.
- [Title] – UsrCustomConfigurationService.

2. Create class of the service

On the [Source code] tab:

- The namespace nested in the *Terrasoft.Configuration*. The name can be random, for example, the *UsrCustomConfigurationService*.
- Namespaces whose data types will be used in your class. For this, use the *using* directive. A complete list of namespaces is provided in the source code below.
- Class, for example, the *UsrCustomConfigurationService* class inherited from the *Terrasoft.Nui.ServiceModel.WebService.BaseService*. Mark the class with the [\[ServiceContract\]](#) and [\[AspNetCompatibilityRequirements\(RequirementsMode = AspNetCompatibilityRequirementsMode.Required\)\]](#) attributes.

The source code with a class declaration is available below:

3. Implement the methods that match the service endpoints

To implement the endpoint of the return of the contact Id by its name, add the `GetContactIdByName(string Name)` public string to the class. The `Name` parameter should receive the name of the contact. After accessing the database via the **EntitySchemaQuery ('Retrieving information with access right control. The EntitySchemaQuery class' in the on-line documentation)** the method will return the Id of the first found contact (or empty string) casted to string.

Full source code with the implementation of the service:

```
namespace Terrasoft.Configuration.UsrCustomNamespace
{
    using System;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using Terrasoft.Core;
    using Terrasoft.Web.Common;
    using Terrasoft.Core.Entities;

    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Required)]
    public class UsrCustomConfigurationService: BaseService
    {

        // Method returning contact identifier by name.
        [OperationContract]
        [WebInvoke(Method = "GET", RequestFormat = WebMessageFormat.Json, BodyStyle =
        WebMessageBodyStyle.Wrapped,
        ResponseFormat = WebMessageFormat.Json)]
        public string GetContactIdByName(string Name) {
            // Default result.
            var result = "";
            // The EntitySchemaQuery instance, addressing the Contact database table.
            var esq = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"Contact");
            // Adding columns to query.
            var colId = esq.AddColumn("Id");
            var colName = esq.AddColumn("Name");
            // Filtering query data.
            var esqFilter =
esq.CreateFilterWithParameters(FilterComparisonType.Equal, "Name", Name);
            esq.Filters.Add(esqFilter);
            // Receiving query results.
            var entities = esq.GetEntityCollection(UserConnection);
            // If the data are received.
            if (entities.Count > 0)
            {
                // Return the "Id" column value of the first record of the query
result.
                result = entities[0].GetColumnValue(colId.Name).ToString();
                // You can also use the below variant:
                // result = entities[0].GetTypedColumnValue<string>(colId.Name);
            }
            // Return result.
            return result;
        }
    }
}
```

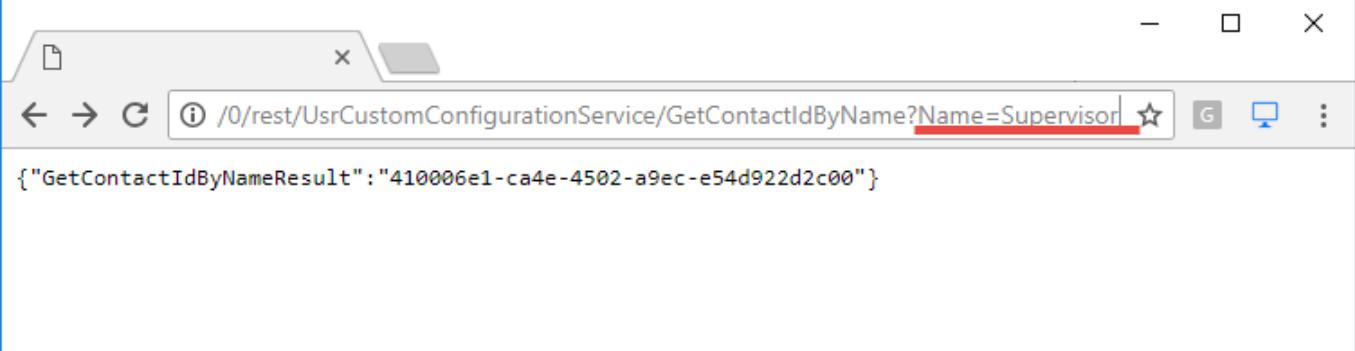
After making changes, save and publish the schema.

As a result, the new configuration service `UsrCustomConfigurationService` will be available in the Creatio. When the

GetContactIdByName endpoint of this service is called, for example, out of web browser, the contact Id (Fig. 2) or the empty string ("") value (Fig. 3) will be returned.

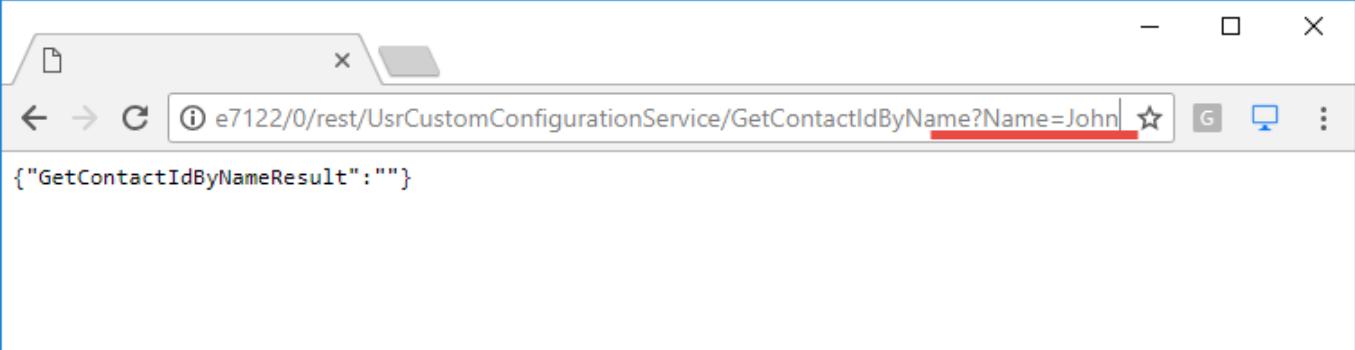
Pay attention to the format of the call result. In the server response, the object that contains property with the name that is a combination of the name of the called method and the "Return" suffix, will be passed. The value of the object property contains the contact Id (or an empty string) returned by the service.

Fig. 2. Request result: The contact Id is found.



A screenshot of a web browser window. The address bar shows the URL: /0/rest/UsrCustomConfigurationService/GetContactIdByName?Name=Supervisor. The main content area displays a JSON object: {"GetContactIdByNameResult": "410006e1-ca4e-4502-a9ec-e54d922d2c00"}

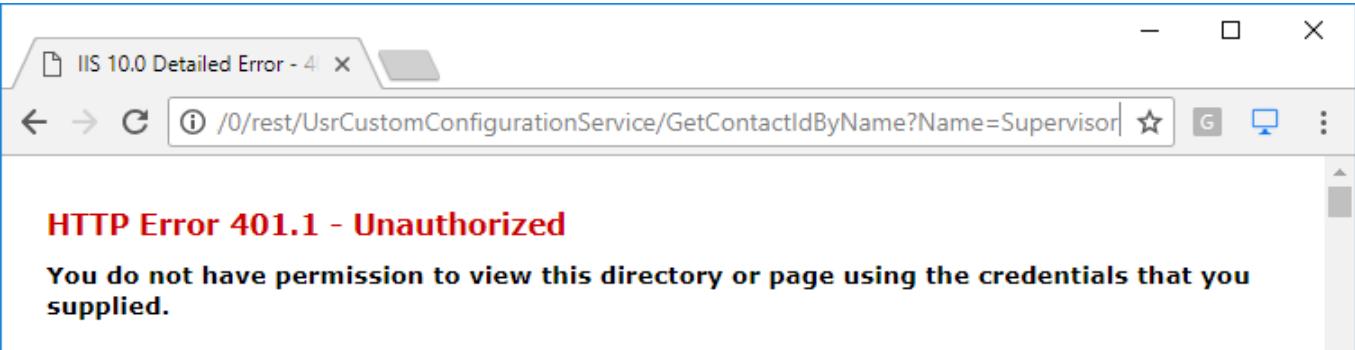
Fig. 3. Request result: The contact Id is not found.



A screenshot of a web browser window. The address bar shows the URL: e7122/0/rest/UsrCustomConfigurationService/GetContactIdByName?Name=John. The main content area displays a JSON object: {"GetContactIdByNameResult": ""}

If the service is called without logging to the application, the authorization error will be displayed (Fig. 4).

Fig. 4. Request result No authorization



A screenshot of a web browser window. The address bar shows the URL: /0/rest/UsrCustomConfigurationService/GetContactIdByName?Name=Supervisor. The main content area displays an error message: **HTTP Error 401.1 - Unauthorized**. You do not have permission to view this directory or page using the credentials that you supplied.

See also:

- [Creating a user configuration service](#)

Creating anonymous web service

Beginner

Easy

Medium

Advanced

Case description

Create custom configuration service that returns the Id of a contact by the provided name. If there are several contacts found, it is only necessary to return the Id of the first contact. If the contact is not found, the service should return an empty string.

You can use the service created based on the example covered in the “[Creating a user configuration service](#)” article as the configuration web service.

Case implementation algorithm

1. Creating configuration service

How to create the configuration service is covered in the “[Creating a user configuration service](#)” article.

Because you are creating an anonymous configuration service, system user connection must be used instead of user connection.

The source code of configuration service, which uses system user connection:

```
namespace Terrasoft.Configuration.UsrCustomConfigurationService
{
    using System;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using Terrasoft.Core;
    using Terrasoft.Web.Common;
    using Terrasoft.Core.Entities;

    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
    AspNetCompatibilityRequirementsMode.Required)]
    public class UsrCustomConfigurationService: BaseService
    {
        // Link to the UserConnection instance required to access the database.
        private SystemUserConnection _systemUserConnection;
        private SystemUserConnection SystemUserConnection {
            get {
                return _systemUserConnection ?? (_systemUserConnection =
    (SystemUserConnection)AppConnection.SystemUserConnection);
            }
        }

        // A method that returns the contact's ID by its name.
        [OperationContract]
        [WebInvoke(Method = "GET", RequestFormat = WebMessageFormat.Json, BodyStyle =
    WebMessageBodyStyle.Wrapped,
        ResponseFormat = WebMessageFormat.Json)]
        public string GetContactIdByName(string Name) {
            // The default result.
            var result = "";
            // An EntitySchemaQuery instance that accesses the Contact table of the
            database.
            var esq = new EntitySchemaQuery(SystemUserConnection.EntitySchemaManager,
    "Contact");
            // Adding columns to the query.
            var colId = esq.AddColumn("Id");
            var colName = esq.AddColumn("Name");
            // Filter the query data.
            var esqFilter =
esq.CreateFilterWithParameters(FilterComparisonType.Equal, "Name", Name);
            esq.Filters.Add(esqFilter);
            // Get the result of the query.
            var entities = esq.GetEntityCollection(SystemUserConnection);
```

```
// If the data is received.
if (entities.Count > 0)
{
    // Return the value of the "Id" column of the first record of the
query result.

    result = entities[0].GetColumnValue(colId.Name).ToString();
    // You can also use this option:
    // result = entities [0]. GetTypedColumnValue <string> (colId.Name);
}
// Return the result.
return result;
}
```

2. Registering the WCF-service.

Create the *UsrCustomConfigurationService.svc* file in the ..\Terrasoft.WebApp\ServiceModel catalog and add the following record into it:

```
<%@ ServiceHost Language="C#" Debug="true"
Service="Terrasoft.Configuration.UsrCustomConfigurationService.UsrCustomConfiguration
Service" %>
```

In the *Service* attribute specify the full name of the configuration service class. Read more about the @ServiceHost WCF-directive in [Microsoft documentation](#).

3. Configuring WCF-service for the http and https protocols.

Add the following record to the *services.config* files located at ..\Terrasoft.WebApp\ServiceModel\http and ..\Terrasoft.WebApp\ServiceModel\https catalogs:

```
<services>
    ...
    <service name="Terrasoft.Configuration.[Service name]">
        <endpoint name="[Service name]EndPoint"
            address=""
            binding="webHttpBinding"
            behaviorConfiguration="RestServiceBehavior"
            bindingNamespace="http://Terrasoft.WebApp.ServiceModel"
            contract="Terrasoft.Configuration.[Service name]" />
    </service>
</services>
```

Configure the service here. The *<services>* element contains a list of configurations of all application services (the *<service>* nested elements). The *name* attribute contains the name of type (class or interface) implementing the service contract. The *<endpoint>* nested element requires address, binding and interface that define the service contract specified in the *name* attribute of the *<service>* element.

You can find detailed description of the service configuration elements in the [documentation](#).

4. Setting up access to WCF-service for all users.

Perform the following changes in the ..\Terrasoft.WebApp\Web.config file:

- Add the *<location>* element defining the relative path and access rights to the service.
- In the *<appSettings>* element change the *value* value for the “AllowedLocations” key by adding the relative path to the service into it.

An example of changes in the ..\Terrasoft.WebApp\Web.config file:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    ...

```

```
<location path="ServiceModel/UsrCustomConfigurationService.svc">
  <system.web>
    <authorization>
      <allow users="*" />
    </authorization>
  </system.web>
</location>
...
<appSettings>
  ...
  <add key="AllowedLocations" value="[Previous
values];ServiceModel/UsrCustomConfigurationService.svc" />
  ...
</appSettings>
...
</configuration>
```

After reloading the application pool in IIS, the service will become available at:

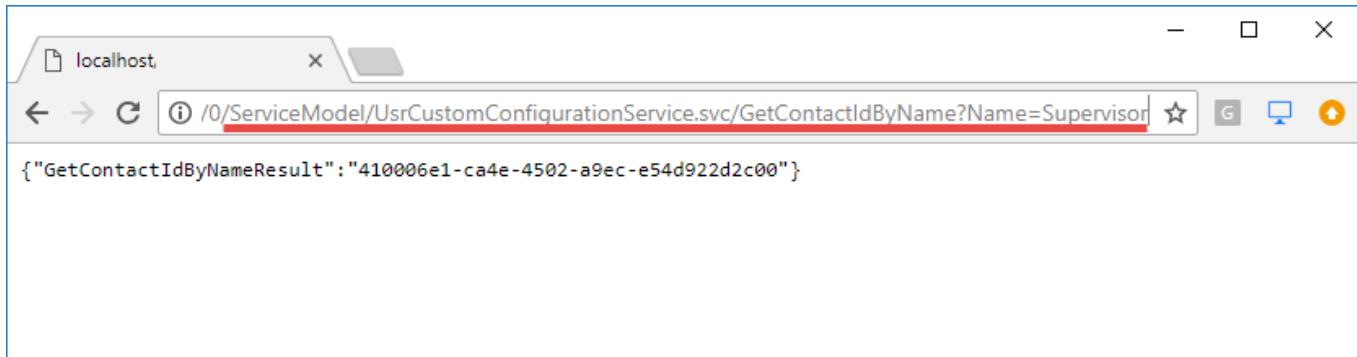
[Application Address]/0/ServiceModel/[Custom Service Name].svc/[Custom Service Endpoint]?[Optional parameters]

For example:

<http://mysite.creatio.com/0/ServiceModel/UsrCustomConfigurationService.svc/GetContactIdByName?Name=Supervisor>

You can address the service, e.g., from a browser (fig.1) either with preliminary login or without it.

Fig. 1. Example of access to the anonymous service from browser.



See also:

- [Creating anonymous web service](#)

Calling configuration services with ServiceHelper

Beginner

Easy

Medium

Advanced

Case description

Add a button calling the configuration web service to the new contact adding page. As a result, the information window of the page will display the result returned by the web service method.

The procedure for calling a web service from a client JavaScript code is available in the "[Calling configuration services with ServiceHelper](#)" article.

Source code

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

Case implementation algorithm

1. Creating configuration service

The current case uses the web service from the “**Creating a user configuration service**” article. The “Method” parameter of the “WebInvoke” attribute is changed for POST in the service.

Service source code:

```
namespace Terrasoft.Configuration.UsrConfigurationService
{
    using System;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using Terrasoft.Core;
    using Terrasoft.Web.Common;
    using Terrasoft.Core.Entities;

    [ServiceContract]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class UsrConfigurationService: BaseService
    {
        // Link to the UserConnection instance needed for addressing the database.
        private SystemUserConnection _systemUserConnection;
        private SystemUserConnection SystemUserConnection {
            get {
                return _systemUserConnection ?? (_systemUserConnection =
(SystemUserConnection)AppConnection.SystemUserConnection);
            }
        }

        // Method returning the contact identifier by name.
        [OperationContract]
        [WebInvoke(Method = "POST", RequestFormat = WebMessageFormat.Json, BodyStyle =
= WebMessageBodyStyle.Wrapped,
        ResponseFormat = WebMessageFormat.Json)]
        public string GetContactIdByName(string Name) {
            // Default result.
            var result = "";
            // The EntitySchemaQuery instance, addressing the Contact database table.
            var esq = new EntitySchemaQuery(SystemUserConnection.EntitySchemaManager,
"Contact");
            // Adding columns to query.
            esq.AddColumn("Id");
            var colName = esq.AddColumn("Name");
            // Query data filtering.
            var esqFilter =
esq.CreateFilterWithParameters(FilterComparisonType.Equal, "Name", Name);
            esq.Filters.Add(esqFilter);
            // Receiving the query results.
            var entities = esq.GetEntityCollection(SystemUserConnection);
            // If data are received.
            if (entities.Count > 0)
            {
                // Return the "Id" column value of the first query result record.
                result = entities[0].GetColumnValue(colId.Name).ToString();
                // You can also use the folowing variant:
                // result = entities[0].GetTypedColumnValue<string>(colId.Name);
            }
            // Return result.
        }
    }
}
```

```
        return result;  
    }  
}
```

2. Creating a replacing edit page

Create a replacing client module and specify [Display schema – Contact card] (*ContactPageV2*) as the parent object in the custom package (Fig. 1). Creating a replacing page is covered in the “[Creating a custom client module schema](#)” article.

Add the *ServiceHelper* module as a dependency to declaring the edit page module.

2. Adding a button to the edit page

Adding a button to the edit page is described in the “[How to add a button to an edit page in the new record add mode](#)” and “[How to add the button on the edit page in the combined mode](#)” articles.

Add a localizable string with the button caption to the replacing module schema of the contact edit page, for example:

- [Name] – “GetServiceInfoButtonCaption”
 - [Value] – “Call service”

3. Adding the button handler and calling the web service method

Use the `callService()` method of the `ServiceHelper` module to call the web service and pass the following values as parameters:

- name of the configuration service class (*UsrCustomConfigurationService*)
 - name of the called service method (*GetContactIdByName*)
 - the object with the initialized incoming parameters for the service method (*serviceData*)
 - the callback function where processing of service results is executed
 - execution context

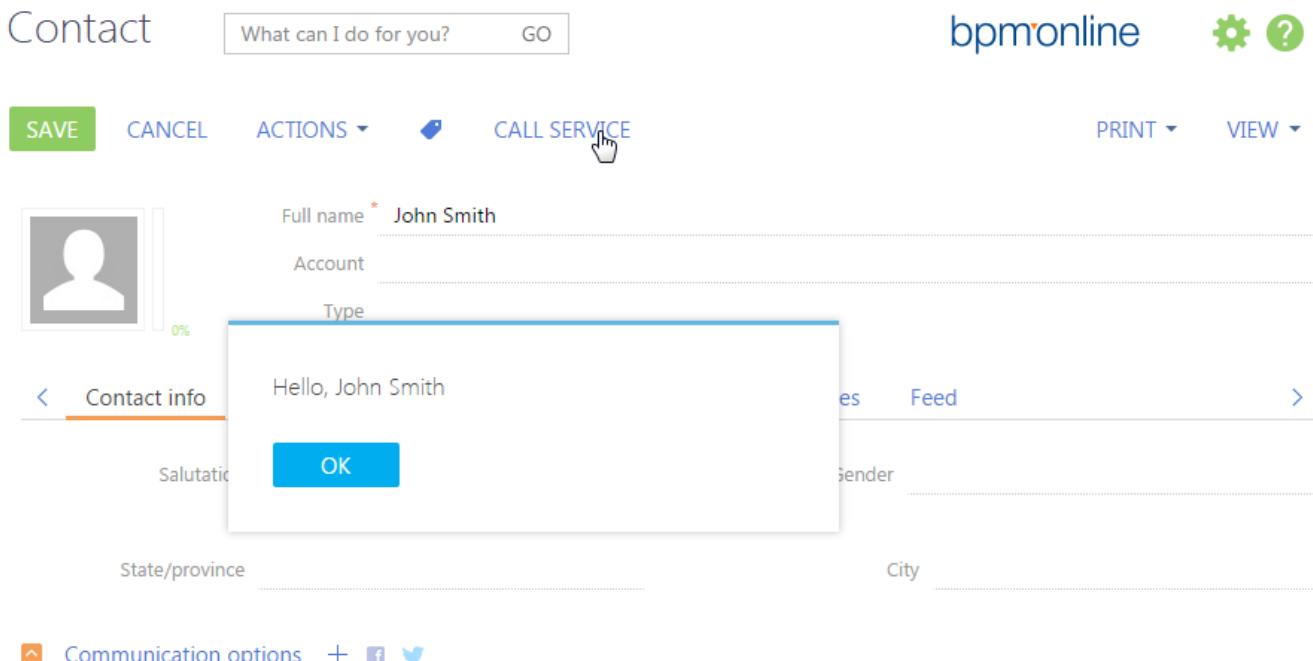
The source code of the edit page replacing module:

```
define("ContactPageV2", ["ServiceHelper"],  
    function(ServiceHelper) {  
        return {  
            // Name of the edit page object schema.  
            entitySchemaName: "Contact",  
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,  
            // View model methods of the edit page.  
            methods: {  
                // Verifies if the [Full name] field is populated on the page.  
                isContactNameSet: function() {  
                    return this.get("Name") ? true : false;  
                },  
                // Handler-method of clicking the button.  
                onGetServiceInfoClick: function() {  
                    var name = this.get("Name");  
                    // Object initializing incoming parameters for the service  
method.  
                    var serviceData = {  
                        // The property name corresponds to the incoming parameter  
name of the service method.  
                        Name: name  
                    };  
                    // Calling the web service and processing the results.  
                    ServiceHelper.callService("UsrConfigurationService",  
"GetContactIdByName",  
                        function(response) {
```

```
        var result = response.GetContactIdByNameResult;
        this.showInformationDialog(result);
    }, serviceData, this);
}
},
diff: /**SCHEMA_DIFF*[
// Metadata for adding a custom button to a page.
{
    // Executing the operation of adding the element to page.
    "operation": "insert",
    // Name of the parent control element where the button is added.
    "parentName": "LeftContainer",
    // The button is added to the control element collection
    // of the parent element (whose metaname is specified in
parentName).
    "propertyName": "items",
    // Name of the added button.
    "name": "GetServiceInfoButton",
    // Additional field property.
    "values": {
        // The added element type - button.
        itemType: Terrasoft.ViewItemType.BUTTON,
        // Binding the button caption to the localizable schema
string.
        caption: {bindTo:
"Resources.Strings.GetServiceInfoButtonCaption"},
        // Binding the handler-method of clicking the button.
        click: {bindTo: "onGetServiceInfoClick"},
        // Binding the "enabled" property of the button.
        enabled: {bindTo: "isContactNameSet"},
        // Field location setup.
        "layout": {"column": 1, "row": 6, "colSpan": 2, "rowSpan": 1}
    }
}
]/**SCHEMA_DIFF*/
);
});
});
```

After you save the schema and update the application page, the [Call service] button will appear on the contact edit page. When you click the button, the configuration service method will be called (fig. 1).

Fig. 1. Case implementation result



See also:

- **Calling configuration services with ServiceHelper**

Calling configuration services using Postman

Beginner Easy **Medium** Advanced

Introduction

To integrate with Creatio **configuration services** you need to execute HTTP requests to these services. Requests can be compiled in any programming language: C#, PHP, etc. However, it is recommended to use HTTP request debugging tools, such as [Postman](#) or [Fiddler](#) for better understanding of general principles for request formatting. This article provides examples of requests to Creatio configuration services using Postman.

An example of a request to Creatio service using Fiddler is described in the “**Executing OData queries using Fiddler**” article.

Preliminary settings

This article uses a custom configuration service created as described in the “**Creating a user configuration service**” article. Use the following [link](#) to access the package for setting up configuration service to your application. The package setup procedure is covered in the “**Transferring changes using packages export and import**” article.

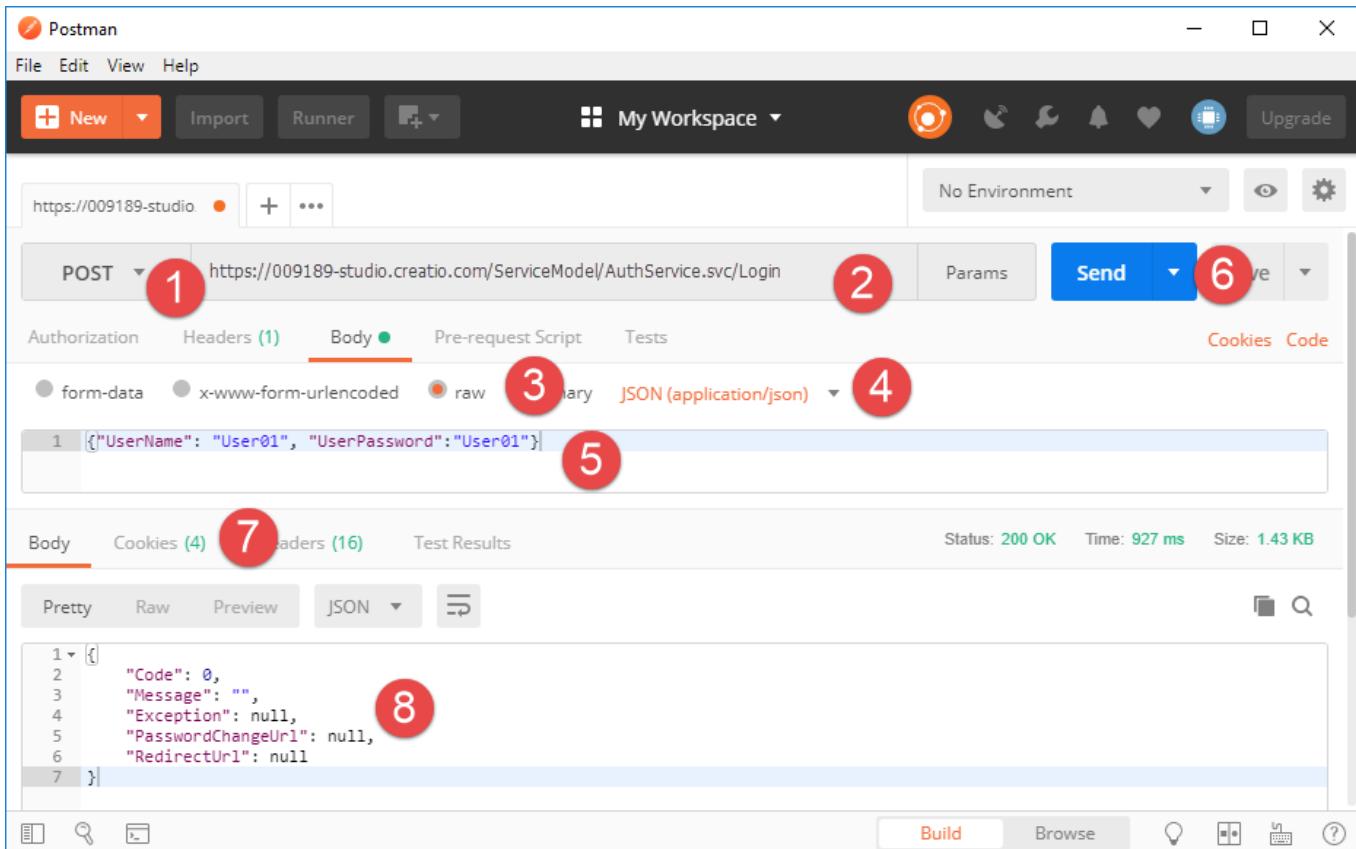
Authentication

Before making requests to configuration service, a third party application must be authenticated and receive the corresponding *cookies*. Creatio authentication uses a separate *AuthService.svc* web service (see “**Authentication of external requests**”).

Authentication is not needed to access the **anonymous services**.

To execute a request to *AuthService.svc* using Postman, do the following (Fig. 1):

Fig. 1. Authentication service request



1. Select the POST HTTP method.

2. Specify the authentication service URL. URL is generated according to the following mask:

`http(s)://[Creatio application address]/ServiceModel/AuthService.svc/Login`

Example:

`https://009189-studio.creatio.com/ServiceModel/AuthService.svc/Login`

3. Select the “raw” method of request body generation on the [Body] tab.

4. Specify the request body type (JSON (application/json)).

5. Add the request body – a JSON object with the authentication data (login and password):

```
{"UserName": "User01", "UserPassword": "User01"}
```

6. Click the [Send] button to send the request to the service.

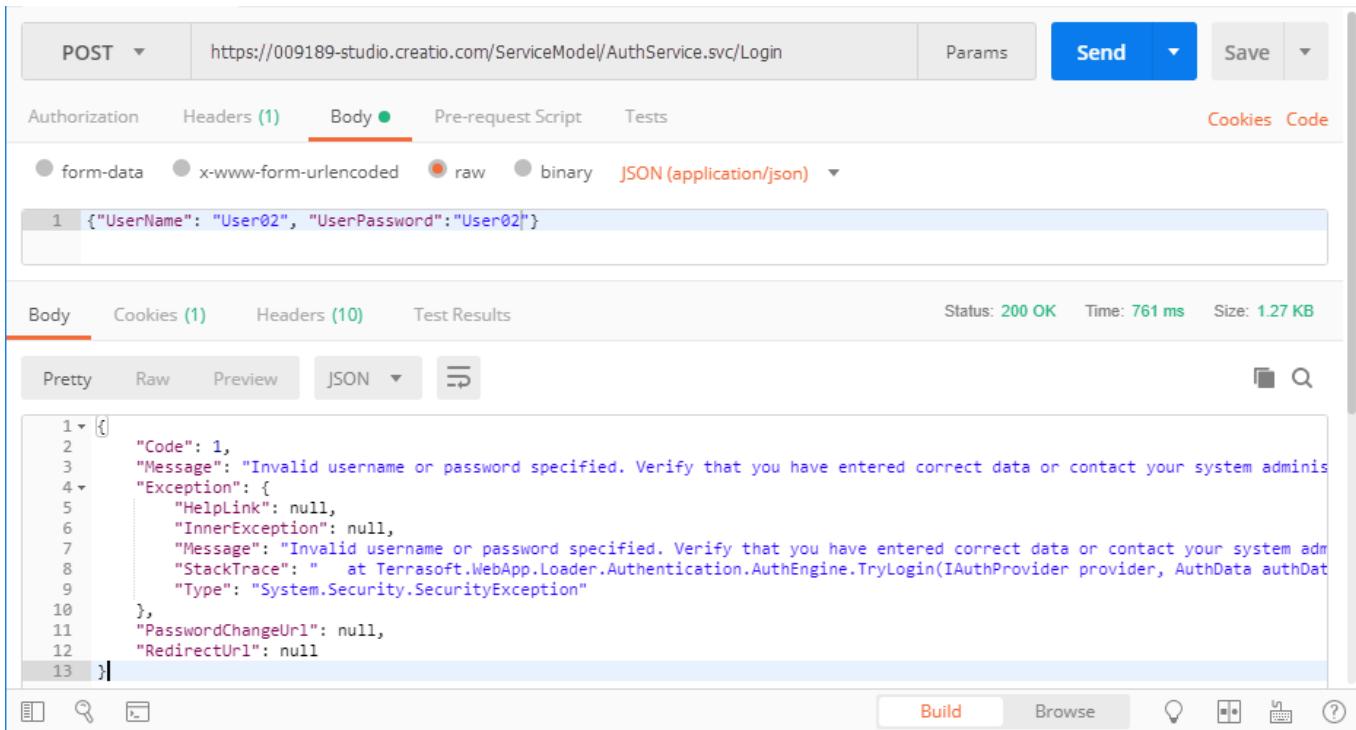
The *cookies* received in the HTTP response (BPMLOADER, .ASPXAUTH, BPMCSRF and UserName) are to be used in all further requests to Creatio services that require authentication data. You can view the received *cookies* on the [Cookies] tab (Fig. 1, 7).

Using the BPMCSRF *cookie* and BPMCSRF token (see below) is required when protection from CSRF attacks is enabled. For more information, see “**Authentication of external requests**”. If BPMCSRF token is not specified, the server will return error with code 403.

Protection from CSRF attacks is disabled on Creatio trial websites. Therefore, there is no need to use both BPMCSRF *cookie* and token in the request titles.

If the authentication has been successful, the response body will contain a JSON object whose *Code* property will be set to “0” (Fig. 1, 8). In case of errors, JSON object properties will contain corresponding code and message. For example, if a user specified in step 5 is not added to the application, the authentication service will return an incorrect login and password error (Fig. 2).

Fig. 2. Authentication service request containing invalid user data

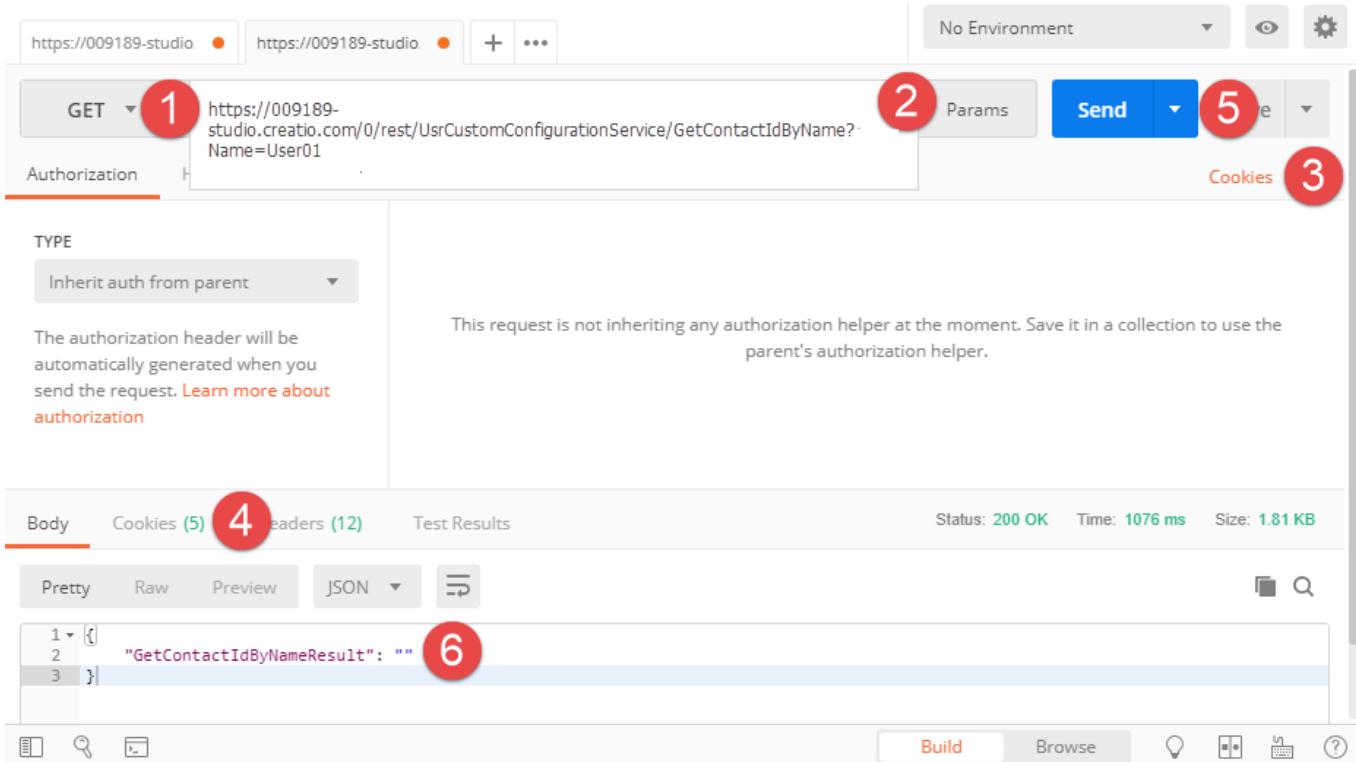


Configuration service request

The *UsrCustomConfigurationService* configuration service used in this article (see “Preliminary settings”, “**Creating a user configuration service**”) can only process HTTP requests via the GET method. Such requests do not have any body. Add the corresponding request body (for example, as described in step 5 of the “Authentication” section), if you need to execute other types of requests.

To generate the *UsrCustomConfigurationService* configuration service request (Fig. 3):

Fig. 3. Configuration service request



1. Select the GET HTTP method.

2. Specify the configuration service URL. URL is generated according to the following mask:

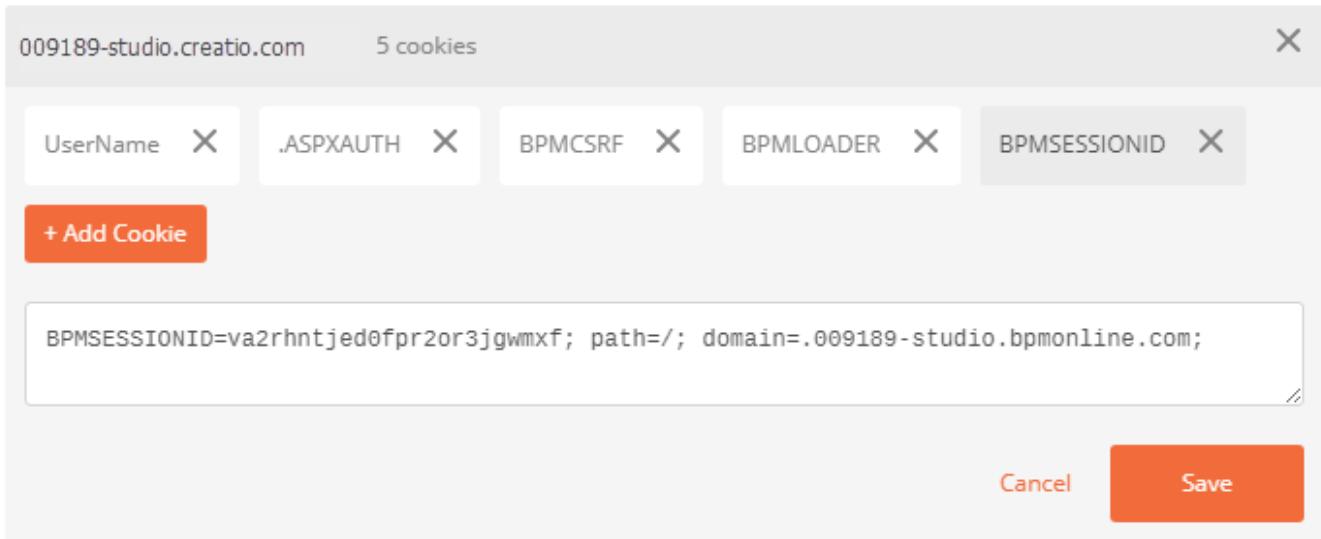
[Application Address]/0/rest/[Configuration Service Name]/[Custom Service Endpoint]?
[Optional Parameters]

Example:

```
https://009189-studio.creatio.com/0/rest/UsrCustomConfigurationService/GetContactIdByName?  
Name=User01
```

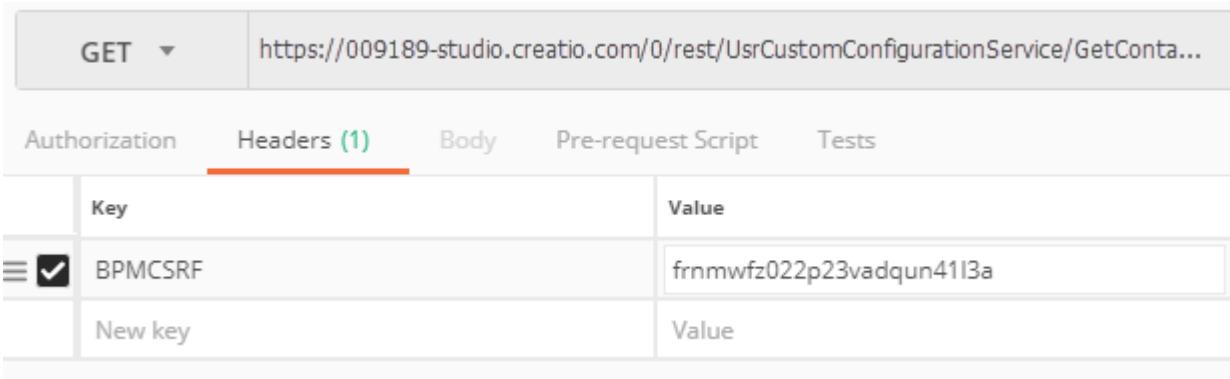
3. Add the necessary cookies BPMLOADER, .ASPxAUTH, BPMCSRF and UserName) received in the HTTP authentication service response (Fig. 4).

Fig. 4. Adding cookie to a request



Using the BPMCSRF cookie and BPMCSRF token (see below) is required when protection from CSRF attacks is enabled. Add the “key-value” pair to the request caption. Use “BPMCSRF” as a key and the BPMCSRF cookie value as a value (Fig. 5).

Fig. 5. Adding the BPMCSRF token to the request



4. Add the BPMSESSIONID cookie to all requests except for the first one after the authentication.

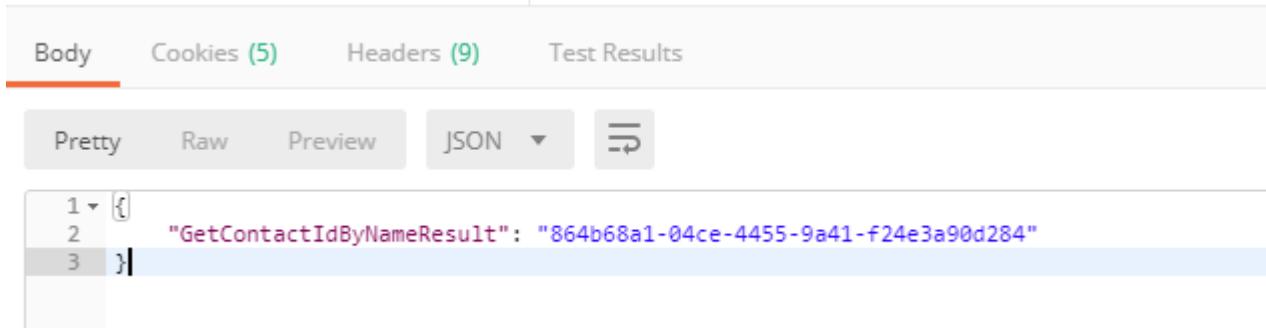
User session will only be created when executing the first configuration service request. The BPMSESSIONID cookie will be returned in the HTTP response (see the [Cookies] tab of the HTTP response fig. 3, 4). Therefore, there is no need to add the BPMSESSIONID cookie to the title of the first request (Fig. 6, 4).

If you do not add the BPMSESSIONID cookie to each subsequent request, each new request will create a new user session. Significant frequency of requests (several or more requests per minute) will increase the RAM consumption which will decrease the performance.

5. Click the [Send] button to send the request to the service.

If the contact specified in the Name parameter is not detected in Creatio, the "GetContactIdByNameResult" property of the JSON object that was returned in the HTTP response will contain the "" value (fig. 3, 4). If the contact exists, the service will return its identifier (fig. 6).

Fig. 6. Request result



The screenshot shows the Postman interface with the 'Body' tab selected. The response body is displayed in JSON format:

```
1 {  
2   "GetContactIdByNameResult": "864b68a1-04ce-4455-9a41-f24e3a90d284"  
3 }
```

See also:

- [Working with requests in Postman](#)
- [Working with request collections in Postman](#)

CRUD operations

Contents

- [Retrieving data from the database](#)
- [Retrieving data based on user permissions](#)
- [Adding data](#)
- [Adding data using subqueries](#)
- [Modifying data](#)
- [Deleting data](#)
- [Working with database entities](#)

Retrieving data from the database

Examples of using the Select class to retrieve data from the database

You can download the package with the **configuration web service** ('Creating a user configuration service' in the [on-line documentation](#)) implementing the cases described below using the following [link](#).

The query result handler method used in the cases

```
private string CreateJson(IDataReader dataReader)  
{  
    var list = new List<dynamic>();  
    var cnt = dataReader.FieldCount;  
    var fields = new List<string>();  
    for (int i = 0; i < cnt; i++)  
    {  
        fields.Add(dataReader.GetName(i));  
    }  
    while (dataReader.Read())  
    {  
        dynamic exo = new System.Dynamic.ExpandoObject();
```

```
        foreach (var field in fields)
        {
            ((IDictionary<String, Object>)exo).Add(field,
dataReader.GetColumnValue(field));
        }
        list.Add(exo);
    }
    return JsonConvert.SerializeObject(list);
}
```

Example 1

Receive the SQL query text.

```
public string GetSqlTextExample()
{
    var result = "";
    var select = new Select(UserConnection)
        .Column(Column.Asterisk())
        .From("Contact");
    result = select.GetSqlText();
    return result;
}
```

Example 2

Select a certain number of records from the needed table (object schema).

```
public string SelectColumns(string tableName, int top)
{
    top = top > 0 ? top : 1;
    var result = "{}";
    var select = new Select(UserConnection)
        .Top(top)
        .Column(Column.Asterisk())
        .From(tableName);
    using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection())
    {
        using (IDataReader dataReader =
select.ExecuteReader(dbExecutor))
        {
            result = CreateJson(dataReader);
        }
    }
    return result;
}
```

Example 3

```
public string SelectContactsYoungerThan(string birthYear)
```

```
{  
    var result = "{}";  
    var year = DateTime.ParseExact(birthYear, "yyyy",  
CultureInfo.InvariantCulture);  
    var select = new Select(UserConnection)  
        .Column("Id")  
        .Column("Name")  
        .Column("BirthDate")  
    .From("Contact")  
    .Where("BirthDate").IsGreater(Column.Parameter(year))  
        .Or("BirthDate").IsNull()  
    .OrderByDesc("BirthDate")  
        as Select;  
    using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection())  
    {  
        using (IDataReader dataReader =  
select.ExecuteReader(dbExecutor))  
        {  
            result = CreateJson(dataReader);  
        }  
    }  
    return result;  
}
```

Example 4

```
public string SelectContactsYoungerThanAndHasAccountId(string  
birthYear)  
{  
    var result = "{}";  
    var year = DateTime.ParseExact(birthYear, "yyyy",  
CultureInfo.InvariantCulture);  
    var select = new Select(UserConnection)  
        .Column("Id")  
        .Column("Name")  
        .Column("BirthDate")  
    .From("Contact")  
    .Where()  
    .OpenBlock("BirthDate").IsGreater(Column.Parameter(year))  
        .Or("BirthDate").IsNull()  
    .CloseBlock()  
    .And("AccountId").Not().IsNull()  
    .OrderByDesc("BirthDate")  
        as Select;  
    using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection())  
    {  
        using (IDataReader dataReader =  
select.ExecuteReader(dbExecutor))  
        {  
            result = CreateJson(dataReader);  
        }  
    }  
}
```

```
        }
    }
    return result;
}
```

Example 5

```
public string SelectContactsJoinAccount()
{
    var result = "{}";
    var select = new Select(UserConnection)
        .Column("Contact", "Id").As("ContactId")
        .Column("Contact", "Name").As("ContactName")
        .Column("Account", "Id").As("AccountId")
        .Column("Account", "Name").As("AccountName")
        .From("Contact")
        .Join(JoinType.Inner, "Account")
        .On("Contact", "Id").IsEqual("Account", "PrimaryContactId")
        as Select;
    using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection())
    {
        using (IDataReader dataReader =
select.ExecuteReader(dbExecutor))
        {
            result = CreateJson(dataReader);
        }
    }
    return result;
}
```

Example 6

Select the identifiers and names of the contacts that are primary for the accounts

```
public string SelectAccountPrimaryContacts()
{
    var result = "{}";
    var select = new Select(UserConnection)
        .Column("Id")
        .Column("Name")
        .From("Contact").As("C")
        .Where()
        .Exists(new Select(UserConnection)
            .Column("A", "PrimaryContactId")
            .From("Account").As("A")
            .Where("A", "PrimaryContactId").IsEqual("C",
"Id"))
        as Select;
    using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection())
    {
```

```
        using (IDataReader dataReader =
select.ExecuteReader(dbExecutor))
{
    result = CreateJson(dataReader);
}
}
return result;
}
```

Example 7

```
public string SelectCountriesWithCitiesCount(int count)
{
    var result = "{}";
    var select = new Select(UserConnection)
        .Column(Func.Count("City", "Id")).As("CitiesCount")
        .Column("Country", "Name").As("CountryName")
        .From("City")
        .Join(JoinType.Inner, "Country")
            .On("City", "CountryId").IsEqual("Country", "Id")
        .GroupBy("Country", "Name")
        .Having(Func.Count("City",
"Id")).IsGreater(Column.Parameter(count))
        .OrderByDesc("CitiesCount")
        as Select;
    using (DBExecutor dbExecutor = UserConnection.EnsureDBConnection())
    {
        using (IDataReader dataReader =
select.ExecuteReader(dbExecutor))
        {
            result = CreateJson(dataReader);
        }
    }
    return result;
}
```

Example 8

Receive the contact identifier by its name

```
public string SelectCountryIdByCityName(string CityName)
{
    var result = "";
    var select = new Select(UserConnection)
        .Column("CountryId")
        .From("City")
        .Where("Name").IsEqual(Column.Parameter(CityName)) as Select;
    result = select.ExecuteScalar<Guid>().ToString();
    return result;
}
```

See also:

- [The Select class](#)

Retrieving data based on user permissions

Examples of using EntitySchemaQuery to retrieve data

You can download the package with the configuration web service implementing the cases described below using the following [link](#).

Example 1

Creating the EntitySchemaQuery instance

```
public string CreateESQ()
{
    var result = "";
    // Receiving an instance of the object schema manager.
    EntitySchemaManager esqManager =
SystemUserConnection.EntitySchemaManager;
    // Receiving an instance of the schema that will be the root schema
for the created
    // EntitySchemaQuery instance.
    var rootEntitySchema = esqManager.GetInstanceByName("City") as
EntitySchema;
    // Creating the EntitySchemaQuery instance that will have
    // rootEntitySchema as the root schema.
    var esqResult = new EntitySchemaQuery(rootEntitySchema);
    // Adding columns that will be selected in the result request.
    esqResult.AddColumn("Id");
    esqResult.AddColumn("Name");
    // Receiving the Select instance associated with the created
EntitySchemaQuery request.
    Select selectEsq = esqResult.GetSelectQuery(SystemUserConnection);
    // Receiving the text of the resulting request of the created
EntitySchemaQuery instance.
    result = selectEsq.GetSqlText();
    return result;
}
```

Example 2

Creating a clone of the EntitySchemaQuery instance

```
public string CreateESQClone()
{
    var result = "";
    EntitySchemaManager esqManager =
SystemUserConnection.EntitySchemaManager;
    var esqSource = new EntitySchemaQuery(esqManager, "Contact");
    esqSource.AddColumn("Id");
    esqSource.AddColumn("Name");
```

```
// Creating an EntitySchemaQuery instance that is a clone of the
esqSource instance.
var esqClone = new EntitySchemaQuery(esqSource);
result =
esqClone.GetSelectQuery(SystemUserConnection).GetSqlText();
return result;
}
```

Example 3

Receiving the result of query execution

```
public string GetEntitiesExample()
{
    var result = "";
    // Creating a request to the City schema, adding the Name column to
    // the request.
    var esqResult = new
EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");
    var colName = esqResult.AddColumn("Name");

    // Running the request to the database and receiving the resulting
    // object collection.
    var entities = esqResult.GetEntityCollection(UserConnection);
    for (int i=0; i < entities.Length; i++) {
        result += entities[i].GetColumnValue(colName.Name).ToString();
        result += "\n";
    }

    // Running the request to the database and receiving an object with
    // the specified identifier.
    var entity = esqResult.GetEntity(UserConnection, new
Guid("100B6B13-E8BB-DF11-B00F-001D60E938C6"));
    result += "\n";
    result += entity.GetColumnValue(colName.Name).ToString();
    return result;
}
```

Example 4

Example of working with the EntitySchemaQuery cache

```
public Collection<string> UsingCacheExample()
{
    // Creating a request to the City schema, adding the Name column to
    // the request.
    var esqResult = new
EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");
    esqResult.AddColumn("Name");

    // Identifying the key that will be used to store the request
    // results in the cache.
```

```
// Creatio cache of the local data cache session level acts as a
cache (since the
// Cache property of the object cannot be identified).
esqResult.CacheItemName = "EsqResultItem";

// Collection that will contain the request results.
var esqCityNames = new Collection<string>();

// Collection that will contain the cached request results.
var cachedEsqCityNames = new Collection<string>();

// Running the request to the database and receiving the resulting
object collection.
// After running the operation the results will be cached.
var entities = esqResult.GetEntityCollection(UserConnection);

// Processing the request results and populating the esqCityNames
collection.
foreach (var entity in entities)
{
    esqCityNames.Add(entity.GetTypedColumnValue<string>("Name"));
}

// Receiving a link to the cache of the esqResult request using the
CacheItemName key as a data table in memory.
var esqCacheStore = esqResult.Cache[esqResult.CacheItemName] as
DataTable;

// Populating the cachedEsqCityNames collection with the values of
the request cache.
if (esqCacheStore != null)
{
    foreach (DataRow row in esqCacheStore.Rows)
    {
        cachedEsqCityNames.Add(row[0].ToString());
    }
}
return cachedEsqCityNames;
}
```

Example 5

Using additional settings of the query

```
public Collection<string> ESQOptionsExample()
{
    // Creating a request instance with the City root schema.
    var esqCities = new
EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");
    esqCities.AddColumn("Name");

    // Creating a request with the Country root schema.
```

```
var esqCountries = new EntitySchemaQuery(UserConnection.EntitySchemaManager, "Country");
esqCountries.AddColumn("Name");

// Creating an instance of settings to return the first 5 strings by the request.
var esqOptions = new EntitySchemaQueryOptions()
{
    PageableDirection = PageableSelectDirection.First,
    PageableRowCount = 5,
    PageableConditionValues = new Dictionary<string, object>()
};

// Receiving a city collection that will contain the first 5 cities of the resulting data set.
var cities = esqCities.GetEntityCollection(UserConnection, esqOptions);

// Receiving a country collection that will contain the first 5 countries of the resulting data set.
var countries = esqCountries.GetEntityCollection(UserConnection, esqOptions);
    var esqStringCollection = new Collection<string>();
foreach (var entity in cities)
{
    esqStringCollection.Add(entity.GetTypedColumnValue<string>("Name"));
}
foreach (var entity in countries)
{
    esqStringCollection.Add(entity.GetTypedColumnValue<string>("Name"));
}
return esqStringCollection;
}
```

See also:

- **The EntitySchemaQuery class**

Adding data

Основы Легкий Средний **Сложный**

Examples of using the Insert class to build queries for data insertion

You can download the package with the configuration web service implementing the cases described below using the following [link](#).

The example code demonstrates available methods for passing parameters to the query. When developing a project, be aware that parameters originating from the user should never be passed to the *Column.Const* method, since this can lead to a successful SQL injection attack.

Example 1

Receive the SQL-query text

```
public string GetSqlTextExample(string ContactName)
{
    var result = "";
    var id = Guid.NewGuid();
    var ins = new Insert(UserConnection)
        .Into("Contact")
        .Set("Id", Column.Parameter(id))
        .Set("Name", Column.Parameter(ContactName));
    result = ins.GetSqlText();
    return result;
}
```

Example 2

Add the contact with the specified name

```
public string InsertContact(string contactName)
{
    contactName = contactName ?? "Unknown contact";
    var ins = new Insert(UserConnection)
        .Into("Contact")
        .Set("Name", Column.Parameter(contactName));
    var affectedRows = ins.Execute();
    var result = $"Inserted new contact with name '{contactName}'.
{affectedRows} rows affected";
    return result;
}
```

Example 3

Add the city with the specified name and connect it to the specified country

```
public string InsertCity(string city, string country)
{
    city = city ?? "unknown city";
    country = country ?? "unknown country";

    var ins = new Insert(UserConnection)
        .Into("City")
        .Set("Name", Column.Parameter(city))
        .Set("CountryId",
            new Select(UserConnection)
                .Top(1)
                .Column("Id")
                .From("Country")
                .Where("Name")
                    .IsEqual(Column.Parameter(country)));
    var affectedRows = ins.Execute();
    var result = $"Inserted new city with name '{city}' located in
'{country}'. {affectedRows} rows affected";
```

```
        return result;
    }
```

See also

- **The Insert class**

Adding data using subqueries

Examples of using the InsertSelect class to retrieve data using subqueries

You can download the package with the configuration web service implementing the cases described below using the following [link](#).

The example code demonstrates available methods for passing parameters to the query. When developing a project, be aware that parameters originating from the user should never be passed to the *Column.Const* method, since this can lead to a successful *SQL* injection attack.

Example 1

Receive the SQL-query text

```
public string GetSqlTextExample(string contactName, string accountName)
{
    var result = "";
    var id = Guid.NewGuid();
    var selectQuery = new Select(UserConnection)
        .Column(Column.Parameter(contactName))
        .Column("Id")
        .From("Account")
        .Where("Name").IsEqual(Column.Parameter(accountName)) as
Select;
    var insertSelectQuery = new InsertSelect(UserConnection)
        .Into("Contact")
        .Set("Name", "AccountId")
        .FromSelect(selectQuery);

    result = insertSelectQuery.GetSqlText();
    return result;
}
```

Example 2

Add the contact with the specified name and account

```
public string InsertContactWithAccount(string contactName, string
accountName)
{
    contactName = contactName ?? "Unknown contact";
    accountName = accountName ?? "Unknown account";

    var id = Guid.NewGuid();
    var selectQuery = new Select(UserConnection)
        .Column(Column.Parameter(contactName))
        .Column("Id")
```

```
.From("Account")
.Where("Name").IsEqual(Column.Parameter(accountName)) as
Select;
var insertSelectQuery = new InsertSelect(UserConnection)
    .Into("Contact")
    .Set("Name", "AccountId")
    .FromSelect(selectQuery);

var affectedRows = insertSelectQuery.Execute();
var result = $"Inserted new contact with name '{contactName}' +
    $" and account '{accountName}'."
    $" Affected {affectedRows} rows.";
return result;
}
```

Example 3

Add the contact with the specified name and connect it to all accounts

```
public string InsertAllAccountsContact(string contactName)
{
    contactName = contactName ?? "Unknown contact";

    var id = Guid.NewGuid();
    var insertSelectQuery = new InsertSelect(UserConnection)
        .Into("Contact")
        .Set("Name", "AccountId")
        .FromSelect(
            new Select(UserConnection)
                .Column(Column.Parameter(contactName))
                .Column("Id")
                .From("Account") as Select);

    var affectedRows = insertSelectQuery.Execute();
    var result = $"Inserted {affectedRows} new contacts with name
'{contactName}'";
    return result;
}
```

See also:

- [The InsertSelect class](#)

Modifying data

Examples of using the Update class to build queries for updating database records

You can download the package with the configuration web service implementing the cases described below using the following [link](#).

In most cases, a request for modification should contain the “Where” condition, which specifies which records exactly should be modified. Otherwise, all records will be modified.

Example 1

Receive the SQL-query text

```
public string GetSqlTextExample(string oldName, string newName)
{
    var result = "";
    var update = new Update(UserConnection, "Contact")
        .Set("Name", Column.Parameter(newName))
        .Where("Name").IsEqual(Column.Parameter(oldName));
    result = update.GetSqlText();
    return result;
}
```

Example 2

Change the contact name to another one

```
public string ChangeContactName(string oldName, string newName)
{
    var update = new Update(UserConnection, "Contact")
        .Set("Name", Column.Parameter(newName))
        .Where("Name").IsEqual(Column.Parameter(oldName));
    var cnt = update.Execute();
    return $"Contacts {oldName} changed to {newName}. {cnt} rows affected.";
}
```

Example 3

Change the user that has modified all existing contact records, to the specified user.

```
public string ChangeAllContactModifiedBy(string Name)
{
    var update = new Update(UserConnection, "Contact")
        .Set("ModifiedById",
            new Select(UserConnection).Top(1)
                .Column("Id")
                .From("Contact")
                .Where("Name").IsEqual(Column.Parameter(Name)));
    var cnt = update.Execute();
    return $"All contacts are changed by {Name} now. {cnt} rows affected.";
}
```

In this example, the *Where* condition refers to the *Select* request. The *Update* request does not contain the *Where* condition, since indiscriminate modification is the goal.

See also:

- [The Update class](#)

Deleting data

Examples of using the Delete class to erase data

You can download the package with the configuration web service implementing the cases described below using the following [link](#).

In most cases, the deletion request should contain the *Where* condition, which specifies which records exactly should be deleted. Otherwise, all records will be deleted.

The example code demonstrates available methods for passing parameters to the query. When developing a project, be aware that parameters originating from the user should never be passed to the *ColumnConst* method, since this can lead to a successful SQL injection attack.

Example 1

Receive the SQL-query text

```
public string GetSqlTextExample(string name)
{
    var result = "";
    var delete = new Delete(UserConnection)
        .From("Contact")
        .Where("Name").IsEqual(Column.Parameter(name));
    result = delete.GetSqlText();
    return result;
}
```

Example 2

Change the contact name to another one

```
public string DeleteContacts(string name)
{
    var delete = new Delete(UserConnection)
        .From("Contact")
        .Where("Name").IsEqual(Column.Parameter(name));
    var cnt = delete.Execute();
    return $"Contacts with name {name} were deleted. {cnt} rows affected";
}
```

See also:

- **The Delete class**

Working with database entities

Beginner

Easy

Medium

Advanced

Examples of using the Entity class to work with the database

You can download the package with the configuration web service implementing the cases described below using the following [link](#).

Example 1

Receiving the value of the [City] schema column with a [Name] name.

```
public string GetEntityColumnData()
{
```

```
var result = "";
// Creating a request to the City schema, adding the Name column to
the request.
var esqResult = new
EntitySchemaQuery(UserConnection.EntitySchemaManager, "City");
var colName = esqResult.AddColumn("Name");
// Running the request to the database and receiving an object with
the specified identifier. You can receive the UIid of the object from
the browser navigation bar of the open record edit page.
var entity = esqResult.GetEntity(UserConnection, new
Guid("100B6B13-E8BB-DF11-B00F-001D60E938C6"));
// Receiving the value of the object column.
result += entity.GetColumnValue(colName.Name).ToString();
return result;
}
```

Example 2

Receiving the name collection of the [City] schema column.

```
public IEnumerable<string> GetEntityColumns()
{
    // Creating the data string object of the City schema (using the
schema Id received from the database).
    var entity = new Entity(UserConnection, new Guid("5CA90B6A-93E7-
4448-BEFE-AB5166EC2CFE"));
    // Receiving the object with the specified id from the database.
    You can receive the UIid of the object from the browser navigation bar
    of the open record edit page.
    entity.FetchFromDB(new Guid("100B6B13-E8BB-DF11-B00F-
001D60E938C6"),true);
    // Receiving the name collection of the object columns.
    var result = entity.GetColumnValueNames();
    return result;
}
```

Example 3

Deleting the [Order] schema record from the database

```
public bool DeleteEntity()
{
    // Creating a request to the Order schema, adding all schema
    columns to the request.
    var esqResult = new
EntitySchemaQuery(UserConnection.EntitySchemaManager, "Order");
    esqResult.AddAllSchemaColumns();
    // Running the request to the database and receiving the object
    with the specified id. You can receive the UIid of the object from the
    browser navigation bar of the open record edit page.
    var entity = esqResult.GetEntity(UserConnection, new
Guid("e3bfa32f-3fe9-4bae-9332-16c162c51e0d"));
```

```
// Deleting the object from the database.  
entity.Delete();  
// Verification whether the object with the specified Id exists in  
the database.  
var result = entity.ExistInDB(new Guid("e3bfa32f-3fe9-4bae-9332-  
16c162c51e0d"));  
return result;  
}
```

Example 4

Order status change

```
public bool UpdateEntity()  
{  
    // Creating a request to the Order schema, adding all schema  
columns to the request.  
    var esqResult = new  
EntitySchemaQuery(UserConnection.EntitySchemaManager, "Order");  
    esqResult.AddAllSchemaColumns();  
    // Running the request to the database and receiving the object  
with the specified id. You can receive the UIId of the object from the  
browser navigation bar of the open record edit page.  
    var entity = esqResult.GetEntity(UserConnection, new  
Guid("58be5223-715d-4b16-a5c4-e3d4ec0412d9"));  
    // Creating a data string object of the OrderStatus schema.  
    var statusSchema =  
UserConnection.EntitySchemaManager.GetInstanceByName("OrderStatus");  
    var newStatus = statusSchema.CreateEntity(UserConnection);  
    // Receiving the object with specified name from the database.  
    newStatus.FetchFromDB("Name", "4. Completed");  
    // Assigns a new value to the StatusId column.  
    entity.SetValue("StatusId",  
newStatus.GetTypedColumnValue<Guid>("Id"));  
    // Saving the changed object in the database.  
    var result = entity.Save();  
    return result;  
}
```

Example 5

Adding the city with the specified name and connecting it to the specified country

```
public bool InsertEntity(string city, string country)  
{  
    city = city ?? "unknown city";  
    country = country ?? "unknown country";  
    var citySchema =  
UserConnection.EntitySchemaManager.GetInstanceByName("City");  
    var entity = citySchema.CreateEntity(UserConnection);  
    entity.FetchFromDB("Name", city);  
    // Sets default values for the object columns.
```

```
entity.SetDefColumnValues();
var contryEntity = new Entity(UserConnection, new Guid("09FCE1F8-515C-4296-95CD-8CD93F79A6CF"));
contryEntity.FetchFromDB("Name", country);
// Assigns the passed-in city name to the Name column.
entity.SetColumnValue("Name", city);
// Assings the UID of the passed-in country to the CountryId UID
column.
entity.SetColumnValue("CountryId",
contryEntity.GetTypedColumnValue<Guid>("Id"));
var result = entity.Save();
return result;
}
```

See also:

- [The Entity class](#)

Work with localizable resources

Contents

- [Enabling multi-language in an object schema](#)
- [Reading multilingual data with EntitySchemaQuery](#)
- [Working with the localized data via Entity](#)
- [Localizing views](#)

Enabling multi-language in an object schema

Beginner

Easy

Medium

Advanced

Introduction

There is often a need to localize one or more columns of an object schema. In other words, certain record data must display in Creatio in multiple languages. The values should display according to the user culture. In Creatio, there is a **multilingual data mechanism** for these purposes.

To create an object schema with localizable columns:

1. Create a new or a replacing object schema.
2. Add the localizable columns, if necessary. Select the [Localizable text] checkbox in the column properties.

You can only localize text columns.

Case description

Create the [Localizable object] object schema with the localizable [Name] column.

Source code

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

Case implementation algorithm

1. Creating an object schema

Create an object schema with the following properties (Fig. 1):

- [Title] – “Localizable object”

- [Name] – “UsrEntityToLocalize”
- [Parent object] – “Base object”

Learn more about creating an object schema in the “[Creating the entity schema](#)” article.

Fig. 1. The [Localizable object] schema properties

The screenshot shows the Creatio Object Designer interface. At the top, there are buttons for Save, Add, Delete, Up, and Down. Below this is a toolbar with a 'Structure' tab selected. A dropdown menu shows 'UsrEntityToLocalize'. Under the structure tree, there is a node 'UsrEntityToLocalize' with children 'Columns' and 'Indexes'. On the right side, there is a 'Properties' panel with tabs for General, Inheritance, and Access rights. The General tab shows the following properties:

Name	UsrEntityToLocalize
Title	Localizable object
Package	Custom

The Inheritance tab shows:

Parent object	Base object (Base)
Replace parent	<input type="checkbox"/>

The Access rights tab shows:

Operations	<input type="checkbox"/>
Records	<input type="checkbox"/>

2. Adding the necessary localized columns

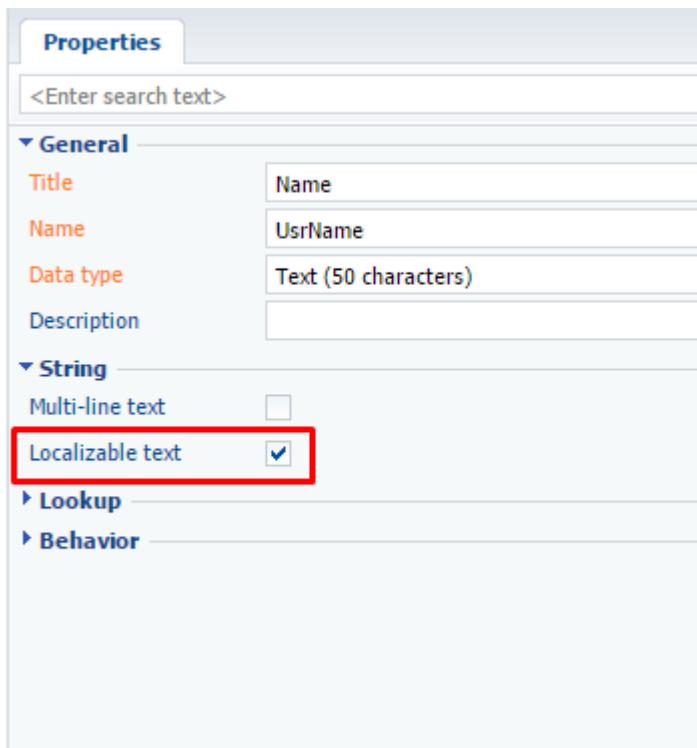
Add a column with the following properties:

- [Title] - “Name”
- [Name] – “UsrName”
- [Data type] – “Text (50 characters)”

Learn more about adding an object schema column in the “[Creating the entity schema](#)” article.

Select [Localizable text] in the added column properties (Fig. 2). The checkbox is only available in the advanced mode of the object designer (see “[Workspace of the Object Designer](#)”).

Fig. 2. The [Name] column properties



Publish the schema to apply the changes.

A **SysUsrEntityToLocalizeLcz localization table** will be created for the *UsrEntityToLocalize* object schema in the database after publishing. The localizable column data will be stored there.

See also

- **Working with the localized data via Entity**
- **Reading multilingual data with EntitySchemaQuery**

Reading multilingual data with EntitySchemaQuery

Beginner

Easy

Medium

Advanced

Introduction

Application has supported multilingual data since version 7.8.3. That means the list data is displayed based on the preferred user language ("culture"). Please refer to the "**Working with data structure (on-line documentation)**" article for more information on data localization.

Reading multilingual data with EntitySchemaQuery

EntitySchemaQuery (ESQ) ('The use of EntitySchemaQuery for creation of queries in database' in the on-line documentation) is a base mechanism for reading the Creatio database data. ESQ supports multilingual data by default.

The multilingual data sampling is performed according to the following rules:

- Users with the primary culture (English) receive the main table data.
- Users with additional culture receive the localization table data. If the localization table contains no data for the user's culture, the main table data is returned.

Example of a localized column query generation

A query generation sample code for the localized [Name] column of the [City] object schema on the server side (C#):

```
// User Connection.  
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];
```

```
// Forming a query.
var esqResult = new EntitySchemaQuery(userConnection.EntitySchemaManager, "City");
// Adding columns to a query.
esqResult.AddColumn("Name");
// Executing a database query and retrieving the entire resulting object collection.
var entities = esqResult.GetEntityCollection(userConnection);
// Retrieving the query text.
var s = esqResult.GetSelectQuery(userConnection).GetSqlText();
// Returning the result.
return s;
```

This code can be added to the custom **configuration service** ('[Creating a user configuration service](#)' in the [on-line documentation](#)) method, for example.

If a default culture is selected in the user profile, the following SQL query will be generated:

```
SELECT
    [City].[Name]  [Name]
FROM
    [dbo].[City]  [City] WITH(NOLOCK)
```

If any culture other than the primary culture is selected in the user profile, the generated SQL query will take into account the localized values for the selected culture.

```
SELECT
    ISNULL([SysCityLcz].[Name], [City].[Name]) [Name]
FROM
    [dbo].[City]  [City] WITH(NOLOCK)
    LEFT OUTER JOIN [dbo].[SysCityLcz]  [SysCityLcz] WITH(NOLOCK) ON
    ([SysCityLcz].[RecordId] = [City].[Id]
    AND [SysCityLcz].[SysCultureId] = @P1)
```

The @P1 parameter takes the record identifier value (*Id*) of the selected culture from the *SysCulture* table.

Disabling the data localization mechanism

To disable the data localization selection mechanism (even if the query is executed on behalf of a user with one of the additional cultures), you must set the ESQ instance to *false* for the *UseLocalization* property.

```
// User Connection.
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];
// Forming a query.
var esqResult = new EntitySchemaQuery(userConnection.EntitySchemaManager, "City");
// Adding a column to a query.
esqResult.AddColumn("Name");
// Disabling the data localization mechanism.
esqResult.UseLocalization = false;
// Executing a database query and retrieving the entire resulting object collection.
var entities = esqResult.GetEntityCollection(userConnection);
// Retrieving the query text.
var s = esqResult.GetSelectQuery(userConnection).GetSqlText();
// Returning the result.
return s;
```

Regardless of which culture is selected in the user's profile, the following SQL query will be generated:

```
SELECT
    [City].[Name]  [Name]
FROM
    [dbo].[City]  [City] WITH(NOLOCK)
```

Custom culture data selection

ESQ enables you to select culture data different from the current user culture and the default culture. To select the

custom culture data, call the *SetLocalizationCultureId(Guid cultureId)* method in the *ESQ* instance before data retrieval, and pass the *id* of the culture with the necessary data to it.

```
// User Connection.  
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];  
  
// Retrieving the id of the necessary culture (e.g. italian).  
var sysCulture = new SysCulture(userConnection);  
if (!sysCulture.FetchPrimaryInfoFromDB("Name", "it-IT"))  
{  
    // Error: The record is not found.  
    return "The culture is not found";  
}  
Guid italianCultureId = sysCulture.Id;  
  
// Forming a query.  
var esqResult = new EntitySchemaQuery(userConnection.EntitySchemaManager, "City");  
// Adding a column to a query.  
esqResult.AddColumn("Name");  
// Installing the necessary localization.  
esqResult.SetLocalizationCultureId(italianCultureId);  
// Executing a database query and retrieving the entire resulting object collection.  
var entities = esqResult.GetEntityCollection(userConnection);  
// Retrieving the query text.  
var s = esqResult.GetSelectQuery(userConnection).GetSqlText();  
// Returning the result.  
return s;
```

As the result, the following SQL inquiry is generated:

```
SELECT  
    ISNULL([SysCityLcz].[Name], [City].[Name]) [Name]  
FROM  
    [dbo].[City] [City] WITH(NOLOCK)  
    LEFT OUTER JOIN [dbo].[SysCityLcz] [SysCityLcz] WITH(NOLOCK) ON  
    ([SysCityLcz].[RecordId] = [City].[Id]  
    AND [SysCityLcz].[SysCultureId] = @P1)
```

The @P1 parameter takes the record identifier value (id) stored in the *italianCultureId* variable.

Working with the localized data via Entity

Beginner

Easy

Medium

Advanced

Introduction

Starting with version 7.9.1, an ability of getting the multilingual data was added to the *Entity.FetchFromDB()* method. The data fetching algorithm is similar to the *EntitySchemaQuery* algorithm (see “**Reading multilingual data with EntitySchemaQuery**” article):

1. The object will receive the data from the main table if current user culture (language) is the primary culture for the application.
2. The object will receive the data from the **localization table** if current user culture (language) is different from the primary culture. If the localization table contains no data for the user’s culture, the main table data is returned.

The examples of using the *Entity.FetchFromDB()* and *Entity.Save()* method overloads and the analysis of their execution for the user with the main (English) and additional (German) cultures (languages) are given below. These methods can be used in the user service methods (see the “**Creating a user configuration service (on-line documentation)**” article).

Reading the data

The example of source code for getting the data from the *Name* localized column of the *AccountType* schema object on the server side (C#);

```
// A user connection.
var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];
// Getting the [Account Type] schema.
EntitySchema schema =
    userConnection.EntitySchemaManager.FindInstanceByName("AccountType");
// Creating an instance of the Entity (object).
Entity entity = schema.CreateEntity(userConnection);
// A collection of column names for the fetch.
List<string> columnNamesToFetch = new List<string> {
    "Name",
    "Description"
};
//Get the data for an object with the "Customer" value in the [Name] column.
entity.FetchFromDB("Name", "Customer", columnNamesToFetch);
// Forming and sending a response.
var name = String.Format("Name: {0}", entity.GetTypedColumnValue<string>("Name"));
return name;
```

If a user who has a default language selected in the profile executes the method containing this code, the following query will be sent to the database:

```
exec sp_executesql N'
SELECT
    [AccountType].[Name] [Name],
    [AccountType].[Description] [Description]
FROM
    [dbo].[AccountType] [AccountType] WITH (NOLOCK)
WHERE
    [AccountType].[Name] = @P1',N'@P1 nvarchar(6)',@P1=N'Customer'
```

In the above query, the "Customer" value is specified in the @P1 parameter, it determines the corresponding record of the database table.

You can view the request using the [SQL Server Profiler](#) (Fig. 1).

Fig. 1. Profiling a query into a database via SQL Server Profiler

EventClass	TextData	ApplicationName	NTUserName
RPC:Completed	exec sp_executesql N' SELECT [Acco...'	.Net SqlClie...	
Audit Logout		.Net SqlClie...	
Audit Logout		.Net SqlClie...	


```
exec sp_executesql N'
SELECT
    [AccountType].[Name] [Name],
    [AccountType].[Description] [Description]
FROM
    [dbo].[AccountType] [AccountType] WITH(NOLOCK)
WHERE
    [AccountType].[Name] = @P1',N'@P1 nvarchar(6)',@P1=N'Customer'
```

Ready. Rows: 8

If a user with an additional language (such as German) selected in the profile executes the method, the following query will be sent to the database:

```
exec sp_executesql N'
SELECT
    ISNULL([SysAccountTypeLcz].[Name], [AccountType].[Name]) [Name],
```

```

ISNULL([SysAccountTypeLcz].[Description], [AccountType].[Description])
[Description]
FROM
    [dbo].[AccountType] [AccountType] WITH(NOLOCK)
    LEFT OUTER JOIN [dbo].[SysAccountTypeLcz] [SysAccountTypeLcz] WITH(NOLOCK) ON
    ([SysAccountTypeLcz].[RecordId] = [AccountType].[Id]
    AND [SysAccountTypeLcz].[SysCultureId] = @P2)
WHERE
    [AccountType].[Name] = @P1 ,N'@P1 nvarchar(6),@P2
uniqueidentifier',@P1=N'Customer',@P2='A5420246-0A8E-E111-84A3-00155D054C03'

```

In the above query, the "Customer" value is specified in the @P1 parameter, it determines the corresponding record of the main database table. The indicator of additional culture from the *SysCulture* table will be in the @P2 parameter. It will define the corresponding record from the *SysAccountTypeLcz* localization table.

Thus, for the user with English culture the *name* variable will have the "Customer" value and for the user with German culture it will be the "Kunde" value.

Saving the localized data

The *Entity.SetValue()* method is used for adding and modifying the localized data. This method can accept arguments of *string* and *LocalizableString* types.

Saving the localized data using string argument

The following saving algorithm is used in passing the *string* argument to the *Entity.SetValue()* method:

- when the user with an additional culture creates a new record, the data is added to both the main table and the localization table (for the corresponding culture);
- when the user with an additional culture modifies the existing *Entity* instance, the result is saved only in the localization table (for the corresponding culture);
- when the user with the main culture creates or modifies the *Entity* object, the data will be added or modified in the main table of the object.

The code example of saving the data using string argument:

```

var userConnection = (UserConnection)HttpContext.Current.Session["UserConnection"];
EntitySchema schema =
userConnection.EntitySchemaManager.FindInstanceByName("AccountType");
Entity entity = schema.CreateEntity(userConnection);
// Set the default values for the columns.
entity.SetDefColumnValues();
// Setting the value for the [Name] column.
entity.SetValue("Name", "New customer");
// Saving.
entity.Save();
var name = String.Format("Name: {0}", entity.GetTypedColumnValue<string>("Name"));
return name;

```

When the user with the default (English) culture executes this code, the following query will be executed in the database:

```

exec sp_executesql N'
INSERT INTO [dbo].[AccountType]([Id], [Name], [CreatedOn], [CreatedBy], [ModifiedOn], [ModifiedBy], [ProcessListeners], [Description])
VALUES(@P1, @P2, @P3, @P4, @P5, @P6, @P7, @P8)',N'@P1 uniqueidentifier,@P2
nvarchar(12),@P3 datetime2(7),@P4 uniqueidentifier,@P5 datetime2(7),@P6
uniqueidentifier,@P7 int,@P8 nvarchar(4000)',@P1='3A820BC8-006D-42B7-A472-
E331FBD73E20',@P2=N'New Customer',@P3='2017-02-10 09:40:23.0909251',@P4='410006E1-
CA4E-4502-A9EC-E54D922D2C00',@P5='2017-02-10 09:40:23.0929256',@P6='410006E1-CA4E-
4502-A9EC-E54D922D2C00',@P7=0,@P8=N'

```

In the above query, the "New customer" value is specified in the @P2 parameter, it is saved in the main database table.

If the user has an additional culture (German) set in their profile, the following code must be executed to save the data with the string argument:

```
var userConnection = (UserConnection) HttpContext.Current.Session["UserConnection"];
EntitySchema schema =
userConnection.EntitySchemaManager.FindInstanceByName("AccountType");
Entity entity = schema.CreateEntity(userConnection);
entity.SetDefColumnValues();
entity.SetColumnValue("Name", "Neue Kunden");
entity.Save();
var name = String.Format("Name: {0}", entity.GetTypedColumnValue<string>("Name"));
return name;
```

The query to the *AccountType* main table will be the same as to the main localization, but the “Neue Kunden” value will be specified in the @P2 parameter.

```
exec sp_executesql N'
INSERT INTO [dbo].[AccountType] ([Id], [Name], [CreatedOn], [CreatedBy], [ModifiedOn], [ModifiedBy], [ProcessListeners], [Description])
    VALUES (@P1, @P2, @P3, @P4, @P5, @P6, @P7, @P8)',N'@P1 uniqueidentifier,@P2 nvarchar(12),@P3 datetime2(7),@P4 uniqueidentifier,@P5 datetime2(7),@P6 uniqueidentifier,@P7 int,@P8 nvarchar(4000)',@P1='94052A88-499D-4072-A28A-6771815446FD',@P2=N'Neue Kunden',@P3='2017-02-10 10:07:00.3454424',@P4='410006E1-CA4E-4502-A9EC-E54D922D2C00',@P5='2017-02-10 10:07:00.3454424',@P6='410006E1-CA4E-4502-A9EC-E54D922D2C00',@P7=0,@P8=N''
```

In addition, the query will be executed in the localization table:

```
exec sp_executesql N'
INSERT INTO [dbo].[SysAccountTypeLcz] ([Id], [ModifiedOn], [RecordId], [SysCultureId], [Name])
    VALUES (@P1, @P2, @P3, @P4, @P5)',N'@P1 uniqueidentifier,@P2 datetime2(7),@P3 uniqueidentifier,@P4 uniqueidentifier,@P5 nvarchar(12)',@P1='911A721A-0E5A-4CC3-B6D9-9E5FE85FEC64',@P2='2017-02-10 10:07:00.3664442',@P3='94052A88-499D-4072-A28A-6771815446FD',@P4='A5420246-0A8E-E111-84A3-00155D054C03',@P5=N'Neue Kunden'
```

The “Neue Kunden” value will be specified in the @P5 parameter in the above request.

The value that does not correspond to the default culture will be added to the *AccountType* table.

To avoid this save the localized data using the localized strings.

Saving the localized data using localized string argument

The code example of saving the data using the localized string:

```
var userConnection = (UserConnection) HttpContext.Current.Session["UserConnection"];
EntitySchema schema =
userConnection.EntitySchemaManager.FindInstanceByName("AccountType");
Entity entity = schema.CreateEntity(userConnection);
entity.SetDefColumnValues();

// Creating a localized string with localized values for different cultures.
var localizableString = new LocalizableString();
localizableString.SetCultureValue(new CultureInfo("en-US"), "New customer en-US");
localizableString.SetCultureValue(new CultureInfo("de-DE"), "Neue Kunden de-DE");

// Setting the value of the column using the localized string.
entity.SetColumnValue("Name", localizableString);
entity.Save();

// The result will be displayed in the current user culture.
var name = String.Format("Name: {0}", entity.GetTypedColumnValue<string>("Name"));
```

```
return name;
```

Regardless of the language in the user's profile, the following queries will be sent to the database upon the code execution:

1. The query with the “New customer en-US” value in the @P2 argument will be sent to the main page:

```
exec sp_executesql N'
INSERT INTO [dbo].[AccountType] ([Id], [Name], [CreatedOn], [CreatedBy], [ModifiedOn], [ModifiedBy], [ProcessListeners], [Description])
    VALUES (@P1, @P2, @P3, @P4, @P5, @P6, @P7, @P8)',N'@P1 uniqueidentifier,@P2 nvarchar(18),@P3 datetime2(7),@P4 uniqueidentifier,@P5 datetime2(7),@P6 uniqueidentifier,@P7 int,@P8 nvarchar(4000)',@P1='5AC81E4A-FCB2-4019-AE5B-0C485A5F65BD',@P2=N'New Customer en-US',@P3='2017-02-10 10:47:21.7471581',@P4='410006E1-CA4E-4502-A9EC-E54D922D2C00',@P5='2017-02-10 10:47:21.7511578',@P6='410006E1-CA4E-4502-A9EC-E54D922D2C00',@P7=0,@P8=N''
```

2. The query with the “Neue kunden de-DE” value in the @P5 argument will be sent to the localization page:

```
exec sp_executesql N'
INSERT INTO [dbo].[SysAccountTypeLcz] ([Id], [ModifiedOn], [RecordId], [SysCultureId], [Name])
    VALUES (@P1, @P2, @P3, @P4, @P5)',N'@P1 uniqueidentifier,@P2 datetime2(7),@P3 uniqueidentifier,@P4 uniqueidentifier,@P5 nvarchar(18)',@P1='6EC9C205-7F8B-455E-BC68-3D9AA6D7B7C0',@P2='2017-02-10 10:47:21.9272674',@P3='5AC81E4A-FCB2-4019-AE5B-0C485A5F65BD',@P4='A5420246-0A8E-E111-84A3-00155D054C03',@P5=N'Neue Kunden de-DE'
```

If the code is executed by a user who has an additional culture set in the profile and the value for the default culture is not specified in the localization string, the record for the user's culture will be added to the primary *AccountType* table.

Localizing views

Beginner

Easy

Medium

Advanced

Introduction

Views are often used to select data. The views, in their turn, can select data from localizable columns. Thus, to perform data selection via views, you need to set up localizable views.

To localize a view:

1. Create a view object schema. Enable multi-language for localizable columns.
2. Add a new localization view in the database.

Case description

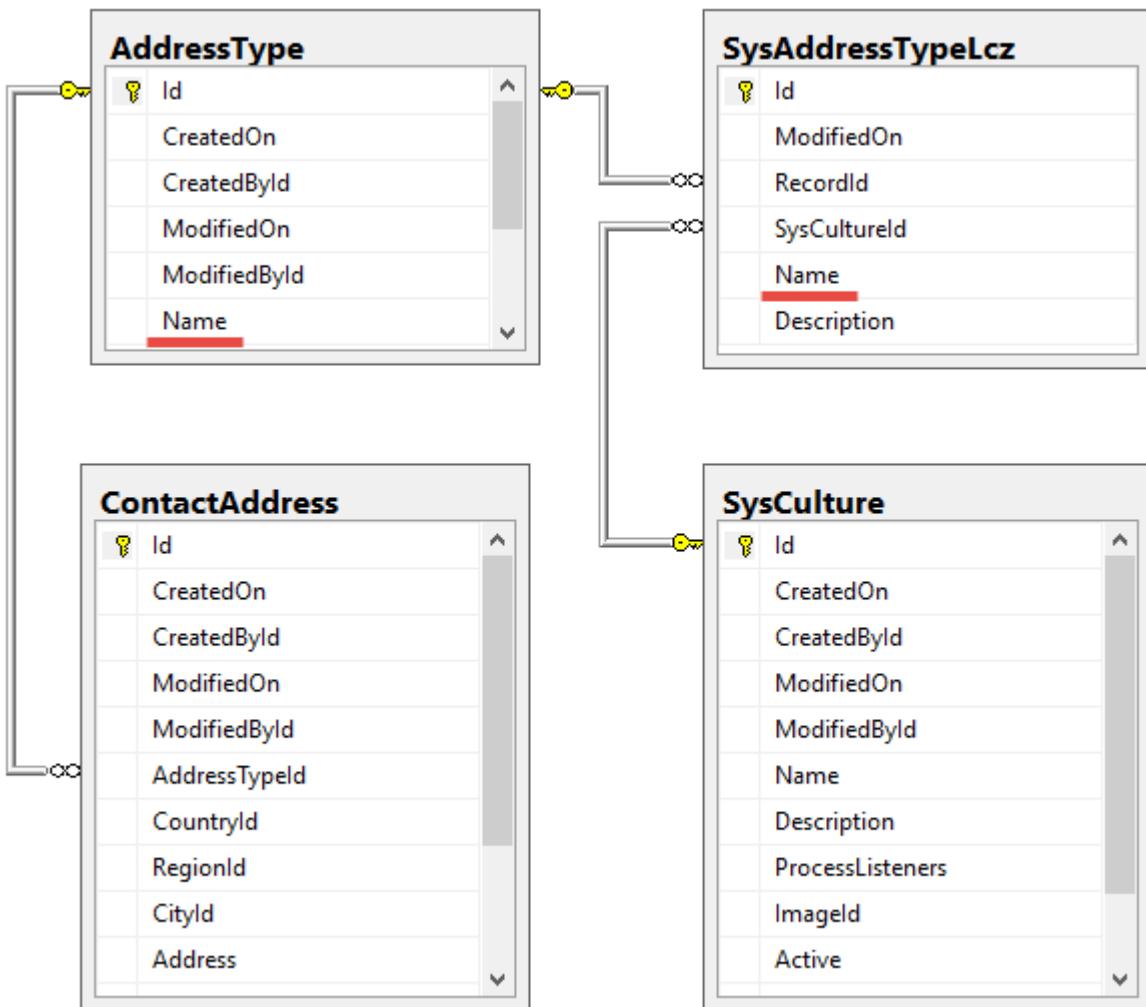
The [Contact address] object schema (*ContactAddress*) is already implemented in Creatio. One of its columns refers to the [Address type] (*AddressType*) lookup (schema). The [Name] (*Name*) column of the [Address type] schema is localizable.

Since almost every object schema has a corresponding table in Creatio, the table connection structure will look as follows (fig. 1):

- *ContactAddress* – the table of contact addresses. It is bound to the *AddressType* table by the *AddressTypeId* column.
- *AddressType* – the table of address types. The *Name* column contains values that correspond to the user culture set by default. The values that correspond to other cultures are contained in the *SysAddressTypeLcz* table.
- *SysAddressTypeLcz* – automatically generated system table of localizable values of address types. It is bound to the *AddressType* table by the *RecordId* column, and to the *SysCulture* table – by the *SysCultureId* column. The *Name* column contains the localizable values of address types for the culture specified in the *SysCultureId* column.

- *SysCulture* – system table with the list of cultures.

Fig. 1. The structure and table connections for the [Contact address], [Address type] schemas and localization tables.



Create a view that selects the following fields:

- *ContactAddress.Address* – contact address
- *AddressType.Name* – localizable name of the address type

Source code

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

Case implementation

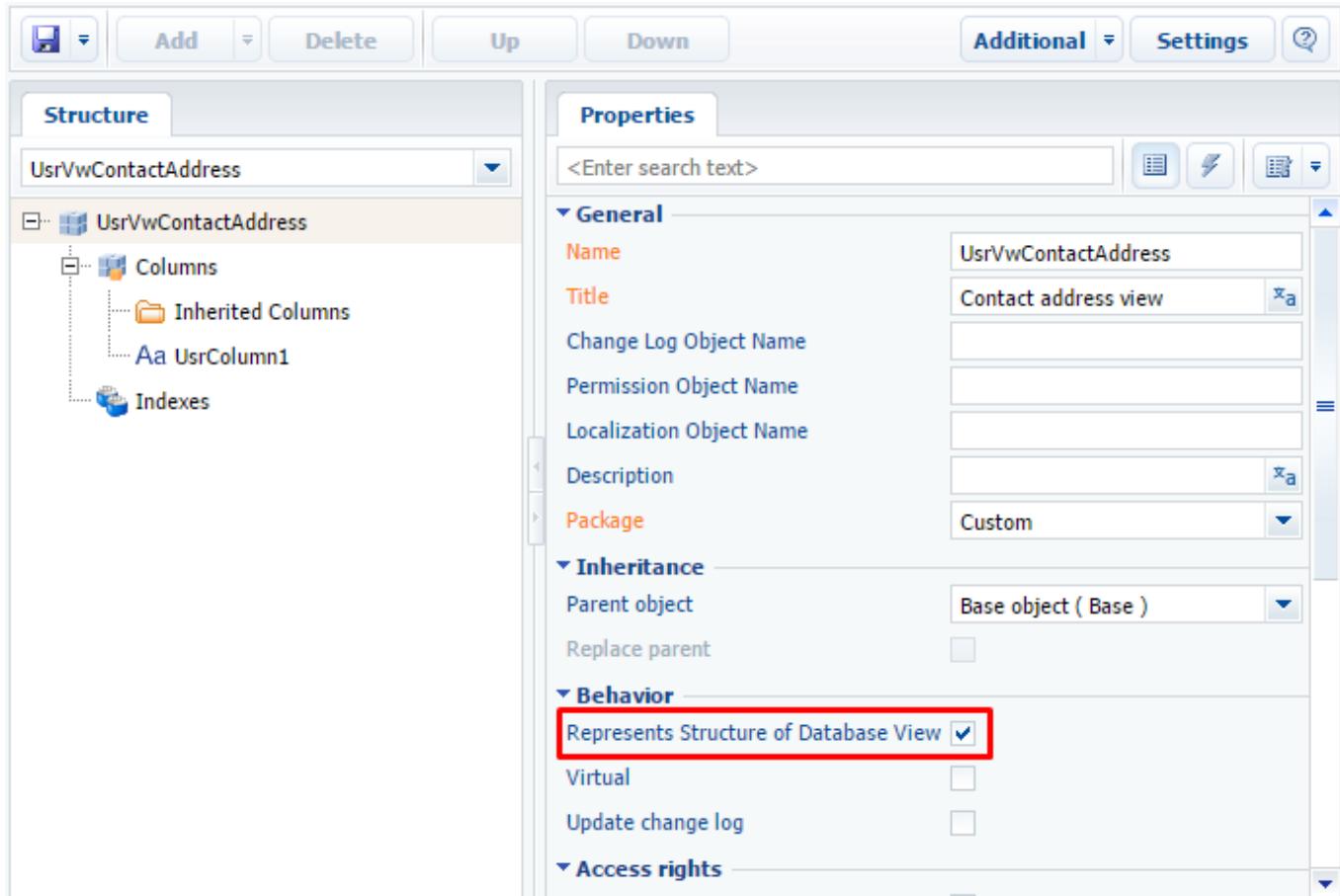
1. Creating a view object schema

Create an object schema with the following parameter values:

- [Name] – “UsrVwContactAddress”. The *Usr* prefix is set via the [Prefix for object name] system setting. The *Vw* (contracted from *View*) indicates that the schema is a view in the database.
- [Title] – “Contact address view”.
- [Parent object] – “Base object”.
- [Represent Structure of Database View] – this checkbox must be selected (fig. 2).

Creating an object schema and adding custom columns to it is described in the “[Creating the entity schema](#)” article.

Fig. 2. The “Represent Structure of Database View” checkbox



Add two string columns to the created schema

The first column will contain non-localizable address values of the culture by default. Set the following parameter values for it:

- [Name] – “UsrAddress”
- [Title] - “Address”
- [Data type] – "Text (50 characters)"

The second column will contain the localizable values of the address type. Set the following parameter values for it:

- [Name] – “UsrAddressType”
- [Title] – “Address type”
- [Data type] – "Text (50 characters)"
- [Localizable text] – the checkbox must be selected (fig. 3). Read more about multi-language functionality in the **“Enabling multi-language in an object schema”** article.

Fig. 3. Multi-language checkbox in the column

The screenshot shows the Creatio Object Designer interface. On the left, the 'Structure' tab is selected, displaying a tree view of schema objects. A node 'UsrVwContactAddress' is expanded, showing its 'Columns' section. Under 'Columns', there is a folder 'Inherited Columns' and two items: 'UsrAddressType' and 'UsrAddress'. The 'UsrAddressType' item is highlighted with a light orange background. On the right, the 'Properties' tab is active, showing various configuration options for the selected column. The 'General' section includes fields for 'Title' (set to 'Address type'), 'Name' (set to 'UsrAddressType'), 'Data type' (set to 'Text (50 characters)'), and 'Description'. The 'String' section includes 'Multi-line text' and 'Localizable text' checkboxes, with 'Localizable text' checked. The 'Lookup' section includes 'Lookup', 'Cascade connection', 'Do not control integrity', and 'List' checkboxes. The 'Behavior' section includes 'Required' (set to 'No') and 'Default value' (set to '(No)').

The [Data type] and [Localizable text] properties are available in the advanced mode of displaying the column properties (see “**Workspace of the Object Designer**”).

Save and publish the schema.

2. Creating the view in the database

To create the *UsrVwContactAddress* view in the database, execute the following SQL script:

```
-- The view name should correspond to the name of the schema table.
CREATE VIEW dbo.UsrVwContactAddress
AS
SELECT
    ContactAddress.Id,
    -- The view columns should correspond to the schema columns.
    ContactAddress.Address AS UsrAddress,
    AddressType.Name AS UsrAddressType
FROM ContactAddress
INNER JOIN AddressType ON ContactAddress.AddressTypeId = AddressType.Id;
```

To create the *UsrVwContactAddress* localizable view in the database, execute the following SQL script:

```
-- The view name should correspond to the schema localization table.
CREATE VIEW dbo.SysUsrVwContactAddressLcz
AS
SELECT
    SysAddressTypeLcz.Id,
    ContactAddress.id AS RecordId,
    SysAddressTypeLcz.SysCultureId,
    -- The view columns should correspond to the schema columns.
    SysAddressTypeLcz.Name AS UsrAddressType
FROM ContactAddress
```

```
INNER JOIN AddressType ON ContactAddress.AddressTypeId = AddressType.Id
INNER JOIN SysAddressTypeLcz ON AddressType.Id = SysAddressTypeLcz.RecordId;
```

The columns of the *UsrVwContactAddress* localizable view must correspond to the localization table columns. Learn more about the localization tables in the “[Localization tables](#)” article.

As a result of case implementation, when the data is read via *EntitySchemaQuery*, the correct values for different languages are displayed in the *UsrAddressType* column of the *UsrVwContactAddress* view.

Case result demonstration

To verify the result, use one of the examples from the “[Reading multilingual data with EntitySchemaQuery](#)” article. To verify the query results, you can create a custom configuration service (see “[Creating a user configuration service \(on-line documentation\)](#)”).

In the created service class, implement the method that will return the list of addresses and their types from the created *UsrVwContactAddress* non-localizable view via the *EntitySchemaQuery* query.

```
[OperationContract]
[WebInvoke(Method = "GET", UriTemplate = "Ex01")]
public string Ex01()
{
    // User connection.
    var userConnection =
(UserConnection)HttpContext.Current.Session["UserConnection"];
    // Query generation.
    var esqResult = new EntitySchemaQuery(userConnection.EntitySchemaManager,
"UsrVwContactAddress");
    // Adding columns to query.
    esqResult.AddColumn("UsrAddress");
    esqResult.AddColumn("UsrAddressType");
    // Executing the database query and receiving the whole resulting collection of
objects.
    var entities = esqResult.GetEntityCollection(userConnection);
    // Display of results.
    var s = "";
    foreach (var item in entities)
    {
        s += item.GetTypedColumnValue<string>("UsrAddress") + Environment.NewLine;
        s += item.GetTypedColumnValue<string>("UsrAddressType") +
Environment.NewLine;
    }
    return s;
}
```

The result of method operation is represented in fig. 4

Fig. 4. The result of verification in localization by default

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">
  123 6th St. Melbourne Delivery 71 Pilgrim Avenue Chevy Chase Work 514 S. Magnolia St. Delivery
</string>
```

To verify the operation of a localized view in the created class, implement the second method:

```
[OperationContract]
[WebInvoke(Method = "GET", UriTemplate = "Ex01")]
public string Ex01()
{
    var userConnection =
(UserConnection)HttpContext.Current.Session["UserConnection"];
    // Receiving the Id of the necessary culture, for example, English.
```

```
var sysCulture = new SysCulture(userConnection);
if (!sysCulture.FetchPrimaryInfoFromDB("Name", "en-US"))
{
    return "Culture didn't find.";
}
Guid CultureId = sysCulture.Id;
var esqResult = new EntitySchemaQuery(userConnection.EntitySchemaManager,
"UsrVwContactAddress");
esqResult.AddColumn("UsrAddress");
esqResult.AddColumn("UsrAddressType");
// Setting up the needed localization.
esqResult.SetLocalizationCultureId(CultureId);
var entities = esqResult.GetEntityCollection(userConnection);
var s = "";
foreach (var item in entities)
{
    s += item.GetTypedColumnValue<string>("UsrAddress") + Environment.NewLine;
    s += item.GetTypedColumnValue<string>("UsrAddressType") +
Environment.NewLine;
}
return s;
}
```

The result of this method operation is represented in fig. 5

Fig. 5. The result of verification in selected localization

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">
  123 6th St. Melbourne Dirección de entrega 71 Pilgrim Avenue Chevy Chase Dirección de trabajo 514 S. Magnolia St. Dirección de entrega
</string>
```

Client-side development

Contents

- **CRUD-operation implementation on client**
- **WebSocket messages transferring mechanism. ClientMessageBridge**
- **Frequently used client-side classes**

Application logical levels

Contents

- **Introduction**
- **Configuration architectural elements**
- **Modules**
- **Module message exchange. Sandbox component**
- **Client view model schemas**
- **Repositories. Types and recommendations on use**

Application logical levels

Beginner

Easy

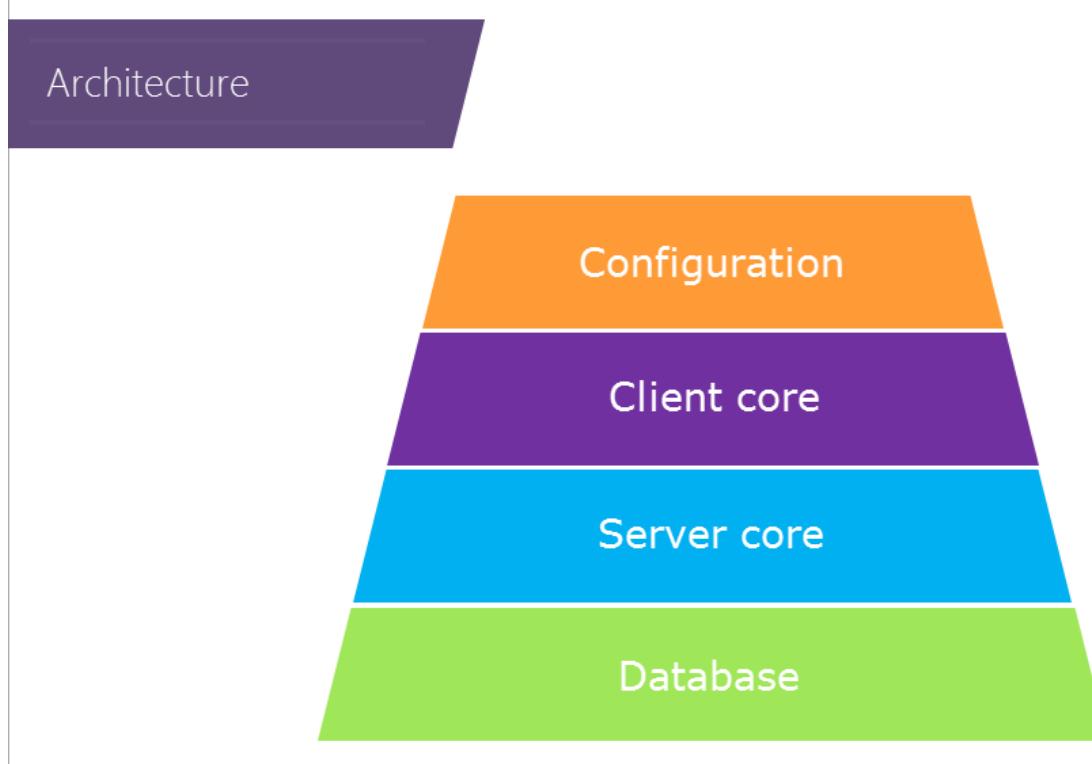
Medium

Advanced

Overview

The Creatio architecture is comprised of the following components (Fig. 1):

Fig. 1. Components



1. Database

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

Database primary functions:

- data storage
- data management
- configuration settings storage.

Database objects:

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

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

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

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

2. Server core

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

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

Server core primary components:

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

3. Client core

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

Client core primary components:

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

4. Configuration

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

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

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

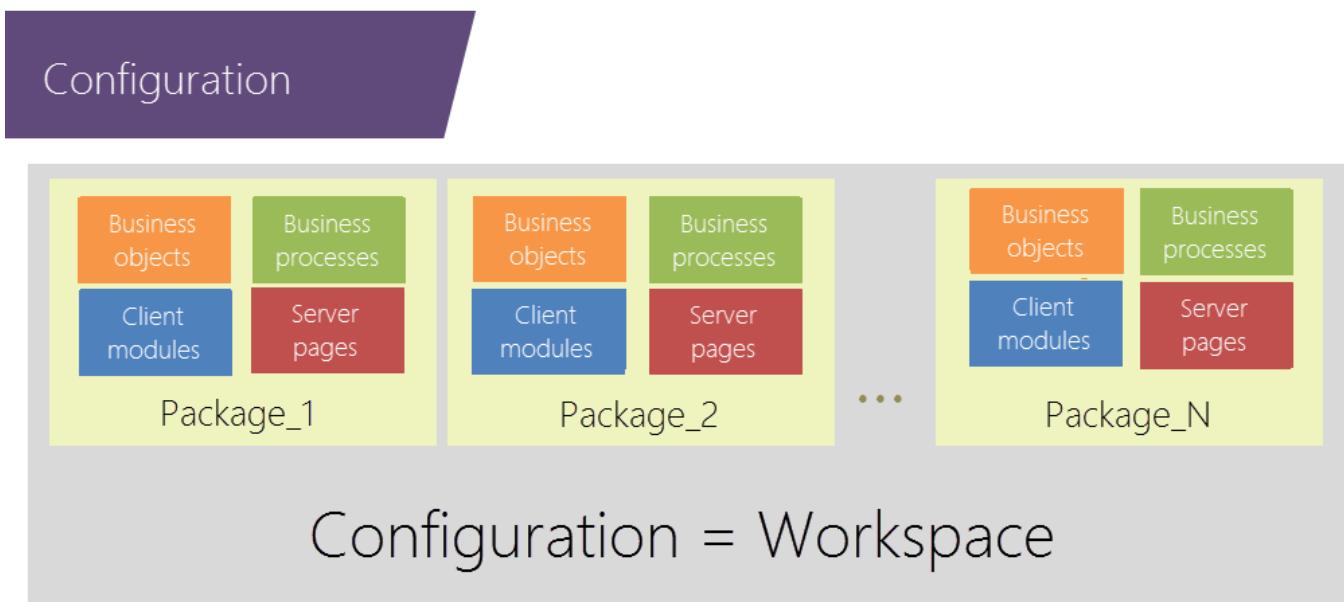
- Objects – entities that store data and connect a database table to a class on the server side.
- Business processes – customizable elements that are visual algorithms of user activities.
- Client modules.

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

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

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

Fig. 2. Creatio configuration



Configuration architectural elements

Beginner

Easy

Medium

Advanced

Overview

Packages

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

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

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

There are two types of Creatio packages:

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

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

Package replacement mechanism

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

execute the logic of the replacing element instead of its parent base element.

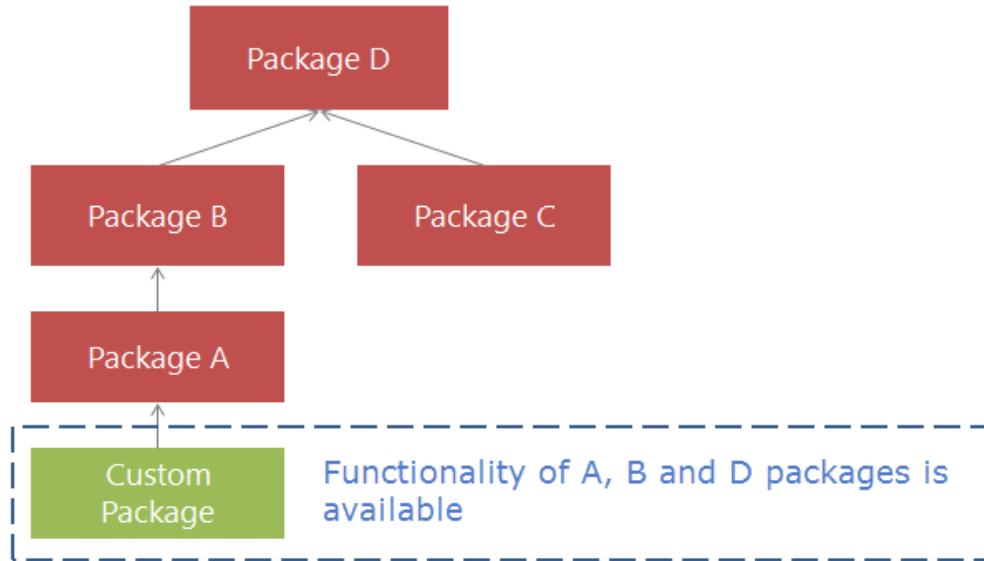
The replacement of a single base element can be implemented in several custom packages. The final implementation of a replacing element in a compiled configuration is determined by the hierarchy of all packages that contain replacing elements for the base package.

Package hierarchy

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

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

Fig. 1. Package dependencies



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

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

Package composition:

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

For more information on working with packages, please refer to the "[Development tools](#)" articles.

Schema

A Creatio configuration is a set of objects, processes, pages and modules.

The base element of a configuration is a schema. Configuration elements are schemas of different types. From the

programming point of view, a schema is a core class inherited from the base Schema class.

Schema types:

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

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

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

Required properties of schemas:

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

Schema primary methods:

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

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

Separate schema managers are used for different schema types.

Properties and methods of different classes are documented in the [library of classes](#).

Object

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

There are base objects and custom objects.

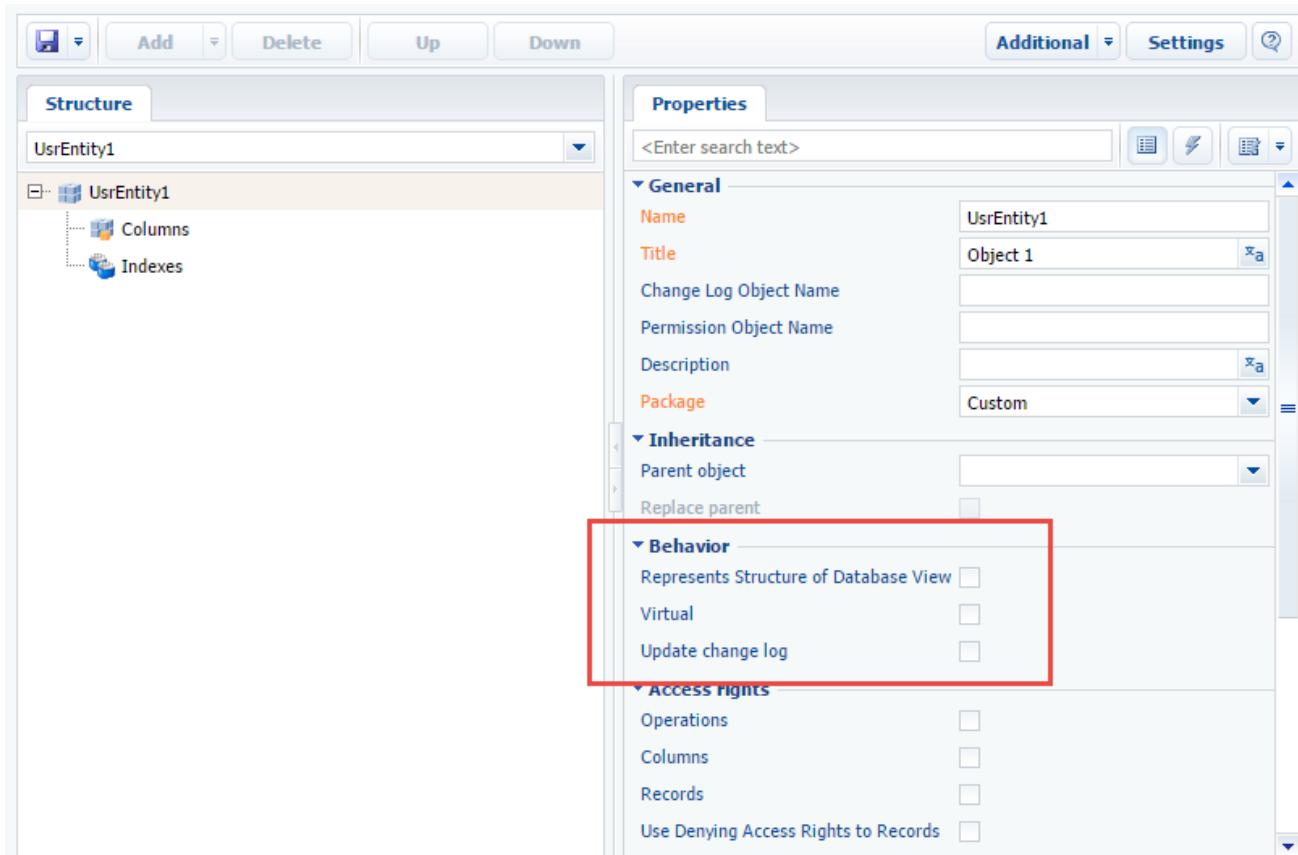
- Base objects are non-editable and are stored in the base packages. They can be replaced in custom packages.
- Custom objects are objects created as part of configurations saved in custom packages.

There are 3 types of objects in Creatio:

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

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

Fig. 2. Object types



A system object has three primary components:

1. Object schema – database table structure and properties. Object schema includes table columns (names and data types), indexes, access rights to object schema. Schema of an object is an instance of the *EntitySchema* class.
2. Object data – a data row of a table and methods for its processing. Each data row is an instance of the *Entity* class.
3. Embedded object process. Event model is implemented for each system object. Handling of object events is implemented through an embedded object process.

Module

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

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

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

The RequireJS loader provides the mechanism for declaring and loading modules, based on the AMD concept.

General operational principles of the RequireJS loader mechanism:

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

Each client schema in Creatio 7.x is characterized by at least one **client module**.

Client core provides mechanisms for working with modules:

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

Client module types

The following client module types are used in Creatio:

1) Non-visual module

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

2) View schema (visual module)

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

3) Extension module (replacing client module)

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

Modules

Contents

- **AMD concept. Module definition**
- **Modular development principles in Creatio**
- **Module types and their specificities**

AMD concept. Module definition

Beginner Easy Medium Advanced

Introduction

Starting from version 7.0, the client part of the Creatio application has a modular structure: it is designed as a set of functional blocks, implemented in separate modules. While the application is running, the modules and their dependencies are loaded in accordance with the [Asynchronous Module Definition \(AMD\)](#) approach.

The AMD approach declares the mechanism for defining and asynchronous loading of modules and their dependencies, which allows you to load only the data needed to work with the system at the moment. The AMD concept is supported by various JavaScript frameworks. In Creatio, the [RequireJS](#) loader is used to work with modules.

Modules

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

In JavaScript, modules are created in accordance with the "[Module](#)" programming pattern. A classic implementation of this pattern is using anonymous functions that return specific values (object, function, etc.) associated with the module. The module value is exported to the global object. Example:

```
// Immediately-invoked functional expression (IIFE). Anonymous function,
// which initializes the "myGlobalModule" property of the global object with a
function,
// that returns module value. Thus, the module actually loads,
// which can later be accessed through the "myGlobalModule" global property.
(function () {
    // Access to a module on which the current module depends.
    // This module already should be loaded to the
    // "SomeModuleDependency" global variable at the moment of access.
    // "this" context in this case is a global object.
    var moduleDependency = this.SomeModuleDependency;
    // The declaration in the property of the global function object that returns the
    module value.
    this.myGlobalModule = function () { return someModuleValue; };
}());
```

When interpreter finds a functional expression like this, it immediately resolves it. As a result, a function that will return the module value will be placed in the *myGlobalModule* property of the global object.

The main disadvantages of this approach are the complexity of declaration and use of the dependency modules for the modules of such type. In particular, the disadvantages are:

1. All module dependencies must already be loaded at the moment of anonymous function execution.
2. The dependency modules are loaded via the `<script></script>` HTML element at the page header. Global variable names are then used to access the modules. At the same time, the developer must clearly understand and implement the order in which all dependency modules are loaded.
3. As a result, the modules are loaded before the page is rendered, therefore the modules cannot access the page controls to implement custom logic.

This means that the modules cannot be loaded dynamically; no additional logic can be applied at page loading, etc. In large projects like Creatio, the complexity of managing a large number of modules with many dependencies that can overlap each other is a problem.

The “RequireJS” loader

[RequireJS](#) is an AMD-based module declaring and loading mechanism that helps avoid the disadvantages of working with large numbers of modules. Basic principles of the RequireJS loader operation:

1. Modules are declared in a special `define()` function, which registers a factory function to instantiate a module. At the same time, it does not load the module immediately when function is called.
2. The module dependencies are passed as a string array and not through the properties of the global object.
3. The loader executes the loading of all dependency modules passed as arguments to `define()`. The modules are loaded asynchronously, and the loader determines their loading order arbitrarily.
4. After the loader completes loading of all specified module dependencies, it will call the factory function that will return the module value. The downloaded dependency modules will be passed to the factory function as arguments.

Module declaration. The “`define()`” function

For the loader to work with an asynchronous module, this module must be declared in the source code by the `define()` function in the following way:

```
define(
    ModuleName,
    [dependencies],
    function (dependencies) {
```

```

    }
);

```

The parameters of the `define()` function are listed in Table 1.

Table 1. - The parameters of the `define()` function

Argument	Value
ModuleName	Module name string. Optional parameter. If the parameter is not specified, the loader will assign the module name, based on its location in the application script tree. However, to access the module from other parts of the application (including the cases when this module must be asynchronously loaded as a dependency of another module), the parameter must be specified.
dependencies	An array of module names that this module depends on. Optional parameter. RequireJS loads all dependencies passed in the array. Note that the order of dependencies in the <code>dependencies</code> array enumeration must correspond to the order of parameters in the enumeration passed to the factory function. The factory function will be called only after all dependencies listed in the <code>dependencies</code> parameter have been loaded. The loading of dependency modules is asynchronous.
function(dependencies)	Anonymous factory function that instantiates the module. Required parameter. The objects that are associated by the loader with the dependency modules listed in the <code>dependencies</code> argument are passed to the function as arguments. Access to the properties and methods of the dependency modules within the created module is carried out through these arguments. The order of modules in the <code>dependencies</code> enumeration must correspond to the order of the factory function arguments. The factory function will be called only after all dependency modules of the current module (listed in the <code>dependencies</code> parameter) have been loaded. The factory must return a value that the loader will associate as the exported value of created module. The return value can be: <ul style="list-style-type: none"> • An <i>object, which is the module for the system</i>. After this module is initially download by the client, it is saved in the browser cache. If the module declaration has been modified after it was downloaded to the client (for example, during the configuration logic implementation), then the cache needs to be cleared and the module must be loaded again. An example of module declaration that returns the declared module as an object is provided below. • <i>The module constructor function</i>. The context object in which the module will be created is passed as an argument to the constructor. Loading this module will result in creating of the module instance (instantiated module) on the client. Reloading of this module to the client with the <code>require()</code> function will create another instance of the module. These two instances of the same module will be treated by the system as two different independent modules. An example of declaring an instantiated module is the <code>CardModule</code> module from the <code>NUI</code> package.

An example of using the `define()` function to declare a `SumModule`, which adds two numbers.

```

// The "SumModule" module has no dependencies.
// So, an empty array is passed as the second argument, and
// parameters are not passed to the anonymous factory function.

define("SumModule", [], function () {
    // The body of the anonymous function contains internal functionality
    // implementation of the module.
    var calculate = function (a, b) { return a + b; };
    // The value returned by the function is an object, which is the module for the

```

```
system.  
    return {  
        // Object description. In this case, the module is an object with the "summ"  
        property.  
        // The value of this property is a function with two arguments, returning  
        the sum of these arguments.  
  
        summ: calculate  
    };  
});
```

The factory function returns the object as the module value, which the module will be for the system.

Modular development principles in Creatio

Beginner Easy Medium **Advanced**

Types of Creatio modules

All client functions in Creatio can be broken down into the following groups:

- Base libraries
- Core
- Sandbox
- Client modules

Base libraries

Base libraries are third-party JavaScript libraries used in the application. The [RequireJS](#) library is used as the module loader. The [ExtJS](#) framework functions are used in the configuration logic for working with interface controls. [JQuery](#), [Backbone](#) and other frameworks are used as well. All third-party JavaScript libraries are placed in the *Terrasoft.WebApp|Resources|ui* folder of the application.

Core

The main purpose of the Creatio client core is to provide a unified interface for interaction of all other client parts of the system. The core provides API for accessing base client libraries, defines the sandbox contents for modules, provides access to system enumerations and constants, implements client mechanism for working with data, etc. The core does not work directly with the system modules. It is only aware of the primary application module (*ViewModule*), which loads all remaining modules.

To access the core functions used in the custom client logic, a module must import the *terrasoft* module as a dependency.

Sandbox

A module is an isolated programming unit. It is not aware of other system modules except for the names of the modules from which it depends. A special object called the *sandbox* is designed for interaction between the modules.

The sandbox provides the two key mechanisms for interaction between the modules in the system:

1. A mechanism for message exchange between the modules. Modules can communicate with each other only through messages. If module needs to inform other modules that its status has changed, it publishes a message using the *sandbox.publish()* method of the sandbox. If a module needs to receive messages about status changes of other modules, it must subscribe to those messages. The subscription is done through calling the *sandbox.subscribe()* sandbox method.
2. Loading modules “on-demand” into the specified area of the application (for visual modules). In the process of implementing custom business logic, you may need to load dynamically the modules that have not been declared as dependencies. These modules can be loaded in the process of the module declaration, in the *define()* function. The *sandbox.load()* sandbox method is used for this.

To enable interaction with other modules, a module must import the *sandbox* module as dependency.

Client modules

Client modules are separate functional blocks that are loaded and run on-demand, according to the AMD technology. All custom functions are implemented in client modules. Despite several functional differences, all Creatio client modules have similar structure that matches the module description format in AMD. For more information about client modules, please see the “**Module types and their specificities**” article.

The “ext-base”, “terrasoft” and “sandbox” modules

Creatio contains modules that are used in most client modules of a configuration. These are the *ext-base* module of the ExtJs framework functions, the *terrasoft* module of the *Terrasoft* objects and name spaces, and the *sandbox* module that implements the mechanism for message exchange between modules. These modules can be accessed in the following way:

```
// Module definition and getting dependency module links.  
define("ExampleModule", ["ext-base", "terrasoft", "sandbox"],  
    // Ext – link to the object that grants access to the ExtJs features.  
    // Terrasoft – link to the object that grants access to the system variables,  
    core variables, etc.  
    // sandbox – used for message exchange between modules.  
    function (Ext, Terrasoft, sandbox) {  
});
```

Specifying base modules in the `["ext-base", "terrasoft", "sandbox"]` dependencies is not required. After creating module's class object, the *Ext*, *Terrasoft* and *sandbox* modules will be available as object properties: *this.Ext*, *this.Terrasoft*, *this.sandbox*.

Declaring module class The *Ext.define()* method

One of the more important ExtJs javascript framework functions in Creatio is class declaration. The *define()* method of the global *Ext* object is used for this. An example of declaring a class with this method:

```
// Terrasoft.configuration.ExampleClass – class name with  
// name space compliance.  
Ext.define("Terrasoft.configuration.ExampleClass", {  
    // Shortened class name.  
    alternateClassName: "Terrasoft.ExampleClass",  
    // Name of the class from which inheritance is made.  
    extend: "Ext.String",  
    // Block for declaring static properties and methods.  
    static: {  
        // Example of a static property.  
        myStaticProperty: true,  
  
        // Example of a static method.  
        getMyStaticProperty: function () {  
            // Example of access to a static property.  
            return Terrasoft.ExampleClass.myStaticProperty;  
        }  
    },  
    // Example of a dynamic property.  
    myProperty: 12,  
    // Example of a class dynamic method.  
    getMyProperty: function () {  
        return this.myProperty;  
    }  
});
```

Examples of various options for creating class instances:

```
// Creating a class instance by full name.  
var exampleObject = Ext.create("Terrasoft.configuration.ExampleClass");  
// Creating a class instance by a shortened name (alias).
```

```

var exampleObject = Ext.create("Terrasoft.ExampleClass");
// Creating a class instance with the specified properties.
var exampleObject = Ext.create("Terrasoft.ExampleClass", {
    // Overriding object property from 12 to 20.
    myProperty: 20,
    // Defining a new method for the current class instance.
    myNewMethod: function () {
        return this.getMyProperty() * 2;
    }
});

```

Inheriting a module class

In a simple implementation of a module, its contents is either a simple object with a set of methods and properties, or a constructor function that the module must return to a function that is called after its loading.

```

define("ModuleExample", [], function () {
    // Example of a module that returns a simple object.
    return {
        init: function () {
            // The method will be called on module initialization,
            // but the module contents will not get into the DOM.
        }
    }
});
define("ModuleExample", [], function () {
    // Example of a module that returns a constructor function.
    return function () {
        this.init = function () {
            // The method will be called on module initialization,
            // but the module contents will not get into the DOM.
        }
    }
});

```

Such simple module cannot add its view to the [Document Object Model](#) (DOM), unless you explicitly implement the *render()* method, which would return a view instance and insert it to the DOM. The logic for calling the *render()* method in a module object is covered on the application core level. The *destroy()* method is not implemented in such module as well. If the module is visual, i.e., it contains the *render()* method, then it will be impossible to unload the view from the DOM, unless the unloading logic is implemented in the *destroy()* method.

In most cases, the module class should be inherited from *Terrasoft.configuration.BaseModule* or *Terrasoft.configuration.BaseSchemaModule*, where the following methods are already implemented:

- *Init()* – a method for module initialization. Initializes the properties of class object and subscribes to messages.
- *render()* – a method for rendering the module view in the DOM. Returns a view. Accepts a single *renderTo* argument, which is the element where the module object view will be inserted.
- *Destroy()* – a method that deletes a module view, view model, unsubscribes from messages and destroys the module class object.

Below is an example of a simple module class inherited from "Terrasoft.BaseModule". This module adds a button to the DOM. Clicking the button will display a text message and then the button is deleted from the DOM.

```

define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        // Short class name.
        alternateClassName: "Terrasoft.ModuleExample",
        // The class from which the inheritance is done.
        extend: "Terrasoft.BaseModule",
        // Required property. If it is not defined, an error will be generated on the
        // "Terrasoft.core.BaseObject" level, since the class is inherited from
        "Terrasoft.BaseModule".
    });
}
);

```

```
Ext: null,
// Required property. If it is not defined, an error will be generated on the
// "Terrasoft.core.BaseObject" level, since the class is inherited from
"Terrasoft.BaseModule".
sandbox: null,
// Required property. If it is not defined, an error will be generated on the
// "Terrasoft.core.BaseObject" level, since the class is inherited from
"Terrasoft.BaseModule".
Terrasoft: null,
// View model.
viewModel: null,
// View. A button is used as an example.
view: null,
// If the init() method is not implemented in this class,
// then, when an instance of the current class is created,
// the init() method of the parent class Terrasoft.BaseModule will be called.
init: function () {
    // Executes the logic of the init() method of the parent class.
    this.callParent(arguments);
    this.initViewModel();
},
// Initializes a view model.
initViewModel: function () {
    // Saving module class context
    // for accessing it from the view model.
    var self = this;
    // Creating a view model.
    this.viewModel = Ext.create("Terrasoft.BaseViewModel", {
        values: {
            // Button caption.
            captionBtn: "Click Me"
        },
        methods: {
            // Button click handler.
            onClickBtn: function () {
                var captionBtn = this.get("captionBtn");
                alert(captionBtn + " button was pressed");
                // Calls a method for unloading the view and view model,
                // which results in deleting the button from the DOM.
                self.destroy();
            }
        }
    });
},
// Creates a view (button),
// binds it to the view model and inserts in the DOM.
render: function (renderTo) {
    // A button is created as a view.
    this.view = this.Ext.create("Terrasoft.Button", {
        // Container where the button will be placed.
        renderTo: renderTo,
        // The id HTML attribute.
        id: "example-btn",
        // Class name.
        className: "Terrasoft.Button",
        // Button caption.
        caption: {
            // Binds the button caption
            // with the captionBtn property of the view model.
            bindTo: "captionBtn"
        },
        // Handler method for the button click event.
    });
}
```

```

        click: {
            // Binds the button click handler
            // to the onClickBtn() method of the view model.
            bindTo: "onClickBtn"
        },
        // Button style. Available styles are defined in the enumeration.
        // Terrasoft.controls.ButtonEnums.style.
        style: this.Terrasoft.controls.ButtonEnums.style.GREEN
    );
    // Binds the view and the view model.
    this.view.bind(this.viewModel);
    // Gets the view that will be inserted in the DOM.
    return this.view;
},
// Deletes unused objects.
destroy: function () {
    // Destroys the view, which results in deleting the button from the DOM.
    this.view.destroy();
    // Deletes the unused view model.
    this.viewModel.destroy();
}
);
// Gets module object.
return Terrasoft.ModuleExample;
});

```

Adding the button using the ViewModel schema is described in the ["How to add a button to a section"](#), ["How to add a button to an edit page in the new record add mode"](#) and ["How to add the button on the edit page in the combined mode"](#) articles.

Synchronous and asynchronous module initialization

There are two ways to initialize a module class instance: synchronously and asynchronously.

Synchronous initialization

A module is initialized synchronously if the `isAsync: true` property (of the configuration object that is passed as a parameter of the `loadModule()` method) is not specified at its loading. For example, if the following is executed:

```
this.sandbox.loadModule([moduleName])
```

...Then the module class methods will be loaded synchronously. The `init()` method will be called first, then the `render()` method will be immediately executed.

Below is an example of a synchronously initialized module.

```

define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        alternateClassName: "Terrasoft.ModuleExample",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
        init: function () {
            // This is executed first upon module initialization.
        },
        render: function (renderTo) {
            // This is executed on the module initialization, right after the init
method.
        }
    });
});

```

Asynchronous initialization

A module is initialized asynchronously if the `isAsync: true` property (of the configuration object that is passed as a parameter of the `loadModule()` method) is specified at its loading. For example, if the following is executed:

```
this.sandbox.loadModule([moduleName], { isAsync: true })
```

In this case, a single parameter will be passed to the `init()` method: a callback function with the current module context. When calling the callback function, the `render()` method of the loaded module is called. The view will be added to the DOM only after the `render()` method is executed.

Below is an example of an asynchronously initialized module.

```
define("ModuleExample", [], function () {
    Ext.define("Terrasoft.configuration.ModuleExample", {
        alternateClassName: "Terrasoft.ModuleExample",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
        // This is executed first upon module initialization.
        init: function (callback) {
            setTimeout(callback, 2000);
        },
        render: function (renderTo) {
            // The method is executed after a 2 second delay.
            // The delay is specified in the setTimeout() function argument, in the
            init() method.
        }
    });
});
```

Chain of modules

Sometimes a model must be shown in the view of other model. For example, the `SelectData` page for selecting a lookup value must be displayed to set a value in a certain field on the current page. In this case, the current page module must not be unloaded, and the lookup selection page view must be displayed in its container. To implement this, use module chains.

To start building a chain, add the `keepAlive` property in the configuration object of the loaded module. For example, a lookup selection module `selectDataModule` must be called from the current page module `CardModule`. To do this, the following code must be executed:

```
sandbox.loadModule("selectDataModule", {
    // Id of the loaded module view.
    id: "selectDataModule_id",
    // The view that will be added to the current page container.
    renderTo: "cardModuleContainer",
    // Specifies that the current module must not be unloaded.
    keepAlive: true
});
```

After the code is executed, a module chain will be created, consisting of the current page module and the lookup selection page module. Clicking the [Add new record] button from the current `selectData` page module will open a new page and add another module to the chain. This way you can add any number of module instances to a chain. Active module (the one that is currently displayed on the page) is always the last element in the chain. If an intermediate element in the chain is set as active, then all elements that are located after it will be destroyed. Use the `loadModule` function to activate a chain element and pass the module Id as its parameter:

```
sandbox.loadModule("someModule", {
    id: "someModuleId"
});
```

The core will destroy all chain elements after the specified one and will execute standard module loading logic (call

the `init()` and `render()` methods). The `render()` method will be passed to the container where the previous active module was placed. All modules in the chain can work (receive and send messages, save data, etc.) as before.

If the `keepAlive` is not added to the configuration object (or added with the `keepAlive: false` value) on loading of the `loadModule()` method, then the module chain will be destroyed.

Module types and their specificities

Beginner

Easy

Medium

Advanced

Introduction

Client Modules are separate functional blocks, downloaded and run on demand in accordance with the AMD technology. System functions are implemented via client modules. All client modules in Creatio share description structures that correspond with AMD module description format.

Client module types

The following client module types are available in Creatio:

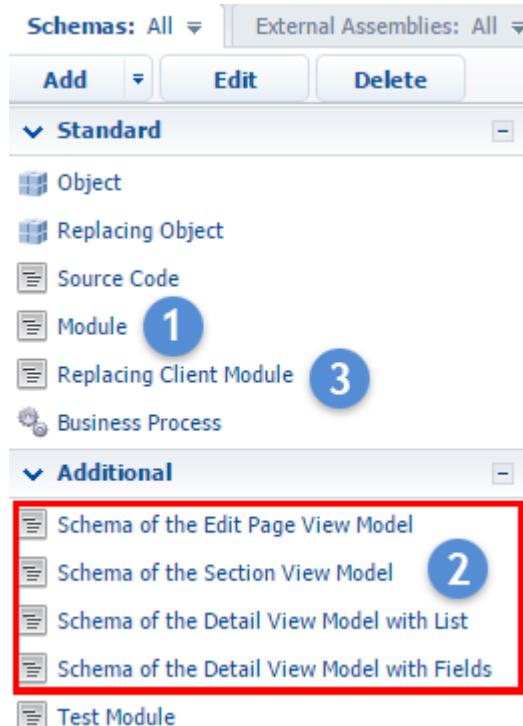
- non-visual modules (module schema)
- visual modules (view model schema)
- expanding modules and replacing client modules

Non-visual modules (module schema)

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

Go to the [Configuration] section, click [Add] and select [Module] to create a non-visual module (Fig. 1, 1).

(Fig. 1, 1). Creating non-visual modules



Visual module (view model schema);

Visual modules are used to implement *ViewModel* presentation models in the system, according to the [MVVM](#)

pattern. Visual modules encapsulate both the data used in the GUI controls and methods for working with that data. Examples of visual modules are the section, detail and page modules.

Go to the [Configuration] section, click [Add] and select [Schema of the View Model] (Fig. 1, 2) to create a visual module. (Fig. 1, 2).

Replacing client modules

Use replacing client modules if you need to modify or expand the functionality of base modules.

Go to the [Configuration] section, click [Add] and select [Replacing client module] (Fig. 1, 3) to create a replacing client module.

Client module features

The “init()” and “render()” methods

A default Creatio client module can contain two methods:

- The *init()* method implements the logic that is executed when the module is loaded. This method is called first by the client core if it's been detected upon module loading. The *init()* method usually implements subscriptions to events of other modules and initializes the module values.
- The *render(renderTo)* method implements the module visualization logic. The client core will automatically call this method (if it is available) upon module loading. Before data visualization, the mechanism for binding the view (*View*) and the view model (*ViewModel*) must be triggered for correct data processing. As a rule, this mechanism is initiated in the *render()* method: the *bind()* method is called in the view object. If the module is loaded into a container, a reference to this container will be passed to the *render()* method as an argument. The visual modules must implement the *render()* method.

Case

Create a module with the *init()* and *render()* methods. Both methods must display a message. The client kernel will first call the *init()* method and then the *render()* method when the module is loaded. A message must alert you each time a method is called.

You can test any visual module, perform client downloads and generate visualization in the base version of Creatio. To do this, generate the following address string:

[Application URL]/[Configuration Number]/NUI/ViewModule.aspx#[Module name]

Example: <http://myserver.com/CreatioWebApp/o/Nui/ViewModule.aspx#CustomModule>

The *CustomModule* module will be returned to the client, and its visual representation will be displayed in the central area of the application.

Case implementation:

1. Create a client module schema: Go to the [Configuration] section, click [Add] and select [Module] (Fig. 1, 1) to create a non-visual module.
2. Set the [Title] property to “Standard module example” and the [Name] property to "ExampleStandartModule". Select the name of the schema package in the [Package] property.
3. Add the following code to the [Source code] tab:

```
// Declaring the "ExampleStandartModule" module. The module does not have any dependencies,
// so an empty array is passed as the second parameter.
define("ExampleStandartModule", [],
    // The factory function returns a module object with two methods.
    function () {
        return {
            // The method will be called first by the core upon loading to the client.
            init: function () {
```

```
        alert("Calling the init() method of the \"ExampleStandartModule\"  
module.");  
    },  
    The method will be called by the kernel when the module is loaded into  
the container. The link to the container is passed to the method  
    // as the renderTo parameter. A message will display a page control id  
element,  
    // which has to display the visual data of the module. centerPanel by  
default.  
    render: function (renderTo) {  
        alert("Calling the render() method of the \"ExampleStandartModule\"  
module. The module is uploaded to the container " + renderTo.id);  
    }  
};  
});
```

4. Save and publish the schema.

You can run the example by executing the following query: *[Application URL]/[Workplace number]/NUI/ViewModule.aspx#ExampleStandartModule*

Calling a function of a module from another module. Utility modules

Although a module is essentially an isolated software unit, the functions of other modules can be used in its logic. The module with the intended functionality needs to be imported as a dependency for that to occur. Access to the dependency module instance is granted through the factory function argument.

You can group auxiliary and service methods into separate *utility modules* and import them into modules that require this functionality.

Case

Create a standard module with the *init()* and *render()* methods. The method for displaying a message window must be taken out to a separate utility module.

Case implementation:

1. Create a schema for the client module with the following properties:

- Assign “Utility module example” to the [Title] property.
- Assign "ExampleUtilsModule" to the [Name] property.

Select the name of the schema package in the [Package] property.

2. Add the following code to the [Source code] tab:

```
// Declaring a utility module. The module does not have any dependencies and only  
contains one method  
// for displaying a message.  
define("ExampleUtilsModule", [],  
    function () {  
        return {  
            // The method for displaying a message. The message displayed in the  
window  
            // is passed to the method as the "information" argument.  
            showInformation: function (information) {  
                alert(information);  
            }  
        };  
    });
```

3. Save and publish the utility module schema.

4. Create a client schema with the following properties:

- [Title]: "Utility module use example".
- [Name]: "UseExampleUtilsStandartModule".

5. Add the following code to the [Source code] tab:

```
// The ExampleUtilsModule dependency module is imported to the module for access to
// the utility method.
// The factory function argument - a link to a loaded utility module.
define("UseExampleUtilsStandartModule", ["ExampleUtilsModule"],
    function (ExampleUtilsModule) {
        return {
            // The utility method for displaying a message window is called in the
            init() and render() functions
            // with a message which is passed to the utility method as an argument.
            init: function () {
                ExampleUtilsModule.showInformation ("Calling the init() method of the
UseExampleResourceStandartModule module.");
            },
            render: function (renderTo) {
                ExampleUtilsModule.showInformation("Calling the render() method of
the UseExampleUtilsStandartModule module. The module is uploaded to the container " +
renderTo.id);
            }
        };
    });
});
```

6. Save and publish the schema.

You can run the example by executing the following query: *[Application URL]/[Workplace number]/NUI/ViewModule.aspx#UseExampleUtilsStandartModule*

Working with resources

Localized strings and images are the resources of the client schema that are most often used in the implementation logic of the module.

Add resources to the client schema in the [Structure] tab of the client schema designer. The application core automatically generates a special [Client module name]Resources module, which contains resources of the client module. The *localizableStrings* property stores schema's localized strings. The *images* property stores image resources.

In order to access a resource module from a client module, you need to import the resource module as a dependency into the client module.



We recommend using localized resources rather than string literals or constants in the module code.

Case

Similar to previous cases, create an *ExampleResourceModule* module with the *init()* and *render()* methods. Use the *ExampleUtilsModule* method of the utility module to display the message window. Contents displayed in message windows must be specified by the values of localized strings in a client schema, rather than string literals.

Case implementation:

1. Create a client module with the following properties:

- [Title]: "Resource module use example".
- [Name]: "ExampleResourceModule".

Select the name of the schema package in the [Package] property.

2. In the created schema, add two localizable strings that will be displayed in the messages. To add a localizable

string in the [Structure] tab, right-click the [LocalizableStrings] element and select [Add].

Assign the following properties for the message string of the *init()* method:

- [Name]: “InitMessage”.
- [Value]: “Calling the init() method of the UseExampleResourceStandartModule module”.

Assign the following properties for the message string of the *render()* method:

- [Name]: “RenderMessage”.
- [Value]: “Calling the render() method of the UseExampleResourceStandartModule module”.

3. Add the following code to the [Source code] tab:

Two dependency modules are loaded into the module: the “ExampleUtilsModule” utility module, created earlier, and the

```
// ExampleResourceModuleResources resource module. The resource module is not
// explicitly created - it is generated by the core on the basis of
// resources added to the client schema.
define("ExampleResourceModule",
    ["ExampleUtilsModule", "ExampleResourceModuleResources"],
    // Now, the messages in init() and render() are not specified by
    // constant values, but localized strings.
    function (utils, resources) {
        return {
            init: function () {
                utils.showInformation(
                    // Access to the localized InitMessage line, in which the message
                    for the init() method is stored.
                    resources.localizableStrings.InitMessage);
            },
            render: function () {
                utils.showInformation(
                    // Access to the localized RenderMessage line, in which the
                    message for the render() method is stored.
                    resources.localizableStrings.RenderMessage);
            }
        );
    }
);
```

4. Save and publish the schema.

Using replacing client modules

The extension modules of the basic functionality do not support inheritance in its traditional sense. You must completely transfer (or copy) the program code of the original module when creating extension modules, and then make your changes in the extension module. Although you do not need to transfer the code of the original module while creating replacing client modules, you still can not use its resources. All resources (localized strings, images) must be duplicated in the replacement schema.

Module message exchange. Sandbox component

Contents

- **Module message exchange**
- **Bidirectional messages**
- **Loading and unloading modules**

Module message exchange

Beginner

Easy

Medium

Advanced

Introduction

A Creatio module is an isolated software unit. It has no information about other Creatio modules apart from the module name list from which it depends. See “**Modular development principles in Creatio**” for more information about Creatio modules.

Sandbox object is used for interaction between the isolated modules. One of the key *sandbox* mechanisms is module message exchange.

Modules can only communicate via messages. A module shall publish a message to communicate its status change to other Creatio modules. If the module needs to receive messages about status change in other modules, it must be subscribed to these messages.

To interact with other Creatio modules, the module must import the *sandbox* module as a dependency.

It is not necessary to specify ["ext-base", "terrasoft", "sandbox"] base modules in dependencies if the module exports class constructor. *Ext*, *Terrasoft* and *sandbox* objects will be available as object properties after creating module class object: *this.Ext*, *this.Terrasoft*, *this.sandbox*.

Message registration

You need to register messages to implement module message exchange.

Message registration is executed automatically if messages are declared in the *messages* module property.

sandbox.registerMessages(messageConfig) method is used to register module messages, where *messageConfig* is a module message configuration object.

Configuration object is a “key-value” collection, where every element is as follows:

```
"MessageName": {  
    mode: [Message operation mode],  
    direction: [Message direction]  
}
```

“MessageName” is a collection element key that contains the message name. The value is a configuration object that contains the following properties:

- *mode* – message operation mode. Must contain *Terrasoft.MessageMode* (*Terrasoft.core.enums.MessageMode*) enumeration value.
- *direction* – message direction. Must contain *Terrasoft.MessageDirectionType* (*Terrasoft.core.enums.MessageDirectionType*) enumeration value.

Message exchange modes (*mode* property):

- Broadcast – message operation mode with a predefined number of subscribers. Corresponds to *Terrasoft.MessageMode.BROADCAST* enumeration value.
- Address – message operation mode when a message can only be processed by one subscriber. Corresponds to *Terrasoft.MessageMode.PTP*. enumeration value.

There can be several subscribers in the address mode, but only one can process messages, usually it is the last registered subscriber.

Message direction (*direction* property):

- Publication (publish) – the module can only publish a message in *sandbox*. Corresponds to *Terrasoft.MessageDirectionType.PUBLISH*. enumeration value.
- Subscription (follow) – the module can only subscribe to a message, published from another module. Corresponds to *Terrasoft.MessageDirectionType.SUBSCRIBE*. enumeration value.
- Bidirectional – allows to publish and subscribe to the same message in different instances of the same class or within the same schema inheritance hierarchy. Corresponds to *Terrasoft.MessageDirectionType.BIDIRECTIONAL*. enumeration value.

Module message registration:

```
// Message configuration object collection.  
var messages = {  
    "MessageToSubscribe": {
```

```

        mode: Terrasoft.MessageMode.PTP,
        direction: Terrasoft.MessageDirectionType.SUBSCRIBE
    },
    "MessageToPublish": {
        mode: Terrasoft.MessageMode.BROADCAST,
        direction: Terrasoft.MessageDirectionType.PUBLISH
    }
};

// Message registration.
this.sandbox.registerMessages(messages);

```

It is not necessary to register messages via the `sandbox.registerMessages()` method in the view model schemas. Declare the message configuration object in `messages` property (see "**Messages. The "messages" property**").

To reject message registration in a module, use `sandbox.unRegisterMessages(messages)` method, where `messages` – is a message name or a message name array. Message registration rejection:

```

// Single message registration rejection.
this.sandbox.unRegisterMessages("MessageToSubscribe");
// Message array registration rejection.
this.sandbox.unRegisterMessages(["MessageToSubscribe", "MessageToPublish"]);

```

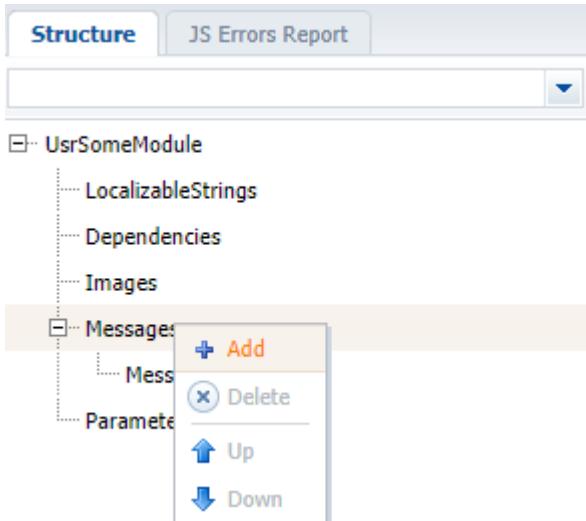
Adding messages to module schema

You can also register messages by adding them to a module schema or via a designer (see "**Module designer**").

To add messages to module schema:

1. On the [Structure] tab of the module schema designer select the [Messages] node, right-click and execute the [Add] command (Fig.1).

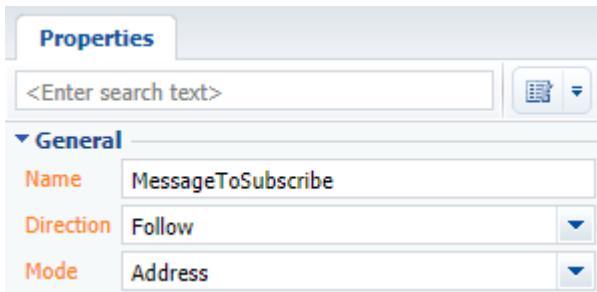
Fig. 1. Adding messages to the module schema structure



2. Set the necessary properties for the added message (Fig.2):

- [Name] – the message name that corresponds to the module configuration object key.
- [Direction] – message direction. Possible values: “Follow” (subscribe) and “Publish” (publish).
- [Mode] – message operation mode. Possible values: “Broadcast” and “Address”.

Fig. 2. Message properties



It is not necessary to add messages to schema structure in **view model schemas**.

Message publication

`sandbox.publish(messageName, messageArgs, tags)` method is used to publish messages.

Method parameters:

- *messageName* – the string that contains the message name, for instance, "MessageToSubscribe".
- *messageArgs* – the object, passed as an argument to the message handler method in the subscription module. If there are no input parameters in the handler method, assign null value to *messageArgs* parameter.
- *tags* – tag array that allows to uniquely identify the message sending module. Usually, the `[this.sandbox.id]` value is used. *Sandbox* defines the message subscribers and publishers according to the tag array.

Only the handlers that meet at least one tag will be run for the message published with the tag array. Messages, published without tags, will only be processed by subscribers without tags.

Message publication method:

```
// Message publication without tags or argument.
this.sandbox.publish("MessageWithoutArgsAndTags");
// Message publication without a handler method argument.
this.sandbox.publish("MessageWithoutArgs", null, [this.sandbox.id]);
// Message publication with a handler method argument.
this.sandbox.publish("MessageWithArgs", {arg1: 5, arg2: "arg2"}, ["moduleName"]);
// Message publication with an arbitrary tag array.
this.sandbox.publish("MessageWithCustomIds", null, ["moduleName", "otherTag"]);
```

When you publish a message in the address mode, you can receive the result of its processing by the subscriber. To do this, the message handler method in the subscription module must return the corresponding result (see "Message subscription"). Message publication:

```
// Message declaring and registration.
var messages = {
    "MessageWithResult": {
        mode: Terrasoft.MessageMode.PTP,
        direction: Terrasoft.MessageDirectionType.PUBLISH
    }
};
this.sandbox.registerMessages(messages);
// Message publication and receipt of the result of its processing by the
subscription module.
var result = this.sandbox.publish("MessageWithResult", {arg1:5, arg2:"arg2"},
["resultTag"]);
// Result display on the browser console.
console.log(result);
```

When you publish a message in the broadcast mode, you can receive the result of its processing via the object, passed as an argument to the handler method.

```
// Message declaring and registration.
var messages = {
```

```

    "MessageWithResult": {
        mode: Terrasoft.MessageMode.BROADCAST,
        direction: Terrasoft.MessageDirectionType.PUBLISH
    }
};

this.sandbox.registerMessages(messages);
var arg = {};
// Message publication and receipt of the result of its processing by the
subscription module.
// Add result property into the object of the subscription module handler method and
populate it with the processing result.
this.sandbox.publish("MessageWithResult", arg, ["resultTag"]);
// Result display on the browser console.
console.log(arg.result);

```

Message subscription

You can subscribe to a message using the `sandbox.subscribe(messageName, messageHandler, scope, tags)` method.

Method parameters:

- `messageName` – the string that contains the message name, for instance, "MessageToSubscribe".
- `messageHandler` – the handler method, run upon the message receipt. It can be either an anonymous function or a module method. A parameter, whose value must be passed upon the message publishing via the `sandbox.publish()` method can be indicated in the method definition.
- `Scope` – `messageHandler` handler method execution context.
- `tags` – tag array that allows to uniquely identify the message sending module. *Sandbox* defines the message subscribers and publishers according to the tag array.

Message subscription method:

```

// Message subscription without handler method arguments.
// Handler method is an anonymous function. Execution context is the current module.
// The getsandboxid() method must return the tag that corresponds to the published
message tag.
this.sandbox.subscribe("MessageWithoutArgs", function(){console.log("Message without
arguments")}, this, [this.getSandBoxId()]);
// Message subscription with a handler method argument.
this.sandbox.subscribe("MessageWithArgs", function(args){console.log(args)}, this,
["moduleName"]);
// Message subscription with an arbitrary tag.
// It can be any tag out of the published message tag array.
// The myMsgHandler handler method must be implemented separately.
this.sandbox.subscribe("MessageWithCustomIds", this.myMsgHandler, this,
["otherTag"]);

```

The message handler method must return the corresponding result for a message in the address mode. Message subscription:

```

// Message declaring and registration.
var messages = {
    "MessageWithResult": {
        mode: Terrasoft.MessageMode.PTP,
        direction: Terrasoft.MessageDirectionType.SUBSCRIBE
    }
};
this.sandbox.registerMessages(messages);
// Message subscription.
this.sandbox.subscribe("MessageWithResult", this.onMessageSubscribe, this,
["resultTag"]);
...
// The handler method is implemented in the subscription module.
// args – object, passed upon message publication.

```

```
onMessageSubscribe: function(args) {
    // Parameter change.
    args.arg1 = 15;
    args.arg2 = "new arg2";
    // Obligatory return of result.
    return args;
},
```

Asynchronous message exchange

Use callback function approach if the message handler method in subscription module generates the result asynchronously.

Message publication and result:

```
// Message publication without tags or argument.
this.sandbox.publish("MessageWithoutArgsAndTags");
// Message publication without a handler method argument.
this.sandbox.publish("MessageWithoutArgs", null, [this.sandbox.id]);
// Message publication with a handler method argument.
this.sandbox.publish("MessageWithArgs", {arg1: 5, arg2: "arg2"}, ["moduleName"]);
// Message publication with an arbitrary tag array.
this.sandbox.publish("MessageWithCustomIds", null, ["moduleName", "otherTag"]);
```

Message subscription:

```
// Message subscription without handler method arguments.
// Handler method is an anonymous function. Execution context is the current module.
//The getsandboxid() method must return the tag that corresponds to the published
message tag.
this.sandbox.subscribe("MessageWithoutArgs", function(){console.log("Message without
arguments")}, this, [this.getSandBoxId()]);
// Message subscription with a handler method argument.
this.sandbox.subscribe("MessageWithArgs", function(args){console.log(args)}, this,
["moduleName"]);
// Message subscription with an arbitrary tag.
// It can be any tag out of the published message tag array.
// The myMsgHandler handler method must be implemented separately.
this.sandbox.subscribe("MessageWithCustomIds", this.myMsgHandler, this,
["otherTag"]);
```

Module with a message:

Below is a module with message publication and subscription:

```
define("UsrSomeModule", [], function() {
    Ext.define("Terrasoft.configuration.UsrSomeModule", {
        alternateClassName: "Terrasoft.UsrSomeModule",
        extend: "Terrasoft.BaseModule",
        Ext: null,
        sandbox: null,
        Terrasoft: null,
        messages: {
            "MessageToSubscribe": {
                mode: Terrasoft.MessageMode.PTP,
                direction: Terrasoft.MessageDirectionType.SUBSCRIBE
            },
            "MessageToPublish": {
                mode: Terrasoft.MessageMode.BROADCAST,
                direction: Terrasoft.MessageDirectionType.PUBLISH
            }
        },
        init: function() {
```

```
        this.callParent(arguments);
        this.sandbox.registerMessages(this.messages);
        this.processMessages();
    },
    processMessages: function() {
        this.sandbox.subscribe("MessageToSubscribe", this.onMessageSubscribe,
this);
        this.sandbox.publish("MessageToPublish", null, [this.sandbox.id]);
    },
    onMessageSubscribe: function() {
        console.log("'MessageToSubscribe' received");
    },
    destroy: function() {
        if (this.messages) {
            var messages = this.Terrasoft.keys(this.messages);
            this.sandbox.unRegisterMessages(messages);
        }
        this.callParent(arguments);
    }
});
return Terrasoft.UsrSomeModule;
});
```

For more information

- **Modular development principles in Creatio**
- **Module types and their specificities**
- **Client view model schemas**
- **Messages. The "messages" property**
- **Bidirectional messages**

Bidirectional messages

Beginner Easy Medium **Advanced**

Introduction

One of the key *sandbox* mechanisms is module message exchange (see “**Module message exchange**”).

It often becomes necessary to publish and subscribe to the same message in different instances of the same class (module) or within the same schema inheritance hierarchy. To perform this the *sandbox* object has bidirectional messages that correspond to the value of the *Terrasoft.MessageDirectionType.BIDIRECTIONAL* enumeration.

Registration of bidirectional messages

To register bidirectional messages in the *messages* property of the schema, use the following confrontation object:

```
messages: {
    "MessageName": {
        mode: [Message mode],
        direction: Terrasoft.MessageDirectionType.BIDIRECTIONAL
    }
}
```

The purpose and possible values of the elements of configuration object used in message registration are described in the “**Module message exchange**” article.

Use case

The following case demonstrates how bidirectional messages work.

The *CardModuleResponse* message is registered in the *BaseEntityPage* schema, which is a base schema for all view

model schemas of the record edit pages.

```
define("BaseEntityPage", [...], function(...) {
    return {
        messages: {
            ...
            "CardModuleResponse": {
                "mode": this.Terrasoft.MessageMode.PTP,
                "direction": this.Terrasoft.MessageDirectionType.BIDIRECTIONAL
            },
            ...
        },
        ...
    };
});
```

For example, the message is published after saving the modified record.

```
define("BasePageV2", [..., "LookupQuickAddMixin", ...],
    function(..., "LookupQuickAddMixin") {
        return {
            ...
            methods: {
                ...
                onSaved: function(response, config) {
                    ...
                    this.sendSaveCardModuleResponse(response.success);
                    ...
                },
                ...
                sendSaveCardModuleResponse: function(success) {
                    var primaryColumnName = this.getPrimaryColumnName();
                    var infoObject = {
                        action: this.get("Operation"),
                        success: success,
                        primaryColumnName: primaryColumnName,
                        uId: primaryColumnName,
                        primaryDisplayColumnName:
this.get(this.primaryDisplayName),
                        primaryDisplayColumnName: this.primaryDisplayName,
                        isInChain: this.get("IsInChain")
                    };
                    return this.sandbox.publish("CardModuleResponse", infoObject,
[this.sandbox.id]);
                },
                ...
            },
            ...
        };
    });
});
```

This functionality is implemented in the *BasePageV2* child schema (i.e. the *BaseEntityPage* schema is parental for the *BasePageV2* schema). Also, the *LookupQuickAddMixin* mixin is specified as a dependency in the *BasePageV2*. The subscription for the *CardModuleResponse* message is performed in this mixin.

A *mixin* is a class designed to extend the functions of other classes. Mixins expand the functionality of schemas, allowing to avoid duplication of commonly used logic in schema methods. Mixins are different from other modules added to the dependency list in a way that their methods can be addressed directly, much like those of a schema (see “[Mixins. The “mixins” property](#)”).

```
define("LookupQuickAddMixin", [...],
    function(..., "Terrasoft.configuration.mixins.LookupQuickAddMixin") {
        Ext.define("Terrasoft.configuration.mixins.LookupQuickAddMixin", {
```

```

        alternateClassName: "Terrasoft.LookupQuickAddMixin",
        ...
        // Declaration of the message.
        _defaultMessages: {
            "CardModuleResponse": {
                "mode": this.Terrasoft.MessageMode.PTP,
                "direction": "BIDIRECTIONAL"
            }
        },
        ...
        // Message registration method.
        _registerMessages: function() {
            this.sandbox.registerMessages(this._defaultMessages);
        },
        ...
        // Initializing an instance of a class.
        init: function(callback, scope) {
            ...
            this._registerMessages();
            ...
        },
        ...
        // Performed after adding a new record to the lookup.
        onLookupChange: function(newValue, columnName) {
            ...
            // Here, the method call chain is executed.
            // As a result, the _subscribeNewEntityCardModuleResponse () method will be called.
            ...
        },
        ...
        // The method in which the subscription to the "CardModuleResponse" message is performed.
        // In the reference field, the adding of the value sent when the message was published
        // is executed in the callback function.
        _subscribeNewEntityCardModuleResponse:
function(columnName, config) {
    this.sandbox.subscribe("CardModuleResponse", function(createdObj) {
        var rows = this._getResponseRowsConfig(createdObj);
        this.onLookupResult({
            columnName: columnName,
            selectedRows: rows
        });
        this, [config.moduleId]);
    },
    ...
});
return Terrasoft.LookupQuickAddMixin;
});

```

Adding a new address on the contact edit page is a good example of how bidirectional messages work.

1. After executing the command of adding a new record on the [Addresses] detail (Fig. When will the timeline be finished? 1), the *ContactAddressPageV2* module is loaded to the module chain and the edit page of the contact address opens (Fig. 2).

Fig. 1. Adding a new record on the [Addresses] detail

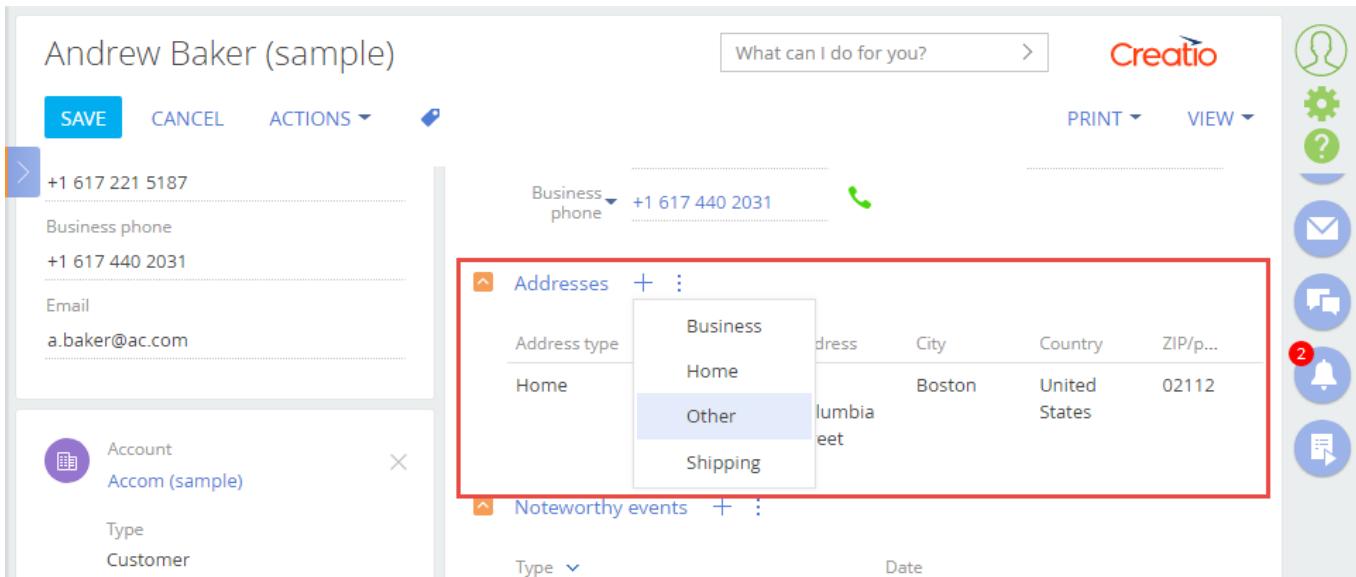
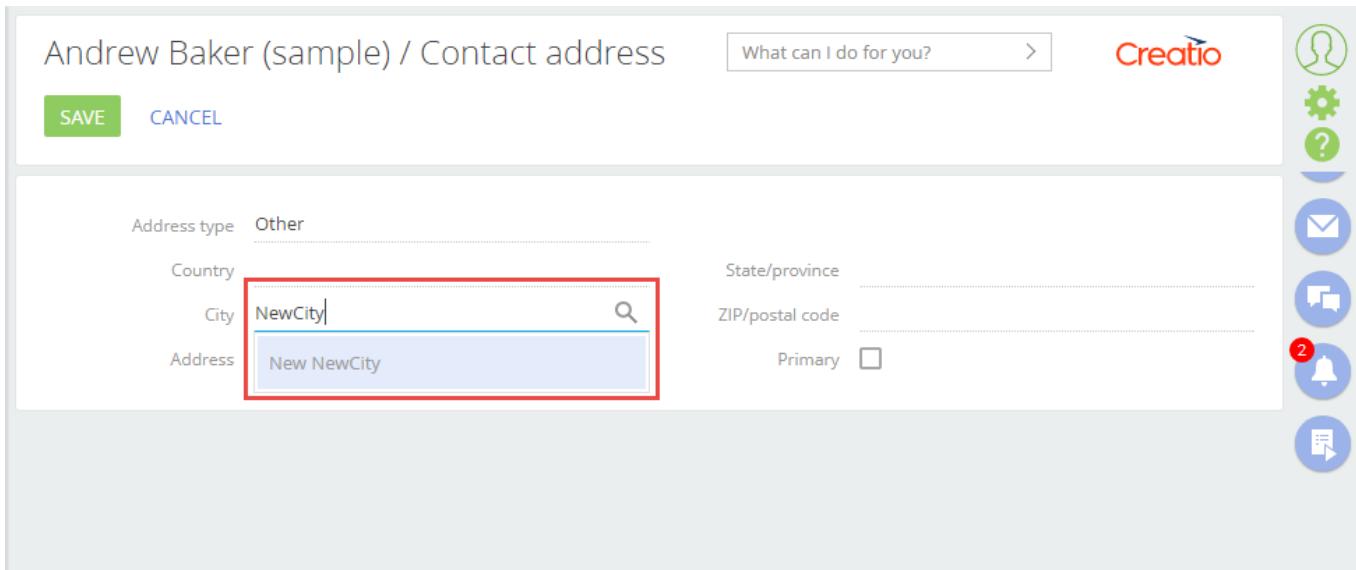


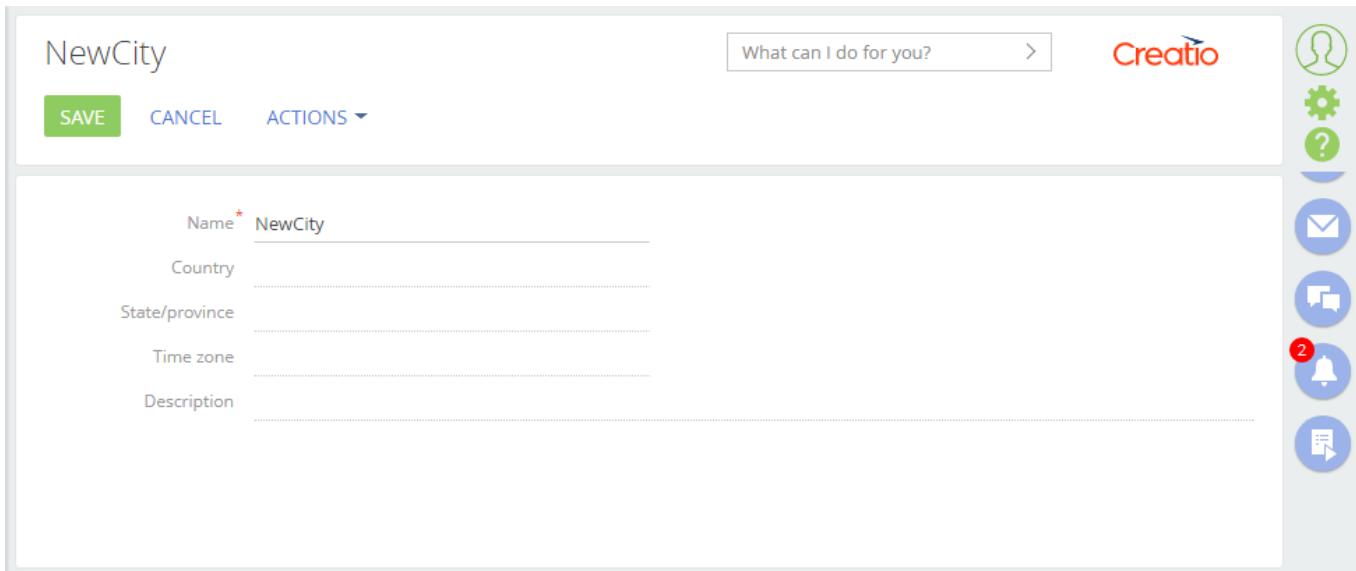
Fig. 2. Contact address edit page



The `CardModuleResponse` message has been already registered in the `ContactAddressPageV2` schema as it has the `BaseEntityPage` and the `BasePageV2` schemas in the inheritance hierarchy. This message is also registered in the `_registerMessages()` method of the `LookupQuickAddMixin` mixin at its initialization as the dependency module of the `BasePageV2`.

2. When adding the new value to the lookup fields of the `ContactAddressPageV2` page (for example, a new city (Fig.2)) the `onLookupChange()` method of the `LookupQuickAddMixin` mixin is called. In this method, in addition to loading the `CityPageV2` module to the module chain, the `_subscribeNewEntityCardModuleResponse()` method is called, in which the subscription for the `CardModuleResponse` message is performed. After this, the city edit page is opened (the `CityPageV2`, Fig. 3).

Fig. 3. City edit page



3. As the *CityPageV2* schema also has the *BasePageV2* schema in the inheritance hierarchy, the *onSaved()* method implemented in the base schema will be executed after saving the record (the [Save] button, Fig. 3). This method calls the *sendSaveCardModuleResponse()* method where the message (*CardModuleResponse*) is published. At the same time, the object with the necessary saving results is passed.

4. The execution of the *callback* subscriber function that process results of saving a new city in the lookup starts after publication of the message (see the *_subscribeNewEntityCardModuleResponse()* method of the *LookupQuickAddMixin* mixin).

The publication and subscription for the bidirectional message was performed in one inheritance hierarchy of the schemas with the *BasePageV2* base schema that contains all necessary functions.

See also

- **Module message exchange**
- **Messages. The "messages" property**

Loading and unloading modules

Beginner Easy Medium **Advanced**

Introduction

A Creatio module is an isolated software unit. It has no information about other Creatio modules apart from the module name list from which it depends. See “**Modular development principles in Creatio**” for more information about Creatio modules.

In certain situations, you might need to load modules that were not declared as dependencies when working with Creatio interface. To load and unload such modules, *sandbox.loadModule()* and *sandbox.unloadModule()* methods are designed.

Loading modules

Use the *sandbox.loadModule(moduleName, config)* method to load undeclared modules. Method parameters:

- *moduleName* – module name.
- *config* – configuration object that contains module parameters. This is a required parameter for visual modules.

Method *sandbox.loadModule()* call parameters:

```
// Loading a module without additional parameters.  
this.sandbox.loadModule("ProcessListenerV2");  
// Loading a module with additional parameters.
```

```
this.sandbox.loadModule("CardModuleV2", {
    renderTo: "centerPanel",
    keepAlive: true,
    id: moduleId
}) ;
```

Module parameters

Additional module loading parameters are aggregated in the *config* configuration object. Most common properties of this object are as follows:

- *id* – module Id. If the Id is not specified, it will be generated automatically. Data type – string.
- *renderTo* – name of the container where visual module view will be displayed. Passed as an argument to the *render()* method of the loaded module. Required for visual modules. Data type – string.
- *keepAlive* – indicates whether the module is added to a module thread. Used for navigation between the module views. Data type – Boolean.
- *isAsync* – indicates asynchronous initialization of the module (see “**Modular development principles in Creatio**”). Data type – Boolean.

Module class constructor parameters. The *instanceConfig* property.

There is an option to pass arguments to the class constructor of the instantiated module. To do this, add the *instanceConfig* property to the *config* configuration object and assign an object with the needed values to it.

The instantiated module is the one returning the a constructor function.

For example, the following module is declared:

```
// A module that returns a class instance.
define("CardModuleV2", [...], function(...) {
    // A class used for creating an edit page module.
    Ext.define("Terrasoft.configuration.CardModule", {
        // Class alias.
        alternateClassName: "Terrasoft.CardModule",
        // Parent class.
        extend: "Terrasoft.BaseSchemaModule",
        // Indicates that schema parameters have been set from without.
        isSchemaConfigInitialized: false,
        // Indicates that history state is used on module loading.
        useHistoryState: true,
        // Schema name of the displayed entity.
        schemaName: "",
        // Indicates that the common display mode (with the section list) is used.
        // If the value is false, SectionModule is present on the page.
        isSeparateMode: true,
        // Object schema name.
        entitySchemaName: "",
        // Primary column value.
        primaryColumnValue: Terrasoft.GUID_EMPTY,
        // Edit page mode. Possible values
        // ConfigurationEnums.CardStateV2.ADD|EDIT|COPY
        operation: ""
    });
    // Return class instance.
    return Terrasoft.CardModule;
})
```

Use the following code to pass the needed values to the module constructor on its loading:

```
// Object, whose properties contain values
// passed as constructor parameters
var configObj = {
    isSchemaConfigInitialized: true,
```

```

useHistoryState: false,
isSeparateMode: true,
schemaName: "QueueItemEditPage",
entitySchemaName: "QueueItem",
operation: ConfigurationEnums.CardStateV2.EDIT,
primaryColumnValue: "{3B58C589-28C1-4937-B681-2D40B312FBB6}"
};

// Loading module.
this.sandbox.loadModule("CardModuleV2", {
    renderTo: "DelayExecutionModuleContainer",
    id: this.getQueueItemEditModuleId(),
    keepAlive: true,
    // Specifying values passed to module constructor.
    instanceConfig: configObj
}
);

```

As a result, the module will be loaded with pre-set property values and no additional messages will be needed to set them.

The following types of properties can be passed to module instance:

- *string*;
- *boolean*;
- *number*;
- *date* (the value will be copied);
- *object* (Literal objects only. You cannot pass class instances, HTMLElement inheritors, etc.).

When passing parameters to module constructor of a Terrasoft.BaseObject inheritor, the following limitations apply: if a parameter is not described in the module class or one of parent classes, it cannot be passed.

Additional module parameters. The “parameters” property.

The *parameters* property is designed to pass additional parameters to a module on its loading in the *config* configuration object. Same property must be defined in the module class (or one of its parent classes) as well.

The *parameters* property is already defined in the *Terrasoft.BaseModule* class.

Thus, when instantiating the module, its *parameters* property will be initialized with values passed in the *parameters* property of the *config* object.

For example, the “parameters” property is defined in the *MiniPageModule* module:

```

define("MiniPageModule", ["BusinessRulesApplierV2", "BaseSchemaModuleV2",
"MiniPageViewGenerator"],
function(BusinessRulesApplier) {
    Ext.define("Terrasoft.configuration.MiniPageModule", {
        extend: "Terrasoft.BaseSchemaModule",
        alternateClassName: "Terrasoft.MiniPageModule",
        ...
        parameters: null,
        ...
    });
    return Terrasoft.MiniPageModule;
});

```

In this case, the pop-up summary module can be instantiated with additional parameters. Example:

```

define("MiniPageContainerViewModel", ["ConfigurationEnumsV2"], function() {
    Ext.define("Terrasoft.configuration.MiniPageContainerViewModel", {
        extend: "Terrasoft.BaseModel",
        alternateClassName: "Terrasoft.MiniPageContainerViewModel",
        ...
    });
    return Terrasoft.MiniPageContainerViewModel;
});

```

```

    ...
    loadModule: function() {
        // The "parameters" property is defined in the parent class
Terrasoft.BaseModel
        var parameters = this.get("parameters");
        ...
        this.sandbox.loadModule("MiniPageModule", {
            renderTo: Ext.get("MiniPageContainer"),
            id: moduleId,
            // Passing parameters to configuration object.
            parameters: parameters
        });
    }
    ...
});
);
}
);

```

Unloading modules

Use the `sandbox.unloadModule(id, renderTo, keepAlive)` method to unload module. Method parameters:

- `id` – module Id. Data type – string.
- `renderTo` – name of the container where visual module view will be deleted. Required for visual modules. Data type – string.
- `keepAlive` – indicates if the module model is saved. On unloading the module, the core can save its model for using its properties, methods and messages. Data type – Boolean. Not recommended.

Method `sandbox.unloadModule()` call parameters:

```

...
// Method that obtains Id of unloaded module.
getModuleId: function() {
    return this.sandbox.id + "_ModuleName";
},
...
// Unloading a non-visual module.
this.sandbox.unloadModule(this.getModuleId());
...
// Unloading a visual module, previously loaded to the "ModuleContainer" container.
this.sandbox.unloadModule(this.getModuleId(), "ModuleContainer");

```

Module thread

Sometimes a model view must be shown in place of other model. For example, to set a field value on the current page, a `SelectData` lookup selection page must be displayed. In such cases, the current page module must not be unloaded, but the module view of the lookup selection page must be displayed in place of its container. For this, use module threads:

To begin a new module thread, add the `keepAlive` property to the configuration object of the loaded module. For example, you need to display lookup selection module (`selectDataModule`) in the current page module (`CardModule`). To do this, execute the following code:

```

sandbox.loadModule("selectDataModule", {
    // Id of the loaded module view.
    id: "selectDataModule_id",
    // The view will be added to the current page container.
    renderTo: "cardModuleContainer",
    // Denies unloading of the current module.
    keepAlive: true
});

```

After the code is executed, a module thread consisting of the current page module and lookup selection module will be created. Clicking the [Add new record] button in the current page module (`selectData`) will open a new page and

add another element to the thread. Thus, unlimited number of modules can be added to the module thread. Active module (the one displayed on the page) will always be the last element in the thread. If an element in a thread is set as active, all elements after it will be destroyed. To activate a chain element, call the *loadModule* function and pass module Id to its parameter:

```
sandbox.loadModule("someModule", {  
    id: "someModuleId"  
});
```

The core will destroy all elements in the thread after the specified one and execute standard module loading logic, calling the *init()* and *render()* methods. The container where the previous active module was placed will be passed to the *render()* method. All modules in the thread can work as before, receiving and sending messages, saving data, etc.

If the *keepAlive* property is not added to the configuration object when the *loadModule()* method is called, or if this property is added as *keepAlive: false*, the module thread will be destroyed.

Client view model schemas

Contents

- [Introduction](#)
- [Mixins. The "mixins" property](#)
- [Attributes. The "attributes" property](#)
- [Messages. The "messages" property](#)
- [Properties. The "properties" property](#)
- [Methods. The "methods" property](#)
- [Rules. The "rules" property](#)
- [Business rules. The businessRules property](#)
- [Modules. The "modules" property](#)
- [The "diff" array](#)
- [The "diff" array. Alias mechanism](#)
- [Schema formatting requirements for compatibility with wizards](#)

Client view model schemas

Beginner

Easy

Medium

Advanced

Introduction

A *custom view model schema* is a visual module schema that implements client part of the application. Custom view model schema is a kind of configuration object for generating views and view models by the *ViewGenerator* and *ViewModelGenerator* generators of Creatio. Custom module types are described in the "**Module types and their specificities**" article.

Source code structure of custom schema

All schemas have a common structure. Schema source code example:

```
define("ExampleSchema", [], function() {  
    return {  
        entitySchemaName: "ExampleEntity",  
        mixins: {},  
        attributes: {},  
        messages: {},  
        methods: {},  
        rules: {},  
        businessRules: /**SCHEMA_BUSINESS_RULES*/{}/**SCHEMA_BUSINESS_RULES*/,  
        modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/,  
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
```

```
};  
});
```

A Schema configuration object is returned by anonymous factory function that is called after loading the module. The object can have following properties:

entitySchemaName – object (model) schema name that will be used by this client schema.

mixins – configuration object that contains mixin declaration. More information about the mixins you can find in the "**Mixins. The "mixins" property**" article.

attributes – configuration object that contains schema attributes. More information about the attributes you can find in the "**Attributes. The "attributes" property**" article.

messages – configuration object that contains schema messages. More information you can find in the "**Messages. The "messages" property**" article.

methods – configuration object that contains schema methods. More information about this property you can find in the "**Methods. The "methods" property**" article.

rules – configuration object that contains schema business rules. More information about this property you can find in the "**Rules. The "rules" property**" article.

businessRules – configuration object that contains schema business rules, which are created or edited via the section wizard or detail wizard. Marker comments `/**SCHEMA_BUSINESS_RULES*/` are used by the wizards and therefore are required. More information about this property you can find in the "**Business rules. The businessRules property**" article.

modules – configuration object that contains schema modules. Marker comments `/** SCHEMA_MODULES */` are used by the wizards and therefore are required. More information about this property you can find in the "**Modules. The "modules" property**" article.

To load a detail on a page the *details* property is used. But the detail is a module and it is appropriate to use the *modules* property.

diff – configuration object array that contains schema view description. Marker comments `/** SCHEMA_DIFF */` are used by the wizards and therefore are required. For more information about the *diff* array configuration, please refer to the "**The "diff" array**" article.

properties – configuration object which contains the view model properties. Detailed information about this property is available in the "**Properties. The "properties" property**" article.

\$-properties – automatically generated properties for the view model attributes. More information can be found in the "**Automatically generated view model properties (on-line documentation)**" article.

Mixins. The "mixins" property

Beginner

Easy

Medium

Advanced

Introduction

A *mixin* is a class designed to extend the functions of other classes. JavaScript does not support multiple inheritances. Mixins, on the other hand, enable extending the schema functionality without duplicating the logic used in the schema methods. Unlike other modules connected to the dependency list, mixins' methods can be addressed directly, much like those of a schema.

Working with mixins

General algorithm

1. Create a mixin.
2. Assign a name to the mixin.
3. Connect the corresponding name array.
4. Implement the mixin functionality.
5. Use the mixin in the custom schema.

Creating a mixin

Creating a mixin is no different from creating an object schema. The procedure for creating a **[Module]** type schema is covered in the “**Creating a custom client module schema**” article.

Assigning a name to the mixin

When naming mixins, use the *-able* suffix in the schema name (e.g., *Serializable* – a mixin that adds a serializing ability to the components). If you cannot make a mixin name as per the formula above, add the *Mixin* ending to the schema name.

You cannot use words like *Utilities*, *Extension*, *Tools*, etc. since this does not enable formalizing the functionality of the mixin.

Learn more about naming mixins in the "[A New Mixin Naming Convention](#)" article.

Connecting the name array

You need to connect a corresponding name array for the mixin (*Terrasoft.configuration.mixins* – for the configuration, *Terrasoft.core.mixins* – for the core).

Implementing the mixin functionality

Mixins are designed in a form of modules that must be connected to the schema dependency list when the schema is declared by the *define()* function.

The same set of actions can be used in different custom schemas of the Creatio application. Create a mixin to avoid duplicating the code in each schema.

The mixin structure looks as follows:

```
define("MixinName", [], function() {
    Ext.define("Terrasoft.configuration.mixins.MixinName", {
        alternateClassName: "Terrasoft.MixinName",
        // Mixin functionality.
    });
    return Ext.create(Terrasoft.MixinName);
})
```

Mixins should not depend on the internal implementation of the schema they will be applied to. This should be an independent mechanism that receives a set of parameters, processes them and returns a result, if needed.

Using the mixin

The mixin implements the functionality needed in the custom schema. To receive the set of mixin actions, specify it in the *mixins* block of the custom schema.

The mixin connection structure looks as follows:

```
// MixinName – the module where the mixin class is implemented.
define("CustomSchemaName", ["MixinName"], function () {
    return {
        // SchemaName – the name of the entity schema.
        entitySchemaName: "SchemaName",
        mixins: {
            // Connecting the mixin.
            MixinName: "Terrasoft.Namespace.MixinName"
        },
        attributes: {},
        messages: {},
        methods: {},
        rules: {},
        modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*/,
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    }
})
```

```
};  
});
```

When using mixins, calling methods becomes quicker than when using separate schemas.

After you connect the mixin, you can use all of its methods, attributes and fields in the custom schema as if they were part of the current custom schema.

If you need to override a function from the mixin, just create a function with similar name in the custom schema. As a result, the function of the schema and not that of the mixin will be used when calling.

Case description

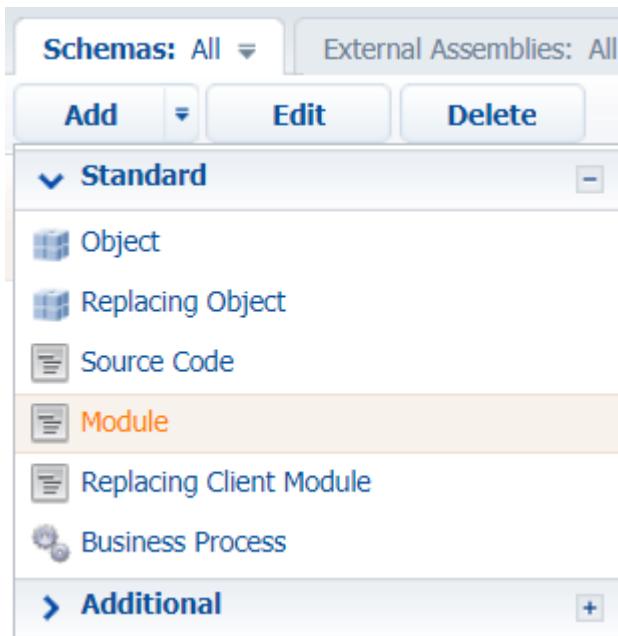
Connect the *ContentImageMixin* mixin to the custom schema, override the mixin method.

Case implementation algorithm

1. Create a mixin

Go to the [Advanced settings] section -> **[Configuration]** -> Custom package -> **[Schemas]**. Click **[Add]** -> **[Module]** (Fig. 1). Creating a module is covered in the “[Creating a custom client module schema](#)” article.

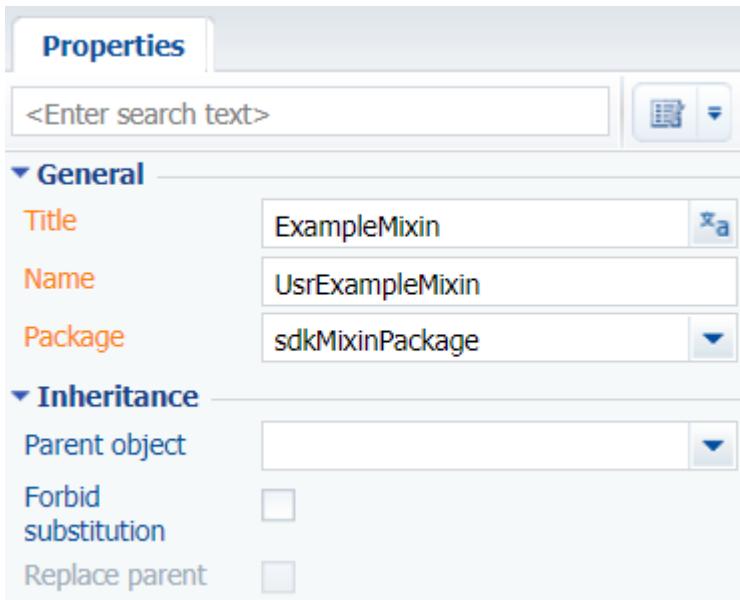
Fig. 1. Adding a module



Specify the following parameters for the created object schema (Fig. 2):

- **[Title]** – "ExampleMixin";
- **[Name]** – "UsrExampleMixin".

Fig. 2. – Configuring the object schema of the [Module] type



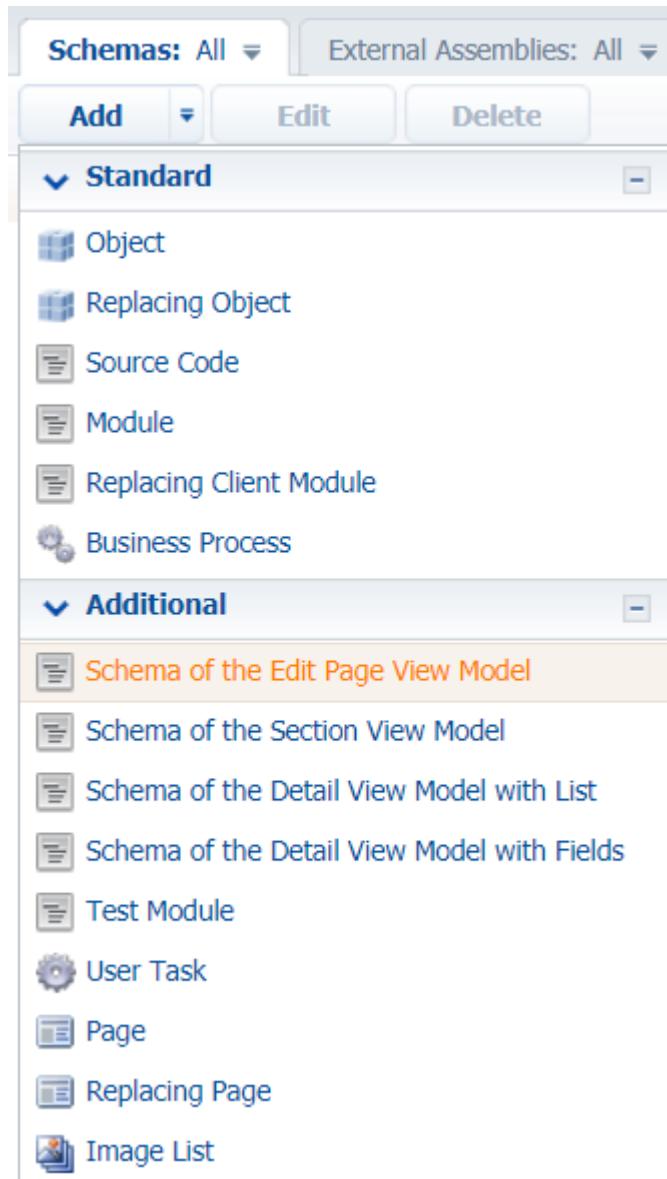
The complete source code of the module is available below:

```
// Defining the module.
define("ContentImageMixin", [ContentImageMixinV2Resources], function() {
    // // Defining the ContentImageMixin class.
    Ext.define("Terrasoft.configuration.mixins.ContentImageMixin", {
        // Alias (a short class name).
        alternateClassName: "Terrasoft.ContentImageMixin",
        // Mixin functionality.
        getImageUrl: function() {
            var primaryImageColumnName = this.get(this.primaryImageColumnName);
            if (primaryImageColumnName) {
                return this.getSchemaImageUrl(primaryImageColumnName);
            } else {
                var defImageResource = this.getDefaultImageResource();
                return this.Terrasoft.ImageUrlBuilder.getUrl(defImageResource);
            }
        }
    });
    return Ext.create(Terrasoft.ContentImageMixin);
})
```

2. Connect the mixin

Run the **[Add] → [Schema of the Edit Page View Model]** menu command on the **[Schemas]** tab in the **[Configuration]** section of the custom package. The procedure for creating a view model schema of the edit page is covered in the “**Creating a custom client module schema**” article.

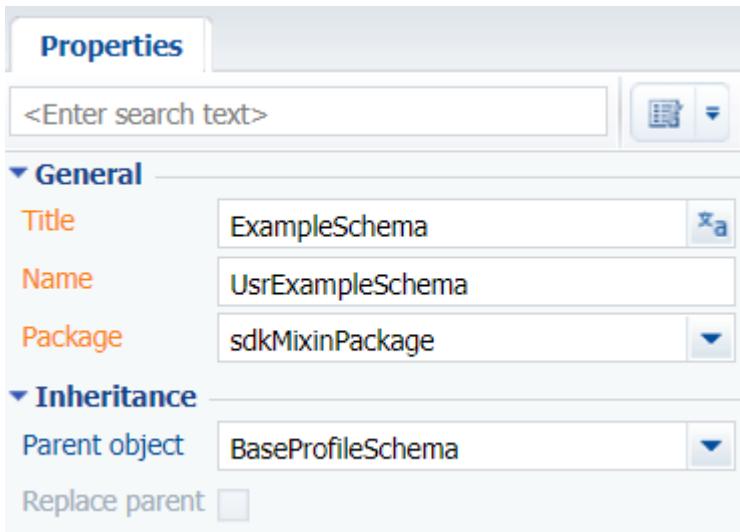
Fig. 3. – Adding a view model schema of the edit page



Specify the following parameters for the created object schema (Fig. 4):

- **[Title]** – "ExampleSchema";
- **[Name]** – "UsrExampleSchema".

Fig. 4. – Configuring the schema of the [Schema of the Edit Page View Module] type



To use the mixin, connect it in the *mixins* block of the *ExampleSchema* custom schema.

3. Override the mixin method

In the *method* block, override the *getReadImageUrl()* mixin method. Use the overridden function in the *diff* block.

The complete source code of the module is available below:

```
// Declaring the module. Specify the
// ContentImageMixin module where the mixin class is declared as a dependency.
define("UsrExampleSchema", ["ContentImageMixin"], function() {
    return {
        entitySchemaName: "ExampleEntity",
        mixins: {
            // Connecting the mixin to the schema.
            ContentImageMixin: "Terrasoft.ContentImageMixin"
        },
        details: {},
        diff: [
            {
                "operation": "insert",
                "parentName": "AddRightsItemsHeaderImagesContainer",
                "propertyName": "items",
                "name": "AddRightsReadImage",
                "values": {
                    "classes": {
                        "wrapClass": ["rights-header-image"]
                    },
                    "getSrcMethod": "getReadImageUrl",
                    "imageTitle": resources.localizableStrings.ReadImageTitle,
                    "generator": "ImageCustomGeneratorV2.generateSimpleCustomImage"
                }
            ],
            methods: {
                getReadImageUrl: function() {
                    // Custom implementation.
                    console.log("Contains custom logic");
                    // Calling the mixin method.
                    this.mixins.ContentImageMixin.getImageUrl.apply(this, arguments);
                }
            },
            rules: {}
        };
    });
});
```

After making changes, save and publish the schema.

Attributes. The "attributes" property

Beginner

Easy

Medium

Advanced

Introduction

Attributes is a configuration object property of the view model schema. It contains configuration objects that describe the model attributes. A model column is the attribute. All object schema columns are included in the *attributes* collection automatically upon generation.

Attribute base properties

The schema attributes have the following base properties:

- *dataValueType* – attribute data type. This property is used for generation of views. The available data types are represented by the *Terrasoft.DataValueType* enumeration.
- *type* – column type. Optional parameter used in the *BaseViewModel* internal work. The available column types are represented by the *Terrasoft.ViewModelColumnType* enumeration.
- *value* – the attribute value. The value of this parameter will be set in the view model at its creation.

You can specify numeric, string and Boolean values in the *value* attribute.

If the attribute type involves the use of a reference type value (array, object, collection, etc.), its initial value must be initialized using methods.

An example of using attribute base properties:

```
attributes: {
    // Attribute name.
    "NameAttribute": {
        // Data type.
        "dataValueType": this.Terrasoft.DataValueType.TEXT,
        // Column type.
        "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
        // Default value.
        "value": "NameValue"
    }
}
```

Attributes additional properties

Attributes can have the following properties:

- *Caption* – attribute title.
- *isRequired* – indicates whether the attribute is required.
- *Dependencies* – dependency from another model attribute. For example, setting dependencies of an attribute on the value of another attribute. The property is used to create calculated fields. More information about the calculated fields and the uses of this parameter can be found in the "**Adding calculated fields**" article.
- *lookupListConfig* – property that configures lookup field features. More information about this parameter can be found in the "**Using filtration for lookup fields. Examples**" article. This is a configuration object that can contain the following optional properties:
 - *columns* – an array of column names that will be added to the query with the Id column and the primary display column.
 - *orders* – an array of configuration objects that determine the sorting of displayed data.
 - *filter* – a method that returns an object of *Terrasoft.BaseFilter* class or its inheritor that will be applied to the query. Can not be used with the "filters" property.
 - *filters* – filters array (methods that return collections of the *Terrasoft.FilterGroup* class). Can not be used with the filter property.

An example of using attribute additional properties:

```
attributes: {
    // Attribute name.
    "Client": {
        // Attribute header.
        "caption": { "bindTo": "Resources.Strings.Client" },
        // Attribute is required.
        "isRequired": true
    },
    // Attribute name.
    "ResponsibleDepartment": {
        lookupListConfig: {
            // Additional columns.
            columns: [ "SalesDirector" ],
            // Sort column.
            orders: [ { columnPath: "FromBaseCurrency" } ],
            // Filter definition function.
            filter: function()
            {
                // Returns filter of Type column, which is equal the "Competitor" constant.
                return this.Terrasoft.createColumnFilterWithParameter(
                    this.Terrasoft.ComparisonType.EQUAL,
                    "Type",
                    ConfigurationConstants.AccountType.Competitor);
            }
        }
    },
    // Attribute name.
    "Probability": {
        // Determination of the column dependency.
        "dependencies": [
            {
                // Depends on the "Stage" column.
                "columns": [ "Stage" ],
                // The name of the handler method for the "Stage" column change.
                // setProbabilityByStage() method is defined in methods property
                // of schema object.
                "methodName": "setProbabilityByStage"
            }
        ]
    }
},
methods: {

    // "Stage" column modification handler method
    setProbabilityByStage: function()
    {
        // Getting the Stage column value.
        var stage = this.get("Stage");
        // The condition for the "Probability" column modification.
        if (stage.value && stage.value ===
            ConfigurationConstants.Opportunity.Stage.RejectedByUs)
        {
            // Setting the "Probability" column value.
            this.set("Probability", 0);
        }
    }
}
```

Messages. The "messages" property

Beginner

Easy

Medium

Advanced

Introduction

Data exchange between modules is organized through messages.

There are two message modes:

- *Address*. Address messages are only received by the last subscriber. To switch to address mode, set the *mode* property to *this.Terrasoft.MessageMode.PTP*.
- *Broadcasting*. Broadcasting messages are received by all subscribers. To switch to broadcasting mode, set the *mode* property to *this.Terrasoft.MessageMode.BROADCAST*.

The list of available message modes is represented by the *Terrasoft.MessageMode* enumeration.

There are two message directions:

- *Publication* – a message that can only be published (outbound). To set the direction for message publishing, set the *direction* property to *this.Terrasoft.MessageDirectionType.PUBLISH*.
- *Subscription* – a message that can only be subscribed to (inbound). To set the direction for message subscription, set the *direction* property to *this.Terrasoft.MessageDirectionType.SUBSCRIBE*.

The same message can not be announced with different directions in the schema inheritance hierarchy.

Message use examples

Message publishing

Declare a message with the “publishing” direction in the schema you want to publish the message in.

```
messages: {
    // Message name.
    "GetColumnsValues": {
        // Message type - address.
        mode: this.Terrasoft.MessageMode.PTP,
        // Message direction - publication
        direction: this.Terrasoft.MessageDirectionType.PUBLISH
    }
}
```

Publishing is done through calling the *publish* method from the *sandbox* class instance.

```
// The GetColumnsValues method for obtaining message publication results.
getColumnsValues: function(argument) {
    // Message publishing.
    return this.sandbox.publish("GetColumnsValues", argument, ["key"]);
}
```

In this code:

- “*GetColumnsValues*” – message name.
- *Argument* – the argument passed to the handler function of the subscriber. An object with message parameters.
- *["Key"]* – an array of tags for filtering messages.

The *sandbox* property is declared in all schemas.

Message publishing can return the handler function result only in the “address” mode.

Message subscription

A message with the “subscription” direction should be declared in the subscription schema.

```
messages: {
    // Message name.
    "GetColumnsValues": {
        // Message type - address.
        mode: this.Terrasoft.MessageMode.PTP,
        // Message direction - subscription.
        direction: this.Terrasoft.MessageDirectionType.SUBSCRIBE
    }
}
```

A subscription is carried out by calling the *subscribe* method in the *sandbox* class instance.

```
this.sandbox.subscribe("GetColumnsValues", messageHandler, context, ["key"]);
```

In this code:

- “GetColumnsValues” – message name.
- *messageHandler* – message handler function.
- *Context* – handler function execution context.
- ["Key"] – an array of tags for filtering messages.

In the “address” mode, the *messageHandler* method should return the object, which is processed as the result of message publication.

```
methods: {
    messageHandler: function(args) {
        // Returning the object that is being processed as a result of message
        publishing.
        return { };
    }
}
```

In broadcast mode, the *messageHandler* method returns nothing.

```
methods: {
    messageHandler: function(args) {
    }
}
```

Properties. The “properties” property

Beginner

Easy

Medium

Advanced

Introduction

The “*properties*” property of the configuration object of the view model schema contains a JavaScript object that describes the properties of the view model.

An example of using the “*properties*” property in the *SectionTabsSchema* schema of the *NUI* package:

```
define("SectionTabsSchema", [],
    function() {
        return {
            ...
            // Declaring the "properties" property.
            properties: {
                // The "parameters" property. Array.
                parameters: [],
                // The "modulesContainer" property. Object.
                modulesContainer: {}
            },
            methods: {

```

```

    ...
    // Initialization method. Always executed first.
    init: function(callback, scope) {
        ...
        // Calling a method that uses the properties of the view
model.
        this.initContainers();
        ...
    },
    ...
    // A method that uses the properties.
    initContainers: function() {
        // Using the "modulesContainer" property.
        this.modulesContainer.items = [];
        ...
        // Usting the "parameters" property.
        this.Terrasoft.each(this.parameters, function(config) {
            config = this.applyConfigs(config);
            var moduleConfig = this.getModuleContainerConfig(config);
            var initConfig = this.getInitConfig();
            var container =
viewGenerator.generatePartial(moduleConfig, initConfig)[0];
            this.modulesContainer.items.push(container);
        }, this);
    },
    ...
},
...
}
);

```

Methods. The "methods" property

Beginner Easy **Medium** Advanced

Introduction

The methods property of the view model schema contains a collection of methods that form the business logic of the schema and affect the view model. Create new methods and override (replace) base methods of parent schemas in this property. By default, the scope of methods is the view model scope.

Examples of method declaration

An example of a replaced method

Add the [Email] column completion requirement logic to the *setValidationConfig* method logic of the *Terrasoft.configuration.BaseSchemaViewModel* class.

```

methods: {
    // Method name.
    setValidationConfig: function() {
        // Calling the logic of the setValidationConfig parent schema method.
        this.callParent(arguments);
        // Setting up the validation for the [Email] column.
        this.addColumnValidator("Email", EmailHelper.getEmailValidator);
    }
}

```

An example of a new method

```
methods: {
```

```
// Method name.  
getBlankSlateHeaderCaption: function() {  
    // Accessing the MasterColumnInfo column values.  
    var masterColumnInfo = this.get("MasterColumnInfo");  
    // Returning method work results.  
    return masterColumnInfo ? masterColumnInfo.caption : "";  
},  
// Method name.  
getBlankSlateIcon: function() {  
    // Returning method work results.  
    return  
this.Terrasoft.ImageUrlBuilder.getUrl(this.get("Resources.Images.BlankSlateIcon"));  
}  
}
```

Rules. The "rules" property

Beginner Easy **Medium** Advanced

Introduction

Rules is a standard system mechanism, which enables the developer to add an implementation of typical functions by configuring view model columns.

The functions of rules are implemented in the *BusinessRuleModule* client module. Add the *BusinessRuleModule* module to the list of schema dependencies to use these functions.

```
define("CustomPageModule", ["BusinessRuleModule"],  
    function(BusinessRuleModule) {  
        return {  
            // Client module implementation  
        };  
    });
```

Rule types are defined in the *RuleType* enumeration of the *BusinessRuleModule* module.

General procedure for declaring the rules:

- All rules are described in the *rules* property of the schema.
- The rules are applied to view model columns, not to controls.
- Rules have names.
- Rule parameters are set in its configuration object.

To learn more about business rules and to see the examples of their use, please refer to the “**Setting the edit page fields using business rules**” chapter.

Business rules. The businessRules property

Beginner Easy **Medium** Advanced

Introduction

In Creatio, the behavior configuration of page / detail fields is done through business rules. Using business rules, you can configure the following field behavior:

- Hiding and displaying fields
- Enable or disable editing
- Compulsory or optional
- Filtering lookup fields depending on other field values

Unlike the business rules defined in the *rules* property of the page view model schema (see “**Rules. The "rules"**

property”), the business rules defined in the *businessRules* property are generated by the detail or the section wizard and have a higher execution priority. The *BusinessRuleModule* enumeration is not used when describing the generated business rule.

When creating a new business rule, the wizard generates a name for it and adds it to the custom schema of the edit page view model.

If the business rule is disabled, the *enabled* property of its configuration object is set to *false*.

When you delete a business rule, its configuration object remains in the custom schema of the edit page view model, but the *removed* property is set to *true*.

We do not recommend editing the *businessRules* property manually!

Editing an existing business rule

After editing the custom business rule in the wizard, the business rule configuration object remains unchanged in the *rules* property of the edit page view model. This creates a new version of the business rule configuration object with the same name in the *businessRules* property.

The business rule defined in the *businessRules* property has a higher execution priority when processing a business rule at runtime. Therefore, subsequent changes to this rule in the *rules* property will not affect the system in any way.

When you delete or disable the business rule, the changes made in the configuration object of the *businessRules* property have a higher priority.

Modules. The "modules" property

Beginner

Easy

Medium

Advanced

Introduction

The *modules* property contains a configuration object responsible for declaration and configuration of modules and details loaded to a page. The `/** SCHEMA_MODULES */` marker comments are required, since they are necessary for the work of the wizards.

To load a detail to a page, use the *details* property. However, since details are essentially modules, we recommend using the *modules* property instead.

An example of using the modules property

```
modules: /**SCHEMA_MODULES*/ {
    // Loading the module
    // Module title. Must be the same as the name property in the diff massive.
    "TestModule": {
        // Optional Loaded module id Will be generated by the system if not
        // specified.
        "moduleId": "myModuleId",
        // If the parameter is not specified, BaseSchemaModuleV2 will be used for
        // loading.
        "moduleName": "MyTestModule",
        // Configuration object. When the module is loaded, it is passed as
        // instanceConfig. It stores a set of initial parameter values for the module.
        "config": {
            "isSchemaConfigInitialized": true,
            "schemaName": "MyTestSchema",
            "useHistoryState": false,
            // Additional module parameters.
            "parameters": {
                // Parameters added to a schema during its initialization.

                "viewModelConfig": {
                    masterColumnName: "PrimaryContact"
                }
            }
        }
    }
}
```

```
        }
    },
},
// Loading a detail.
// Detail name.
"Project": {
    // The name of a schema detail.
    "schemaName": "ProjectDetailV2",
    "filter": {
        // Section object schema column.
        "masterColumn": "Id",
        // Detail object schema column.
        "detailColumn": "Opportunity"
    }
}
}/**SCHEMA_MODULES*/
```

The "diff" array

Beginner Easy **Medium** Advanced

Introduction

The “diff” array is an array of modifications described in the “*diff*” property of a schema. The array is used for generating module views in the system UI. Each array element is a metadata. The UI controls are generated based on these metadata.

The *diff* property contains an array of configuration objects that are responsible for schema display. The *diff* array contains objects that configure display of containers, controls, modules, fields and other visual components.

The **diff** array object properties

The *diff* array elements have the following properties:

- *operation* – can have the following values:
 - *set* – schema element value is set by the *values* parameter.
 - *merge* – the values from the parent, replacing and replacement schemas are merged. The properties from *values* parameter have the highest priority.
 - *remove* – the element is removed from the schema.
 - *move* – the element is moved to another parent element.
 - *insert* – the element is inserted in the schema.
- *name* – the name of schema element that the operation is applied to.
- *parentName* – the name of schema parent element where the element is placed as a result of the *insert* or *move* operation;
- *propertyName* – the name of parent element parameter in the *insert* operation. Also used in the *remove* operation if it is needed to remove specific element parameters and not the element itself;
- *index* – the index in which the parameter is being moved or inserted. The parameter is used in the *insert* and *move* operations. If the parameter is not specified, the element will be inserted as the last element in the array.
- *values* – the object whose properties will be set or merged with schema element properties. It is used in the *set*, *merge* and *insert* operations.

Creatio has a set of basic elements that can be displayed on a page. They are specified in the *Terrasoft.ViewItemType* list (Table. 1).

Table 1. – Element type

Name	Description
------	-------------

GRID_LAYOUT	Grid element that contains placements of other elements
TAB_PANEL	Set of tabs.
DETAIL	Detail.
MODEL_ITEM	View model element.
MODULE	Module.
BUTTON	Button.
LABEL	Label.
CONTAINER	Container.
MENU	Drop-down list.
MENU_ITEM	Drop-down list element.
MENU_SEPARATOR	Drop-down list separator.
SECTION_VIEWS	Section views.
SECTION_VIEW	Section view.
GRID	List.
SCHEDULE_EDIT	Scheduler.
CONTROL_GROUP	Group of controls.
RADIO_GROUP	Group of radio buttons.
DESIGN_VIEW	Customizable view.
COLOR_BUTTON	Color.
IMAGE_TAB_PANEL	Set of tabs with icons.
HYPERLINK	Hyperlink.
INFORMATION_BUTTON	Information button with tooltip.
TIP	Tooltip.

An example of using the “diff” property

```
diff: /**SCHEMA_DIFF*/ [
  {
    "operation": "insert",
    "name": "CardContentWrapper",
    "values": {
      "id": "CardContentWrapper",
      "itemType": Terrasoft.ViewItemType.CONTAINER,
      "wrapClass": ["card-content-container"],
      "items": []
    }
  },
  {
    "operation": "insert",
    "name": "CardContentContainer",
    "parentName": "CardContentWrapper",
    "propertyName": "items",
    "values": {
      "itemType": Terrasoft.ViewItemType.CONTAINER,
      "items": []
    }
  },
  {
    "operation": "insert",
    "name": "HeaderContainer",
    "values": {
      "itemType": Terrasoft.ViewItemType.CONTAINER,
      "items": []
    }
  }
]
```

```

    "parentName": "CardContentContainer",
    "propertyName": "items",
    "values": {
        "itemType": Terrasoft.ViewItemType.CONTAINER,
        "wrapClass": ["header-container-margin-bottom"],
        "items": []
    }
},
{
    "operation": "insert",
    "name": "Header",
    "parentName": "HeaderContainer",
    "propertyName": "items",
    "values": {
        "itemType": Terrasoft.ViewItemType.GRID_LAYOUT,
        "items": [],
        "collapseEmptyRow": true
    }
}
]/**SCHEMA_DIFF*/

```

The "diff" array. Alias mechanism

Beginner

Easy

Medium

Advanced

General Information

The *Alias* mechanism – provides partial backward compatibility when user interface is changed in the new versions of the product. In the process of developing new versions, sometimes it becomes necessary to move page elements to new areas. In case the users have customized the page, the changes could lead to unpredictable consequences. The *Alias* mechanism helps to avoid this by interacting with the json-applier class which is *diff* array builder. This class merges all the parameters of the base and custom replacing schemas.

Diff – an array of objects, responsible for displaying schema elements. It can have containers, controls, modules and fields. For more information about the *diff* array, see the "[The "diff" array](#)" article.

Details

The *alias* property contains information about the previous element name. This information is taken into account when building a *diff* array of modifications, and informs that both the elements with a new name, and the elements with the name specified in *alias* must be considered. The *alias* is a configuration object that links two different elements – the new one and the old one. When building a *diff* array the *alias* configuration object can be used to exclude application of certain properties and operations to the element in which the alias is declared. The *alias* object can be added to any element in *diff* array.

Alias object structure

The *alias* object contains three custom properties:

- *name* – the name associated with the new element. This name will be used to locate the elements in the replaced schemas and connect them with the new element.

The value of the *name* element of the *diff* array should not be equal to the *alias.name* property.

- *excludeProperties* – array of properties of the *values* object of the *diff* modification array element. These properties will not be used when generating *diff*.
- *excludeOperations* – array of operations that should not be applied to this element when the *diff* modification array is generated.

Usage example of the *alias* object :

```
// diff. array
diff: /**SCHEMA_DIFF*/ [
```

```
{
    // The operation with the element.
    "operation": "insert",
    // Element new name.
    "name": "NewElementName",
    // Element values.
    "values": {
        // ...
    },
    // Alias configuration object.
    "alias": {
        // Element previous name.
        "name": "OldElementName",
        // Exclude properties array.
        "excludeProperties": [ "layout", "visible", "bindTo" ],
        // Exclude operations array.
        "excludeOperations": [ "remove", "move", "merge" ]
    }
},
//...
]
```

An example of the Alias mechanism usage for multiple schema replacement

There is an initial element of the *diff* array with the name "Name" and a set of properties. The element is located in the *Header* container. This schema was replaced several times and each time the "Name" element is moved and modified in every possible way.

Diff property of the base schema

```
diff: /**SCHEMA_DIFF*/ [
{
    // Insert operation.
    "operation": "insert",
    // The name of the parent element to insert into.
    "parentName": "Header",
    // The name of the parent element with which operation is performed.
    "propertyName": "items",
    // Element name.
    "name": "Name",
    // Element property values object.
    "values": {
        // Layout.
        "layout": {
            // Column number.
            "column": 0,
            // Row number.
            "row": 1,
            // Number of joined columns.
            "colSpan": 24
        }
    }
}
] /**SCHEMA_DIFF*/
```

Diff property after first replacement of the base schema:

```
diff: /**SCHEMA_DIFF*/ [
{
    // Merging properties of the two elements.
    "operation": "merge",
    "name": "Name",
    "values": {
```

```

    "layout": {
      "column": 0,
      // Row number. The element is moved.
      "row": 8,
      "colSpan": 24
    }
  }
]
/**SCHEMA_DIFF*/

```

Diff property after second replacement of the base schema:

```

diff: /**SCHEMA_DIFF*/ [
{
  //Moving the element.
  "operation": "move",
  "name": "Name",
  //The name of the parent element where the element is moved.
  "parentName": "SomeContainer"
}
]
/**SCHEMA_DIFF*/

```

In the new version, the "Name" element was moved from the *SomeContainer* element to the *ProfileContainer* element and must remain there regardless of the client customization. For this, the element gets a new name "NewName" and an *alias* configuration object is added to it.

```

diff: /**SCHEMA_DIFF*/ [
{
  // Insert operation.
  "operation": "insert",
  // The name of the parent element in which insert is carried out.
  "parentName": "ProfileContainer",
  // The name of the parent element property with which operation is performed.
  "propertyName": "items",
  // Element new name.
  "name": "NewName",
  // Object with element property values.
  "values": {
    // Binding to a property value or a function
    "bindTo": "Name",
    // Layout.
    "layout": {
      // Column number.
      "column": 0,
      // Row number.
      "row": 0,
      // Number of joined columns.
      "colSpan": 12
    }
  },
  // Alias configuration object.
  "alias": {
    // Element previous name.
    "name": "Name",
    // Array of excluded properties.
    "excludeProperties": [ "layout" ],
    // Array of ignored operations.
    "excludeOperations": [ "remove", "move" ]
  }
}
]
/**SCHEMA_DIFF*/

```

Alias has been added in the new element. The parent element and its location on the edit page also has been changed. The *excludeProperties* property stores a set of properties that will be ignored when the difference is applied. The *excludeOperations* property stores a set of operations that will not be applied to the element from replacements.

In this example *layout* properties of all "Name" element inheritors are excluded and *remove* and *move* operations are not allowed. This indicates that the "NewName" element will only contain a root *layout* property and all of the "Name" element properties from replacements except *Layout*. Same applies to operations.

The result for the *diff* array builder will be:

```
diff: /**SCHEMA_DIFF*/ [
  {
    // Insert operation.
    "operation": "insert",
    // The name of the parent element in which insert is carried out.
    "parentName": "ProfileContainer",
    // The name of the parent element property with which operation is performed.
    "propertyName": "items",
    // Element new name.
    "name": "NewName",
    // Object with element property values.
    "values": {
      // Bind to a property value or a function
      "bindTo": "Name",
      // Layout.
      "layout": {
        // Column number.
        "column": 0,
        // Row number.
        "row": 0,
        // Number of joined columns.
        "colSpan": 12
      },
    }
  },
] /**SCHEMA_DIFF*/
```

Schema formatting requirements for compatibility with wizards

Beginner Easy **Medium** Advanced

General information

In general, a client schema has three components:

1. Automatically generated code that contains a schema description, its dependencies, localized resources, and messages.
2. Visualization styles (may not be present in certain types of client schemas).
3. Schema code – syntactically correct JavaScript code that defines the module.

Changes made to client schemas using wizards (adding a field, changing the tab position, adding a detail or a module to the edit page layout, etc.) are saved by modifying the *diff*, *modules*, *details* and *businessRules* properties of the schema structure. For more information about the schema structure, please see the "[Client view model schemas](#)" article. Due to technical limitations, marker comments for these properties are used to identify them uniquely in the schema code.

Marker comments

Marker comments identify the *diff*, *modules* and *details* properties of the schema structure if it is edited with the help of wizards.

In addition to the basic validation, the checking procedure for client schemas will indicate the schemas without the

necessary comments when a wizard is run. Schema validation rules are given in Table 1.

Table 1. Schema validation rules

Schema type	Required marker comments
View model schema of the <i>EditViewModelSchema</i> edit page	details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*, modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*, diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/ businessRules: /**SCHEMA_BUSINESS_RULES*/{}/**SCHEMA_BUSINESS_RULES*/
View model schema of <i>ModuleViewModelSchema</i> section	modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*, diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
View model schema of <i>EditControlsDetailViewModelSchema</i> detail with edit fields	modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*, diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
View model schema of <i>DetailViewModelSchema</i> detail	modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*, diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
View model schema of <i>GridDetailViewModelSchema</i> detail with editable list	modules: /**SCHEMA_MODULES*/{}/**SCHEMA_MODULES*, diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/

The schema types are determined by the *ClientUnitSchemaType* enumeration.

The declaration rules of the *diff* property

The *diff* property contains an array of configuration objects that are responsible for schema display. The *diff* array may contain objects that configure display of containers, controls, modules, fields and other visual components. For more information about the *diff* array, see the "The "diff" array" article.

Proper use of converters

The *converter* is a function executed in the *viewModel* environment that receives *viewModel*, properties and returns a result of the corresponding type. For the wizards to work correctly, the value of the *diff* property must be in JSON format. Therefore, the converter value must be the name of the view model method, and not the *inline* function.

An example the converter improper use:

```
diff: /**SCHEMA_DIFF*/ [
  {
    //...
    "bindConfig": {
      converter: function(val) {
        // ...
      }
    }
  }
]/**SCHEMA_DIFF*/
```

An example the converter proper use:

```
methods: {
  someFunction: function(val) {
    //...
  }
},
```

```
diff: /**SCHEMA_DIFF*/ [
  {
    //...
    "bindConfig": {
      "converter": "someFunction"
    }
    //...
  }
] /**SCHEMA_DIFF*/
```

Parent element (container)

Parent element is a DOM element into which the module draws its view. For correct work of the wizard, it is necessary that the parent container have only one child element.

An example of incorrect view placement in parent element:

```
<div id="OpportunityPageV2Container" class="schema-wrap one-el" data-item-marker="OpportunityPageV2Container">
  <div id="CardContentWrapper" class="card-content-container page-with-left-el" data-item-marker="EntityLoaded"></div>
  <div id="DuplicateContainer" class="DuplicateContainer"></div>
</div>
```

An example of correct view placement in parent element:

```
<div id="OpportunityPageV2Container" class="schema-wrap one-el" data-item-marker="OpportunityPageV2Container">
  <div id="CardContentWrapper" class="card-content-container page-with-left-el" data-item-marker="EntityLoaded"></div>
</div>
```

When adding, changing, moving an element in the *diff* (the *insert*, *merge*, *move* operations), the *parentName* property (the parent element name) is required.

An example of incorrect view element specification in the *diff* property:

```
{
  "operation": "insert",
  "name": "SomeName",
  "propertyName": "items",
  "values": {}
}
```

An example of correct view element specification in the *diff* property:

```
{
  "operation": "insert",
  "name": "SomeName",
  "propertyName": "items",
  "parentName": "SomeContainer",
  "values": {}
}
```

In case if *parentName* property is missing, at the wizard launch, an error will be displayed, indicating that the page cannot be set up by the wizard.

The *parentName* property value must match the name of the parent element in the corresponding base page schema. For example, for edit pages, it is "CardContentContainer".

The Name uniqueness

Each new *diff* array element must have a unique name.

An example of incorrect adding of elements to the *diff* array:

```
{  
    "operation": "insert",  
    "name": "SomeName",  
    "values": {}  
},  
{  
    "operation": "insert",  
    "name": "SomeName",  
    "values": {}  
}
```

An example of correct adding of elements to the *diff* array:

```
{  
    "operation": "insert",  
    "name": "SomeName",  
    "values": {}  
},  
{  
    "operation": "insert",  
    "name": "SomeSecondName",  
    "values": {}  
}
```

The non-existing parent element

If you specify the name of a non-existing container element as the parent element in the *parentName* property, the "Schema cannot have more than one root object" error will occur, since the added element will be placed in the root container.

The placement of view elements

In order to be able to customize and modify the view elements, they must be located on the markup grid. In the Creatio, each grid row has 24 cells (columns). The *layout* property is used to place elements on the grid.

The grid element properties:

column – left column index

row – upper row index

colSpan – the number of columns occupied

rowSpan – the number of rows occupied

An example of element placement:

```
{  
    "operation": "insert",  
    "parentName": "ParentContainerName",  
    "propertyName": "items",  
    "name": "ItemName",  
    "values": {  
        // Element location.  
        "layout": {  
            // Start with a "0" column.  
            "column": 0,  
            // Place in the 5th row of the grid.  
            "row": 5,  
            // Take 12 columns wide.  
            "colSpan": 12,  
            // Take 1 row height.  
            "rowSpan": 1  
        },  
    },
```

```
"contentType": Terrasoft.ContentType.ENUM
}
}
```

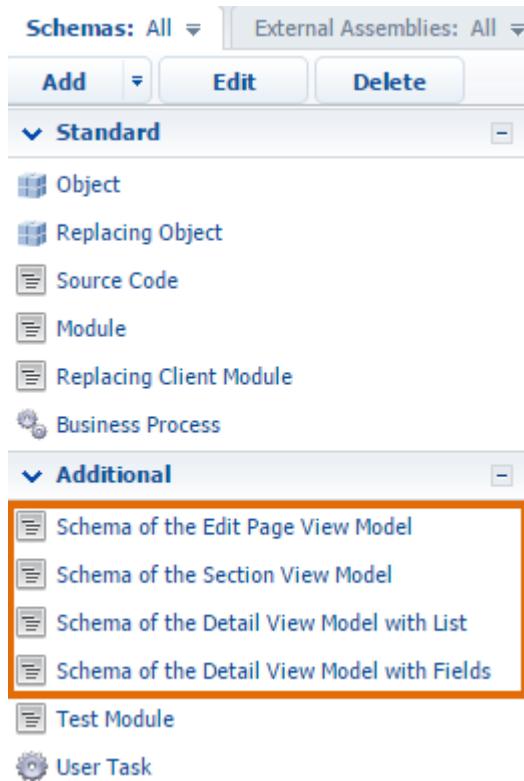
Number of operations

If the client schema is changed without using a wizard, it is recommended to add no more than one operation for one element in the edited schema for the correct operation of the wizard.

Inheritance rules

It is obligatory for the client schema to be a descendant of the *BaseModulePageV2* base schema. It is recommended to create client schemas using the menu commands in the [Configuration] section (Figure 1) or with the help of the wizards.

Fig. 1. The commands for client schemas creation that are compatible with wizards



Specifying an object schema for a client schema

In the client schema, you must fill in the *entitySchemaName* property in which the object (model) schema name must be specified. It is sufficient to specify it in one of the inheritance hierarchy schemas.

An example of the *entitySchemaName* property declaration:

```
define("ClientSchemaName", [], function () {
    return {
        // Object schema (model).
        entitySchemaName: "EntityName",
        //...
    };
});
```

CRUD-operation implementation on client

- The EntitySchemaQuery class. Building of paths to columns
- The EntitySchemaQuery class. Adding columns to a query
- The EntitySchemaQuery class. Getting query result
- The EntitySchemaQuery class. Filters handling

The EntitySchemaQuery class. Building of paths to columns

Beginner

Easy

Medium

Advanced

Building of paths to columns relative to root schema. Examples

The starting point of the EntitySchemaQuery building mechanism is a *root schema* and *feedback principle* (for more details see article **Root schema. Building paths to columns**).

In order to add a column from a table to a query you must build the path to this column. There are different variants for adding columns to queries. Examples of the name formation of columns in each variant are shown below.

1) Root schema column

In this case, the column name is built as **[Column name in root schema]**.

- Root schema: Contact
- Example: column with contact address
- Column name: Address
- Example of creation of the EntitySchemaQuery query that returns values of this column:

Example 1

```
// Let's create [EntitySchemaQuery] class instance with [Contact] root schema.
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
// Add [Address] column then add [Address] alias to it.
esq.addColumn("Address", "Address");
```

2) Schema column, lookup column of current schema refers to

The column name is built on the principle **[Lookup column name].[Schema column name, lookup refers to]**.

Country schema are joined to City root schema by the JOIN operator (LEFT OUTER JOIN by default) in a resultant query. A join condition (On condition of JOIN operator) is formed on the following principle:
[Name of joinable schema].[Id] = [Root schema name].[Name of column that refers to joinable schema + Id]

In common cases you can continue to build a feedback chain.

- Root schema: Contact
- Example: column with account name, column with name of main contact of account
- Column names: Account.Name, Account.PrimaryContact.Name
- Example of creation of EntitySchemaQuery query that returns values of these columns:

Example 2

```
//Let's create [EntitySchemaQuery] class instance with [Contact] root schema.
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
// Add [Account] lookup column.
// Then add [Name] column from [Account] schema,
// to which [Account] lookup column refers, and assign [AccountName] alias to it .
esq.addColumn("Account.Name", "AccountName");
// Add [Account] lookup column.
// Then add [PrimaryContact] lookup column from [Account] schema,
// to which [Account] lookup column refers.
```

```
// Add [Name] column from [Contact] schema,
// to which [PrimaryContact] lookup column refers and assign [PrimaryContactName]
// alias to it.
esq.addColumn("Account.PrimaryContact.Name", "PrimaryContactName");
```

3) Schema column on random external key

Column name is built on the following principle **[Name of _joinable_schema: Name of _column_for_linking_of_joinable_schema:Name of _column_for_linking_of_current_schema]**.

If ID column is used as column for linking in current schema, it can be omitted, i.e. column name will have the following view:

[Name of _joinable _schema:Name of _ column _ for _linking of _joinable _schema].

In general, you can build the column names by the chains of reverse connections of any length.

- Example: column with name of the contact that has added city
- Column name: [Contact:Id:CreatedBy].Name
- Example of creation of EntitySchemaQuery of returning value of this column:

Example 3

```
// Let's create [EntitySchemaQuery] class instance with root schema [Contact].
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
// Add one more [Contact] schema to [Owner] column
// and select [Name] column from it. Assign [OwnerName] alias to it.
esq.addColumn("[Contact:Id:Owner].Name", "OwnerName");
// Join [Contact] schema to [Acount] lookup column on [PrimaryContact] column
// and select [Name] column from it.
// Assign [PrimaryContact] alias to it.
esq.addColumn("Account.[Contact:Id:PrimaryContact].Name", "PrimaryContactName");
```

The EntitySchemaQuery class. Adding columns to a query

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

Terrasoft.EntitySchemaQuery query column is Terrasoft.EntityQueryColumn class instance. You can specify main characteristics of column instance in its properties: title, display value, checkboxes, sorting order and direction etc.

addColumn() method that returns instance of column, added to query, is designed for adding columns to queries. The column name relative to root schema is formed in addColumn() methods in accordance with rules, described in **The EntitySchemaQuery class. Building of paths to columns**. This method has several variants that allow for adding columns with different parameters to a query (table 1).

Table 1. — Method of adding columns to query

Method

addColumn(column,[columnAlias])

Creates and adds Terrasoft.EntityQueryColumn column instance to query column collection.

<i>column</i>	String/Terrasoft.BaseQueryColumn	Is a column adding path (is specified relative to <i>rootSchema</i>) or query column instance Terrasoft.BaseQueryColumn.
---------------	----------------------------------	---

<i>columnAlias</i>	String (optional)	Column alias.
--------------------	-------------------	---------------

addAggregationSchemaColumn(columnPath, aggregationType, [columnAlias], aggregationEvalType)

Creates and adds Terrasoft.FunctionQueryColumn functional column instance with set aggregation type (Terrasoft.FunctionType.AGGREGATION) to query column collection.

<i>columnPath</i>	String	Is a column adding path (it is specified relative to <i>rootSchema</i>).
-------------------	--------	---

<i>aggregationType</i>	Terrasoft.AggregationType	Is a type of used aggregation function.
<i>columnAlias</i>	String (optional)	Is a column alias.
<i>aggregationEvalType</i>	Terrasoft.AggregationEvalType	Is an application field of aggregation function.

Aggregation types (Terrasoft.AggregationType)

<i>AVG</i>	Is an average value of all times.
<i>COUNT</i>	Is a number of all items.
<i>MAX</i>	Is a maximum value among all items.
<i>MIN</i>	Is a minimum value among all items.
<i>NONE</i>	Means that the type of aggregation function is not determined.
<i>SUM</i>	Is a sum of the values of all items.

Application field of aggregation function (Terrasoft.AggregationEvalType)

<i>NONE</i>	Means that application field of aggregation function is not determined.
<i>ALL</i>	Means that this function is applied to all items.
<i>DISTINCT</i>	Means that this function is applied to unique values.

***addParameterColumn*(paramValue, paramDataType, [columnAlias])**

It creates and adds Terrasoft.ParameterQueryColumn parameter column instance to column correlation.

<i>paramValue</i>	Mixed	Is a parameter value. The value should correspond to data type.
<i>paramDataType</i>	Terrasoft.DataValueType	Is a parameter data type.
<i>columnAlias</i>	String (optional)	Is a column alias.

***addFunctionColumn*(columnPath, functionType, [columnAlias])**

It creates and adds Terrasoft.FunctionQueryColumn function column instance to column collection.

<i>columnPath</i>	String	Is a column adding path (it is specified relative to rootSchema).
<i>functionType</i>	Terrasoft.FunctionType	Is a function type.
<i>columnAlias</i>	String (optional)	Is a column alias.

Function type (Terrasoft.FunctionType)

<i>NONE</i>	Means that functional expression type is not determined.
<i>MACROS</i>	Is a macro substitution.
<i>AGGREGATION</i>	Is an aggregation function.
<i>DATE_PART</i>	Is a date part.
<i>LENGTH</i>	Is a length of byte value.

***addDatePartFunctionColumn*(columnPath, datePartType, [columnAlias])**

It creates and adds Terrasoft.FunctionQueryColumn function column instance with [Date Part] type (Terrasoft.FunctionType.DATE_PART) to query column collection.

<i>columnPath</i>	String	Is a column adding path (it is specified relative to rootSchema).
<i>datePartType</i>	Terrasoft.DatePartType	Is a data part, used as a value.
<i>columnAlias</i>	String (optional)	Column alias.

Data part(Terrasoft.DatePartType)

<i>NONE</i>	Is a blank field.
<i>DAY</i>	Is a day.
<i>WEEK</i>	Is a week.
<i>MONTH</i>	Is a month.
<i>YEAR</i>	Is a year.
<i>WEEK_DAY</i>	Is a week day.
<i>HOUR</i>	Is an hour.
<i>HOUR_MINUTE</i>	Is a minute.

addMacrosColumn(macrosType, [columnAlias])

It creates and adds Terrasoft.FunctionQueryColumn function column instance with [Macros] type (Terrasoft.FunctionType.MACROS) that doesn't require parameterization (for example, current month, current user, primary column etc.) to column collection.

<i>macrosType</i>	Terrasoft.QueryMacroType	Is a column macros type.
<i>columnAlias</i>	String (optional)	Is a column alias.

Macros column types (Terrasoft.QueryMacroType)

<i>NONE</i>	Means that macros type is not determined.
<i>CURRENT_USER</i>	Means current user.
<i>CURRENT_USER_CONTACT</i>	Means current user contact.
<i>YESTERDAY</i>	Means yesterday.
<i>TODAY</i>	Means today.
<i>TOMORROW</i>	Means tomorrow.
<i>PREVIOUS_WEEK</i>	Means previous week.
<i>CURRENT_WEEK</i>	Means current week.
<i>NEXT_WEEK</i>	Means next week.
<i>PREVIOUS_MONTH</i>	Means previous month.
<i>CURRENT_MONTH</i>	Means current month.
<i>NEXT_MONTH</i>	Means next month.
<i>PREVIOUS_QUARTER</i>	Means previous quarter.
<i>CURRENT_QUARTER</i>	Means current quarter.
<i>NEXT_QUARTER</i>	Means next quarter.
<i>PREVIOUS_HALF_YEAR</i>	Means previous half year.
<i>CURRENT_HALF_YEAR</i>	Means current half year.
<i>NEXT_HALF_YEAR</i>	Means next half year.
<i>PREVIOUS_YEAR</i>	Means previous year.
<i>CURRENT_YEAR</i>	Means current year.
<i>PREVIOUS_HOUR</i>	Means previous hour.
<i>CURRENT_HOUR</i>	Means current hour.
<i>NEXT_HOUR</i>	Means next hour.
<i>NEXT_YEAR</i>	Means next year.
<i>NEXT_N_DAYS</i>	Means next N days. It requires parameterization.

<i>PREVIOUS_N_DAYS</i>	Means previous N days. It requires parameterization.
<i>NEXT_N_HOURS</i>	Means next N hours. It requires parameterization.
<i>PREVIOUS_N_HOURS</i>	Means previous N hours. It requires parameterization.
<i>PRIMARY_COLUMN</i>	Means primary column.
<i>PRIMARY_DISPLAY_COLUMN</i>	Means primary display column.
<i>PRIMARY_IMAGE_COLUMN</i>	Means primary image column.

addDatePeriodMacrosColumn(macrosType, [macrosValue], [columnAlias])

It creates and adds Terrasoft.FunctionQueryColumn function column instance with [Macros] type (Terrasoft.FunctionType.MACROS) to query column columns. The function adds column with macros type that requires parameterization. For example, next N days, the 3d quarter of the year etc.

<i>macrosType</i>	Terrasoft.QueryMacroType	Is a macros column type.
<i>macrosValue</i>	Number/Date (optional)	Is an auxiliary variable for macros.
<i>columnAlias</i>	String (optional)	Is a column alias.

Examples of addition of columns to query

Example 1. — Adding query column from root schema to query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addColumn("DurationInMinutes", "ActivityDuration");
```

Example 2. — Adding aggregation column query with SUM aggregation type, applied to all table records, to query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addAggregationSchemaColumn("DurationInMinutes", Terrasoft.AggregationType.SUM,
    "ActivitiesDuration", Terrasoft.AggregationEvalType.ALL);
```

Example 3. — Adding aggregation column query with COUNT aggregation type, applied to table unique records, to query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addAggregationSchemaColumn("DurationInMinutes", Terrasoft.AggregationType.COUNT,
    "UniqueActivitiesCount", Terrasoft.AggregationEvalType.DISTINCT);
```

Example 4. — Adding parameter column with TEXT data type to query column type

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addParameterColumn("DurationInMinutes", Terrasoft.DataValueType.TEXT,
    "DurationColumnName");
```

Example 5. — Adding function column with LENGTH function type (value size in bytes) to query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addFunctionColumn("Photo.Data", Terrasoft.FunctionType.LENGTH, "PhotoLength");
```

Example 6. — Adding function column with Date-Part function type (date part) to query column collection. Week day is used as a value

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addDatePartFunctionColumn("StartDate", Terrasoft.DatePartType.WEEK_DAY,
"StartDay");
```

Example 7. — Adding function column with MACROS type that don't require parameterization, i.e. PRIMARY_DISPLAY-COLUMN (Primary Display Column), to query column collection

```
var esq = this.Ext.create(Terrasoft.EntitySchemaQuery, {
    rootSchemaName: "Activity"
});
esq.addMacroColumn(Terrasoft.QueryMacrosType.PRIMARY_DISPLAY_COLUMN,
"PrimaryDisplayColumnName");
```

The EntitySchemaQuery class. Getting query result

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

The EntitySchemaQuery query result is a Creatio property collection. Each instance of a collection is a string of a data set, returnable by query. You can get query results in the following ways:

- Get a definite string of a data set by a primary key through calling the `getEntity` method (example 1).
- Get entire resultant data set by calling `getEntityCollection` method (example 2).

Example 1. — Getting a definite data set string

```
// Get [Id] of card object.
var recordId = this.get("Id");
// Create Terrasoft.EntitySchemaQuery class instance with [Contact] root schema.
var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
// Add column with name of main contact of accounts that refers to given contact.
esq.addColumn("Account.PrimaryContact.Name", "PrimaryContactName");
// Get one record from selection on the basis of [Id] of card object and display it
// in an info window.
esq.getEntity(recordId, function(result) {
    if (!result.success) {
        // error processing/logging, for example
        this.showInformationDialog("Data query error");
        return;
    }
    this.showInformationDialog(result.entity.get("PrimaryContactName"));
}, this);
```

Example 2. — Getting entire data set

```
var message = "";
// Create Terrasoft.EntitySchemaQuery class instance with [Contact] root schema.
var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
// Add column with account name that refers to given account.
esq.addColumn("Account.Name", "AccountName");
// Add column with name of main contact account that refers to given contact.
esq.addColumn("Account.PrimaryContact.Name", "PrimaryContactName");
// Get entire record colelction and display it in an infor window.
esq.getEntityCollection(function (result) {
```

```

if (!result.success) {
    // error processing/logging, for example
    this.showInformationDialog("Data query error");
    return;
}
result.collection.each(function (item) {
    message += "Account name: " + item.get("AccountName") +
        " - primary contact name: " + item.get("PrimaryContactName") + "\n";
});
this.showInformationDialog(message);
}, this);

```

When retrieving the lookup columns, `this.get()` method returns object but not the database record identifier. To access identifier, use the "value" property, for example, `this.get('Account').value`.

Table 1. — Query result getting method

Method

`getEntity(primaryColumnName, callback, scope)`

returns entity instance on set primary key [primaryColumnName]. It calls [callback] function in [scope] context after data receipt.

<code>primaryColumnName</code>	String/Number	Is a primary key value.
<code>callback</code>	Function	Is a function, called upon receipt of server response.
<code>scope</code>	Object	Context where [callback] function will be called.

`getEntityCollection(callback, scope)`

returns collection of entity instances that represent current query results. It calls [callback] function in [scope] context after data receipt.

<code>callback</code>	Function	Is a function that will be called upon receipt of server response.
<code>scope</code>	Object	Is a context where [callback] function will be called.

The EntitySchemaQuery class. Filters handling

Beginner

Easy

Medium

Advanced

A filter is a set of conditions, applied to query data display. According to SQL terms, a filter is a separate predicate (condition) of the WHERE operator.

Creation and application of filters in EntitySchemaQuery

To create simple filter in EntitySchemaQuery use `CreateFilter()` method that returns created `Terrasoft.CompareFilter` filter object. In addition to simple filters, methods for special filter types are implemented in `EntitySchemaQuery` (Table 1).

Table 1. — EntitySchemaQuery methods for creation of filters

Filter creation method

`createFilter(comparisonType, leftColumnPath, rightColumnPath)`

Creates instance of `Terrasoft.comparefilter` class filter for comparing values of two columns.

<code>comparisonType</code>	<code>Terrasoft.ComparisonType</code>	Is a comparison operation type.
<code>leftColumnPath</code>	String	Is a path to verified column relative to root schema <code>rootSchema</code> .
<code>rightColumnPath</code>	String	Is a path to column filter relative to root schema

rootSchema.

createInFilter(leftExpression, rightExpressions)

Creates In-filter instance.

<i>leftExpression</i>	Terrasoft.BaseExpression	Is an expression, verified in a filter.
<i>rightExpressions</i>	Terrasoft.BaseExpression[]	Is an array of expressions that will be compared with left expresion.

createBetweenFilter(leftExpression, rightLessExpression, rightGreaterExpression)

Creates Between-filter instance.

<i>leftExpression</i>	Terrasoft.BaseExpression	Is an expression, verified in a filter.
<i>rightLessExpression</i>	Terrasoft.BaseExpression	Is an initial expression of filtration range.
<i>rightGreaterExpression</i>	Terrasoft.BaseExpression	Is a final expression of filtration range.

createCompareFilter(comparisonType, leftExpression, rightExpression)

Creates Compare-filter instance.

<i>comparisonType</i>	Terrasoft.ComparisonType	Is a type of comparison operation.
<i>leftExpression</i>	Terrasoft.BaseExpression	Is an expression, verified in a filter.
<i>rightExpression</i>	Terrasoft.BaseExpression	Is a filtration expression.

createExistsFilter(columnPath)

Creates Exists-filter instance for comparison of [Exists on set condition] types and sets value of expression of column, located on set path, as verified value.

<i>columnPath</i>	String	Path to column, for the expression of which the filter is built.
-------------------	--------	--

createIsNotNullFilter(leftExpression)

Creates IsNull-filter instance.

<i>leftExpression</i>	Terrasoft.BaseExpression	Is an expression that is verified on IS NOT NULL condition.
-----------------------	--------------------------	---

createIsNullFilter(leftExpression)

Creates IsNull-filter instance.

<i>leftExpression</i>	Terrasoft.BaseExpression	Is an expression that is verified on IS NULL condition.
-----------------------	--------------------------	---

createNotExistsFilter(columnPath)

Creates Exists-filter instance for comparison [Out of set condition] and set expression of the column, located in set path, as verified value.

<i>columnPath</i>	String	Is a path to the verified column, for expression of which the filter is built.
-------------------	--------	--

createColumnFilterWithParameter(comparisonType, columnPath, paramValue)

Creates Compare-filter instance for comparison of the column with set value.

<i>comparisonType</i>	Terrasoft.ComparisonType	Is a type of comparison operation.
<i>columnPath</i>	String	Is a path to the verified column relative to root schema <i>rootSchema</i> .
<i>paramValue</i>	Mixed	Is a parameter value.

createColumnInFilterWithParameters(columnPath, paramValues)

Creates Inofilter instance for verification of coincidence of set column value with the value of one of parameters.

<i>columnPath</i>	String	Is a path to the verified column relative to root schema <i>rootSchema</i> .
<i>paramValues</i>	Array	Is a parameter value array.

createColumnBetweenFilterWithParameters(columnPath, lessParamValue, greaterParamValue)

Creates Between-filter instance that verifies whether the column is within set range.

<i>columnPath</i>	String	Is a path to the verified column relative to root schema <i>rootSchema</i> .
<i>lessParamValue</i>	Mixed	Initial value of the filter.
<i>greaterParamValue</i>	Mixed	Final value of the filter.

createColumnIsNotNullFilter(columnPath)

Creates IsNull-filter for verification of set column.

<i>columnPath</i>	String	Is a path to the verified column relative to root schema <i>rootSchema</i> .
-------------------	--------	--

createColumnIsNullFilter(columnPath)

Creates IsNull-filter instance for verification of set column.

<i>columnPath</i>	String	Is a path to verified column relative to root schema <i>rootSchema</i> .
-------------------	--------	--

createPrimaryDisplayColumnFilterWithParameter(comparisonType, paramValue)

Creates filter object for comparison of primary column for the purpose of displaying with parameter value.

<i>comparisonType</i>	Terrasoft.ComparisonType	Comparison type.
<i>paramValue</i>	Mixed	Parameter value.

The EntitySchemaQuery instance has a filter property that is a collection of filters of a given query. The filter property is the Terrasoft.FilterGroup class instance that, in its turn, is a collection of Terrasoft.BaseFilter items. To add a filter to a query, take the following actions:

- create filter instance for given query (createFilter method () , methods for creation of special type filters);
- add created filter instance or query filters collection (add() method of collection).

All filters added to the Filters collection are interconnected by the AND logical operation. With the LogicalOperation property of the filters collection, the user can specify a logical operation by which filters should be joined. The property takes the following values from Terrasoft.core.enums.LogicalOperatorType list:

- AND
- OR

The possibility for controlling filters, used in building of a resultant data set, is implemented in EntitySchemaQuery. Each filter collection item has the isEnabled property that determines whether this item takes part in building of resultant queries (true means that it takes part and false means that it doesn't take part). Similarly, the isEnabled property is also determined for the entire filter collection. Set this property to false to deactivate filtration for a query. The collection of query filters will remain unchanged. If a query filter collection is created initially, you can use different combinations for filtering queries in the future while not introducing changes directly into the collection.

Example of control of filters in query is shown below (example 1).

Example 1

```
// Creation of query instance with "Contact" root schema.
var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
```

```
});

esq.addColumn("Name");
esq.addColumn("Country.Name", "CountryName");

// Creation of the first filter instance.
var esqFirstFilter =
esq.createColumnFilterWithParameter(Terrasoft.ComparisonType.EQUAL, "Country.Name",
"Mexico");

// Creation of the second filter instance.
var esqSecondFilter =
esq.createColumnFilterWithParameter(Terrasoft.ComparisonType.EQUAL, "Country.Name",
"USA");

// Filters will be updated by OR logical operator in query filters collection.
esq.filters.logicalOperation = Terrasoft.LogicalOperatorType.OR;

// Adding created filters to collection.
esq.filters.add("esqFirstFilter", esqFirstFilter);
esq.filters.add("esqSecondFilter", esqSecondFilter);

// This collection will include objects, i.e. query results, filtered by two filters.
esq.getEntityCollection(function (result) {
    if (result.success) {
        result.collection.each(function (item) {
            // Processing element collection.
        });
    }
}, this);

// It is indicated that the second filter will be used in building of resultant
query.
// This filter is not deleted from query filters collection.
esqSecondFilter.isEnabled = false;

// This collection will include objects, i.e. query results, filtered only by the
first filter.
esq.getEntityCollection(function (result) {
    if (result.success) {
        result.collection.each(function (item) {
            // Processing of collection items.
        });
    }
}, this);
```

Column paths are built in the EntitySchemaQuery filters in accordance with common rules for building paths to columns relative to root schema (described in article **The EntitySchemaQuery class. Building of paths to columns**).

Examples of the use of other methods for creating filters are represented below.

Example 2

```
// Creation of query instance with "Contact" root schema.
var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
    rootSchemaName: "Contact"
});
esq.addColumn("Name");
esq.addColumn("Country.Name", "CountryName");

// Select all contacts where country is not specified.
var esqFirstFilter = esq.createColumnIsNullFilter("Country");
```

```
// Select all contacts, date of birth of which fall at the period from 1.01.1970 to
1.01.1980.
var dateFrom = new Date(1970, 0, 1, 0, 0, 0, 0);
var dateTo = new Date(1980, 0, 1, 0, 0, 0, 0);
var esqSecondFilter = esq.createColumnBetweenFilterWithParameters("BirthDate",
dateFrom, dateTo);

// Add created filters to query collection.
esq.filters.add("esqFirstFilter", esqFirstFilter);
esq.filters.add("esqSecondFilter", esqSecondFilter);

// This collection will include objects, i.e. query results, filtered by two filters.
esq.getEntityCollection(function (result) {
    if (result.success) {
        result.collection.each(function (item) {
            // Processing of collection items.
        });
    }
}, this);
```

WebSocket messages transferring mechanism. ClientMessageBridge

Contents

- [ClientMessageBridge. Message history save mechanism](#)
- [ClientMessageBridge. API description](#)
- [ClientMessageBridge. The client-side WebSocket message handler](#)

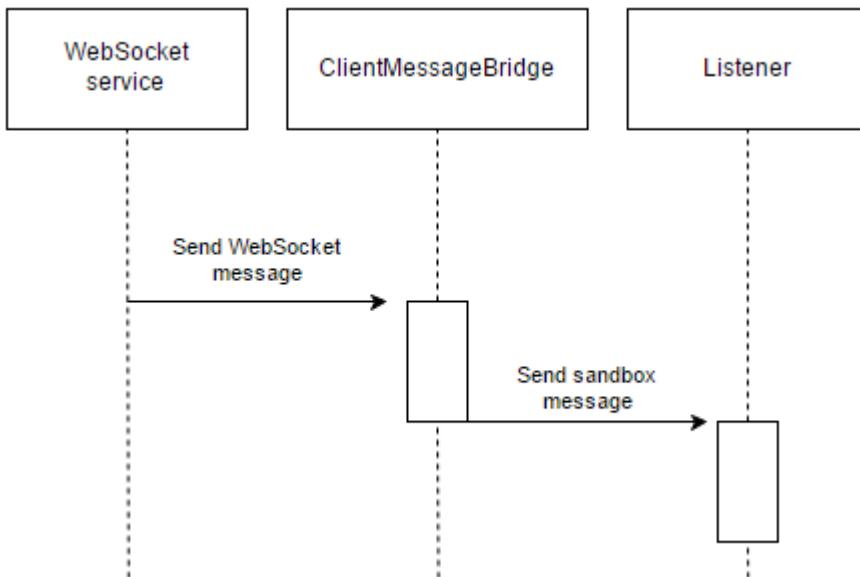
ClientMessageBridge. Message history save mechanism

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

General information

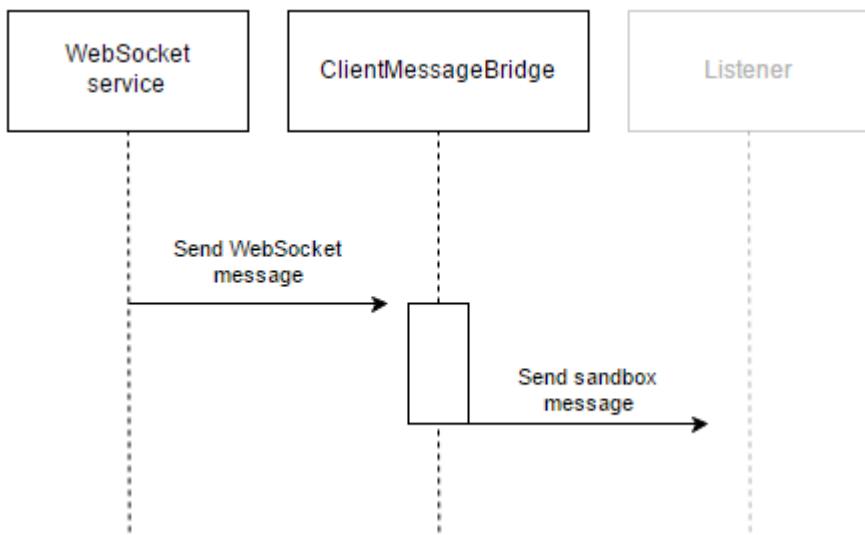
In a best-case scenario a handler is located within the system at the time of publication of the message (Fig. 1).

Fig. 1. Perfect interaction mechanism



However, there may be situations when a handler is not yet loaded (Fig. 2).

Fig. 2. The process of interaction with an absent handler



In order to not lose unprocessed messages, you can wait for a listener before publication. If a listener is absent, the messages are saved in history. Before publication, each message is checked for a listener. When the listener is loaded, all stored messages are published in the order they were received. After the messages have been published, the history is cleared.

Configuring the processing of messages stored in the history example

To implement the described feature, you have to set the `isSaveHistory` checkbox to `true`.

```

init: function() {
    // Calling the init() parent method.
    this.callParent(arguments);
    // Adding a new configuration object to the configuration object collection.
    this.addMessageConfig({
        // sender – name of the message received via WebSocket.
        sender: "OrderStepCalculated",
        // Name of the message sent within the system.
        messageName: "OrderStepCalculated",
        // isSaveHistory – checkbox indicating that the history must be saved.
        isSaveHistory: true
    });
}

```

History saving mechanism

The `ClientMessageBridge` class is the heir of the `BaseMessageBridge` abstract class, which contains abstract methods (`saveMessageToHistory`, `getMessagesFromHistory`, `deleteSavedMessages`). In `ClientMessageBridge` message history saving is implemented with the use of the `localStorage` of the browser, and the implementation of abstract methods enables you to manipulate data in storage. To work with the `localStorage` class, use the `Terrasoft.LocalStore` class.

Methods:

- `saveMessageToHistory` – ensures that new messages are saved in the message collection
- `getMessagesFromHistory` – provides an array of messages based on the transferred name
- `deleteSavedMessages` – deletes all messages based on the transferred name

If there is a need to implement another type of storage, you must create an heir class of the `BaseMessageBridge` class and implement all abstract methods (`saveMessageToHistory`, `getMessagesFromHistory`, `deleteSavedMessages`).

ClientMessageBridge. API description

Beginner

Easy

Medium

Advanced

Properties

WebSocketMessageConfigs: Array

Collection of configuration objects.

LocalStoreName: String

Name of the repository, where the message history is stored.

LocalStore: Terrasoft.LocalStore

An instance of class that implements access to local repository.

Methods

init()

Initializes default values.

getSandboxMessageListenerExists(sandboxMessageName)

Checks for available listeners of message with passed name.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

Returned value:

Boolean – the result of checking for available message listeners

publishMessageResult(sandboxMessageName, webSocketMessage)

Publishes the message within the system.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

webSocketMessage: Object – message received by WebSocket.

Returned value:

* – result obtained from the message handler.

beforePublishMessage(sandboxMessageName, webSocketBody, publishConfig)

Handler that is called before publishing the message within the system.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

webSocketBody: Object – message received by WebSocket.

publishConfig: Object – configuration object of message broadcast.

afterPublishMessage(sandboxMessageName, webSocketBody, result, publishConfig)

Handler that is called after publishing the message within the system.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

webSocketBody: Object – message received by WebSocket. *result: * – result of publishing the message within the system.*

publishConfig: Object – configuration object of message broadcast.

addMessageConfig(config)

Adds a new configuration object to a collection of configuration objects.

Parameters:

config: Object – configuration object.

Configuration object parameter:

```
{  
    "sender": "webSocket sender key 1",  
    "messageName": "sandbox message name 1",  
    "isSaveHistory": true  
}
```

Where:

- *sender: String* – name of the message that is expected from WebSocket.
- *messageName: String* – message name that will be used when sending the message within the system.
- *isSaveHistory: Boolean* – determines whether message history must be saved.

saveMessageToHistory(sandboxMessageName, webSocketBody)

Saves message in the repository if there is no subscriber and the save checkbox is selected in the configuration object.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

webSocketBody: Object – message received by WebSocket.

getMessagesFromHistory(sandboxMessageName)

Gets an array of saved messages from the repository.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

deleteSavedMessages(sandboxMessageName)

Deletes saved messages from the repository.

Parameters:

sandboxMessageName: String – message name that will be used when sending the message within the system.

ClientMessageBridge. The client-side WebSocket message handler

Beginner

Easy

Medium

Advanced

General information

The ClientMessageBridge schema is used to broadcast messages received via WebSocket. If additional logic in the extending ClientMessageBridge schemas wasn't specified for each message received via WebSocket, a broadcast is used to send messages within the system through the SocketMessageReceived sandbox. After subscribing to a message, you can easily process the data received through WebSocket.

For additional message handling, it is necessary to implement an extending schema before publishing and changing the message name. Using the available API, you can configure specific message types in the extending schema.

Setting up a new subscriber example

Case description

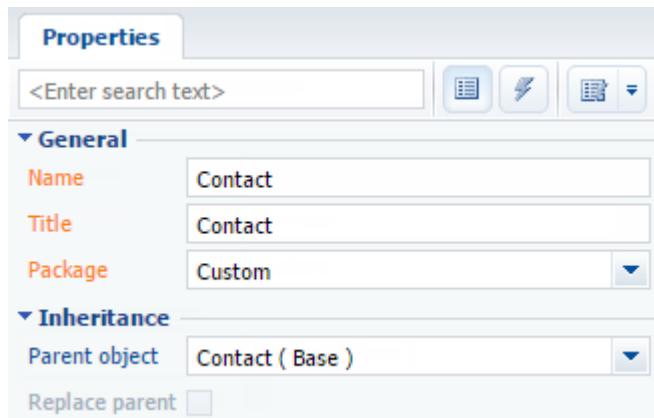
When a contact is saved, you need to publish a message with the *NewUserSet* name that contains information about the contact's birth date and name on the server side. On the client side, you must implement the *NewUserSet* messaging within the system. Additionally, before messaging, you must process the *birthday* message property received through WebSocket, and you must invoke the *afterPublishUserBirthday* utility class method after messaging. Finally, you need to implement the subscription to a message sent on the client side, for example, in the schema of the contact edit page.

Case implementation algorithm

1. Create the replacing [Contact] object

Before adding message publishing via WebSocket, you have to create a replacing object and set the [Contact] as a parent (Fig. 1).

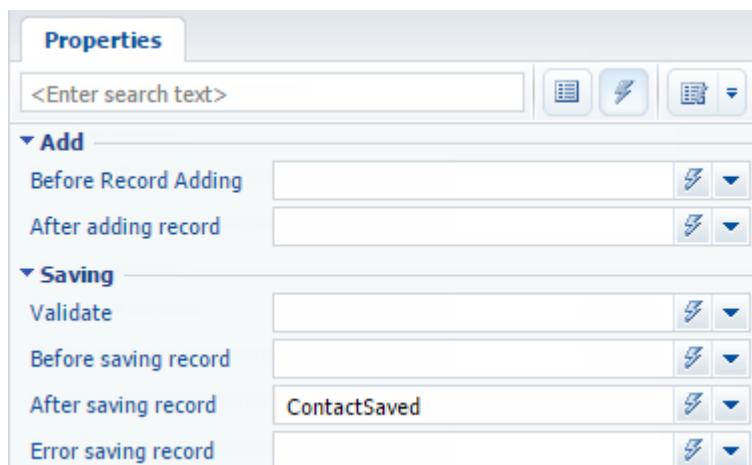
Fig. 1. Creating the replacing [Contact] object



2. Create the "Record saved" event

Next, you need to add message publishing via WebSocket after the contact record has been saved. To do this, go to the tab with the object events (Fig. 2, 1) and click the "Record saved" button (Fig. 2, 2).

Fig. 2. Creating the "Record saved" event



3. Implement event subprocess in the "Record saved" event

In the Record Saved event handler, implement the event subprocess which is run by the ContactSaved message. To do this:

- Add an event subprocess element (Fig. 3, 1);
- Add a message element (Fig. 3, 2), setting ContactSaved as the message name (Fig. 4);

- Add a script element (Fig. 3, 3);
- Connect the message object and script (Fig. 3, 4).

Fig. 3. Creating message handler subprocess

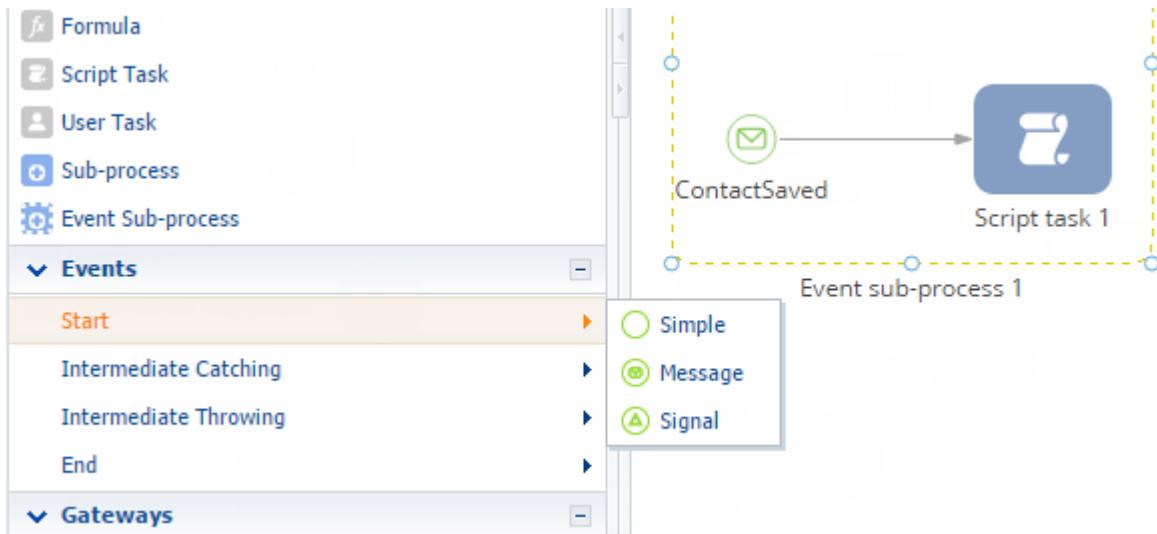
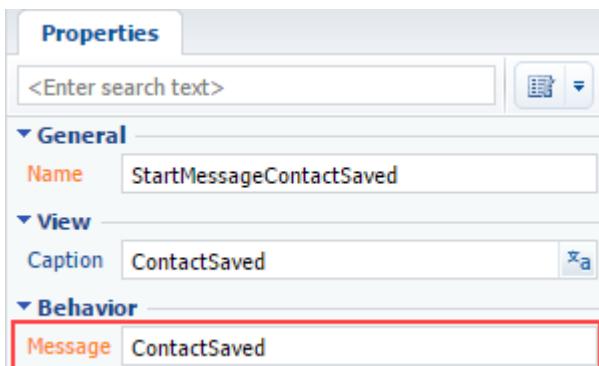


Fig. 4. Initial message properties



4. Add message publication logic through WebSocket

To do this, double-click to open the [Script-task] event subprocess and add the following source code:

```
// Receiving contact name
string userName = Entity.GetTypedColumnValue<string>("Name");
// Receiving contact birth date.
DateTime birthDate = Entity.GetTypedColumnValue<DateTime>("BirthDate");
// Forming message text.
string messageText = "{\"birthday\": \"\" + birthDate.ToString("s") + "\", \"name\":";
messageText += userName + "}";
// Setting message name.
string sender = "NewUserSet";
// Publishing message through WebSocket.
MsgChannelUtilities.PostMessageToAll(sender, messageText);
return true;
```

After that, save and close the tab containing the [Script-task] element source code, and then save and publish the whole event subprocess.

5. Implement message sending inside the application

To do this, create a replacing client module in a custom package(Fig. 5) and set the ClientMessageBridge of the NUI package as a parent object (Fig. 6).

Fig. 5. Creating a replacing client module

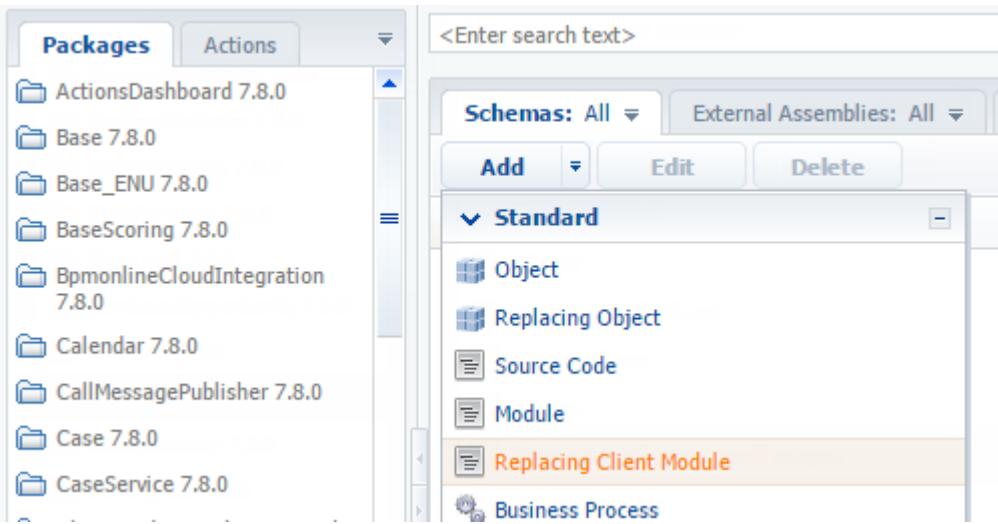
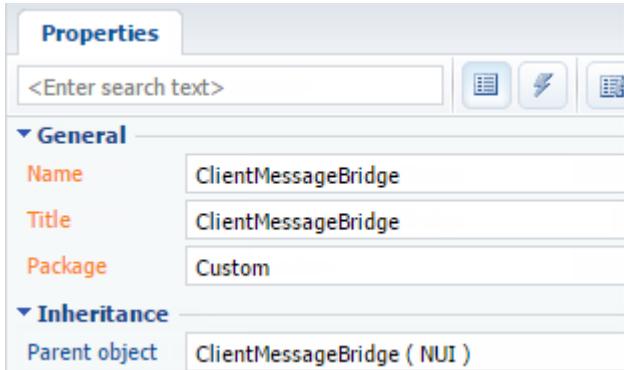


Fig. 6. Client module properties.



Read more about replacing client modules in the "[Creating a custom client module schema](#)" article.

To implement the distribution of messages NewUserSet within the system, it is necessary to add the following source code in the schema:

```
define("ClientMessageBridge", ["ConfigurationConstants"],
    function(ConfigurationConstants) {
        return {
            // Messages.
            messages: {
                // Message name.
                "NewUserSet": {
                    // Message type – broadcasting, without a specific subscriber.
                    "mode": Terrasoft.MessageMode.BROADCAST,
                    // Message direction – publication.
                    "direction": Terrasoft.MessageDirectionType.PUBLISH
                }
            },
            methods: {
                // Schema initialization.
                init: function() {
                    // Parent method calling
                    this.callParent(arguments);
                    // Adding new configuration object to the configuration object
                    collection.
                    this.addMessageConfig({
                        // Name of the message received via WebSocket.
                        sender: "NewUserSet",
                        // Name of the message sent within the system.

```

```

        messageName: "NewUserSet"
    );
},
// Method executes after the message publication.
afterPublishMessage: function(
    // Name of the message sent within the system.
    sandboxMessageName,
    // Message contents.
    webSocketBody,
    // Message result
    result,
    // Message configuration object.
    publishConfig) {
    // Check whether the message matches the one added to the
configuration object.
    if (sandboxMessageName === "NewUserSet") {
        // Saving the content to local variables.
        var birthday = webSocketBody.birthday;
        var name = webSocketBody.name;
        // Displaying content in the console.
        window.console.info("Published message: " +
sandboxMessageName +
            ". Data: name: " + name +
            "; birthday: " + birthday);
    }
}
);
});
});

```

Go to the *messages* section and bind the *NewUserSet* broadcast message, which can only be published within the system. Go to the the *methods* section and restart the *Init* parent method to add the messages received via WebSocket in the configurational schema message object. To track messaging launch time, reload the *afterPublishMessage* parent method.

After the schema has been saved and the application page has been refreshed, the *NewUserSet* messages received via WebSocket will be sent within the system. Read more about debugging in the browser in the "**Client code debugging**" article.

6. Implement message subscription

To obtain an object transmitted via WebSocket, you must subscribe to *NewUserSet* messages in any scheme, for example, "Page contact V2". To do this, you need to create a replacing client module (see section 5), specifying the "Contact page display schema" as a parent object. To do this, add the following source code:

```

define("ContactPageV2", [],
function(BusinessRuleModule, ConfigurationConstants) {
    return {
        //entitySchemaName: "Contact",
        messages: {
            // Message name.
            "NewUserSet": {
                // Message type – broadcasting, without a specific subscriber.
                "mode": Terrasoft.MessageMode.BROADCAST,
                // Message direction – subscription.
                "direction": Terrasoft.MessageDirectionType.SUBSCRIBE
            }
        },
        methods: {
            // Schema initialization.
            init: function() {
                // Init() parent method calling.
            }
        }
    }
});

```

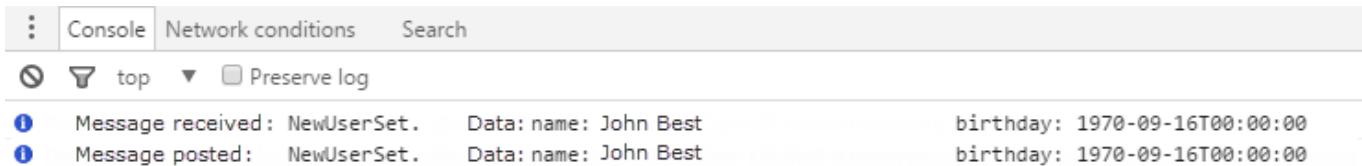
```
        this.callParent(arguments);
        // Subscription to receiving the NewUserSet message.
        this.sandbox.subscribe("NewUserSet", this.onNewUserSet, this);
    },
    // Receiving NewUserSet message event handler.
    onNewUserSet: function(args) {
        // Saving the message content to local variables.
        var birthday = args.birthday;
        var name = args.name;
        // Displaying content in the console.
        window.console.info("Message received: NewUserSet. Data: name: "
+
            name + "; birthday: " + birthday);
    }
}
});
```

Go the messages section and bind the `NewUserSet` broadcast message, which can only be published within the system. Go to the `methods` section and restart the `Init` parent method to subscribe to the `NewUserSet` message and indicate the `onNewUserSet`, method-handler that processes in the message and displays the result in the browser console.

After you have added the source code, you must save the schema and refresh the application page in the browser.

The result of the case is two informational messages in the browser console after you save the contact (Fig. 7).

Fig. 7. Browser console



Frequently used client-side classes

Contents

- The **DataManager** class
 - The **SourceCodeEditMixin** class

The DataManager class

Beginner **Easy** **Medium** **Advanced**

Introduction

Sometimes it may be necessary to create, modify and delete entity data without saving these changes to the database in the process of working with the client part of the application. Saving changes to the database must take place when the `Save` method is explicitly called. These functions are implemented in the *DataManager* and *DataManagerItem* classes.

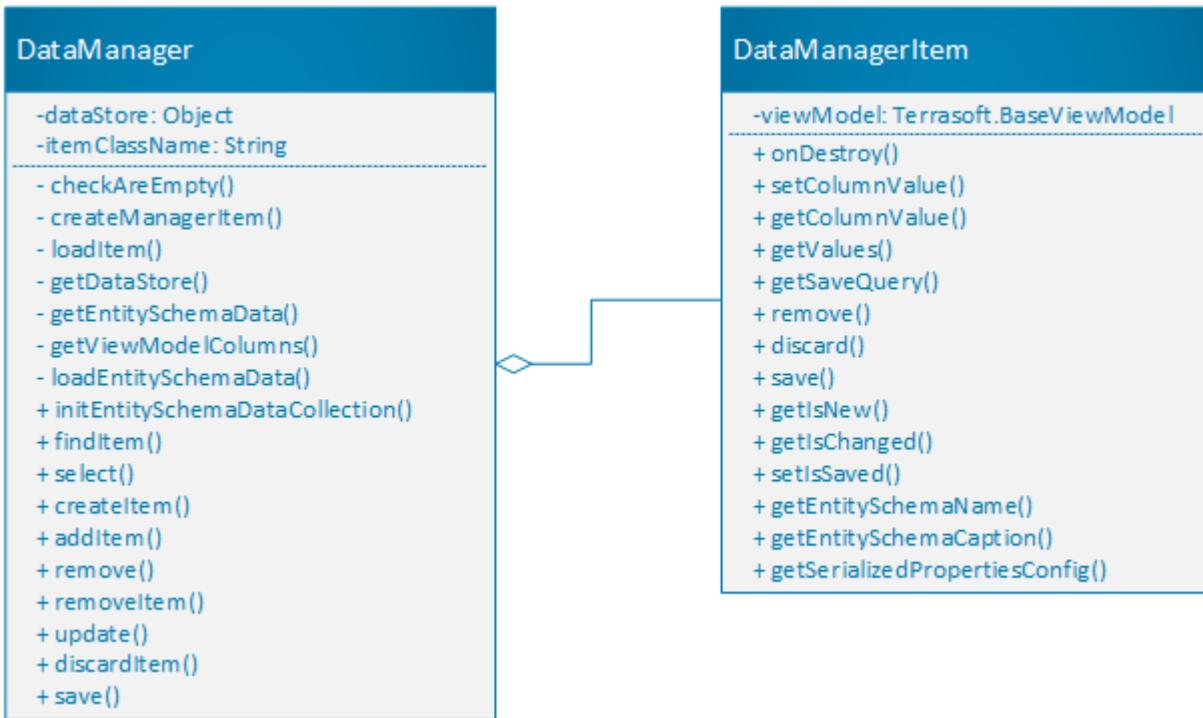
The *DataManager* class is a singleton available through the *Terrasoft* global object. This class provides the *dataStore* repository. The contents of one or more database tables can be loaded into the repository. Example:

```
dataStore: {  
  SysModule: sysModuleCollection,  
  SysModuleEntity: sysModuleEntityCollection  
}
```

sysModuleCollection and *sysModuleEntityCollection* are the data collections of the *DataManagerItem* type of the *SysModule* and *SysModuleEntity* schemas. Each collection record is a record of the corresponding database table.

DataManager and *DataManagerItem* class diagram is available on Fig. 1.

Fig. 1 Class diagram



The “Terrasoft.manager.DataManager” class

Properties

Table 1. Primary properties of the *DataManager* class

dataStore
Object

The data collection repository.

itemClassName
String

Name of the record class. Has the *Terrasoft.DataManagerItem* value.

Methods

Table 2. Primary methods of the *DataManager* class

select

If there is no data with the *config.entitySchemaName* name in the *dataStore*, then the method forms and executes the request to the database and returns the received data, or the method will return the data collection from the *dataStore*.

Parameters:

- *config {Object}* – configuration object;
- *callback {Function}* – callback function;
- *scope {Object}* – the callback function context.

createItem

Creates a new record of the *config.entitySchemaName* type with the *config.columnValues* column values.

Parameters:

- *config {Object}* – configuration object;
- *callback {Function}* – callback function;
- *scope {Object}* – the callback function context.

addItem

Adds the *item* record to the schema data collection.

Параметры:

- *item {Terrasoft.DataManagerItem}* – added record.

findItem

Returns the record of the schema data collection with the *entitySchemaName* name and *id* Id.

Parameters:

- *entitySchemaName {String}* – data collection name;
- *id {String}* – record Id.

remove

Sets the *isDeleted* flag for the *item* record. The record will be deleted from the database after saving the changes.

Parameters:

- *item {Terrasoft.DataManagerItem}* – deleted record.

removeItem

Deletes the record from the schema data collection.

Parameters:

- *item {Terrasoft.DataManagerItem}* – deleted record.

update

Updates the record with the *config.primaryColumnName* primary column value by the *config.columnValues* values.

Parameters:

- *config {Object}* – configuration object;
- *callback {Function}* – callback function;
- *scope {Object}* – the callback function context.

discardItem

Cancels changes for the *item* record made in current working session with the *DataManger* object.

Parameters:

- *item {Terrasoft.DataManagerItem}* – a record with the canceled changes.

save

Saves the schema data collections specified in the *config.entitySchemaNames* to the database.

Parameters:

- *config {Object}* – configuration object;
- *callback {Function}* – callback function;
- *scope {Object}* – the callback function context.

The “Terrasoft.DataManagerItem” class

Properties

Table 3. Primary properties of the *DataManagerItem* class

viewModel

Terrasoft.BaseViewMode

Object projection of the record in the database.

Methods

Table 4. Primary methods of the *DataManagerItem* class

setColumnValue

Sets the new *columnName* value for the *columnName* column.

Parameters:

- *columnName {String}* – column name;
- *columnValue {String}* – column value.

getColumnValue

Returns the value of the *columnName* column.

Parameters:

- *columnName {String}* – column name.

getValues

Returns values of all record columns.

remove

Sets *isDeleted* flag to the record.

discard

Cancels changes for the record made in current working session with the *DataManager* object.

save

Saves changes in the database.

getIsNew

Returns the flag that the record is new.

getIsChanged

Returns the flag that the record was modified.

Examples

Getting records from the [Contact] table:

```
// Definition of the configuration object.  
var config = {  
    //Entity Schema Name.  
    entitySchemaName: "Contact",  
    // Remove duplicates in the resulting dataset.  
    isDistinct: true  
};  
// Receiving data.  
Terrasoft.DataManager.select(config, function (collection) {  
    // Saving received records to local storage.  
    collection.each(function (item) {  
        Terrasoft.DataManager.addItem(item);  
    });  
}, this);
```

Adding new record to the *DataManager* object:

```
// Definition of the configuration object.  
var config = {  
    // Entity Schema Name.  
    entitySchemaName: "Contact",  
    // Column values.  
    columnValues: {  
        Id: "00000000-0000-0000-0000-000000000001",  
        Name: "Name1"  
    }  
};  
// Creating a new record.  
Terrasoft.DataManager.createItem(config, function (item) {  
    Terrasoft.DataManager.addItem(item);  
, this);
```

Getting the record and changing the column value:

```
// Getting a record.  
var item = Terrasoft.DataManager.findItem("Contact",  
    "00000000-0000-0000-000000000001");  
// Setting a new value for "Name2" to the [Name] column.  
item.setColumnValue("Name", "Name2");
```

Deleting the record from the *DataManager* object:

```
// Definition of the configuration object.  
var config = {  
    // Entity Schema Name.  
    entitySchemaName: "Contact",  
    // Primary column value.  
    primaryColumnValue: "00000000-0000-0000-000000000001"  
};  
// Sets the isDeleted attribute for item.  
Terrasoft.DataManager.remove(config, function () {  
, this);
```

Cancels changes made in current working session with the *DataManager* object.

```
// Getting a record.  
var item = Terrasoft.DataManager.findItem("Contact",  
    "00000000-0000-0000-000000000001");  
// Undo changes for writing.  
Terrasoft.DataManager.discardItem(item);
```

Saves changes in the database.

```
// Definition of the configuration object.  
var config = {  
    // Entity Schema Name.  
    entitySchemaNames: ["Contact"]  
};  
// Saving changes to the database.  
Terrasoft.DataManager.save(config, function () {  
, this);
```

The SourceCodeEditMixin class

Beginner

Easy

Medium

Advanced

Introduction

When developing controls, it may be necessary to implement the string value editing functionality with the HTML, JavaScript, or LESS code. The *SourceCodeEditMixin* class was created for that.

SourceCodeEditMixin is a **mixin**, which provides a user-friendly string editing interface for class enrichment. Its concept resembles that of multiple inheritance.

The “Terrasoft.SourceCodeEditMixin” class

Properties

Table 1. Primary properties of the *SourceCodeEditMixin* class

sourceCodeEdit
Terrasoft.SourceCodeEdit

An instance of the source code editor control.

sourceCodeEditContainer
Terrasoft.Container

An instance of the container with the source code editor.

Methods

Table 2. Primary methods of the *SourceCodeEditMixin* class

openSourceCodeEditModalBox

A method that implements the opening of a window for editing source code.

loadSourceCodeValue

Abstract method. Must be implemented in the main class. Implements the logic of obtaining the value to edit.

saveSourceCodeValue

Abstract method. Must be implemented in the main class. Implements the logic of saving editing results in the main class object.

Parameters:

- *value {String}*

destroySourceCodeEdit

A method that implements the purification of mixin resources.

getSourceCodeEditModalBoxStyleConfig

Returns a key-value object that describes styles in the modal window for editing source code.

getSourceCodeEditStyleConfig

Returns a key-value object that describes styles applied to source editor controls.

getSourceCodeEditConfig

Returns a key-value object that describes the properties that the created instance of the source code editor controls will have.

Main properties of the created instance of source code editor controls (see *getSourceCodeEditConfig* method) are listed in table 3.

Table 3. Created source editor properties

showWhitespaces
Boolean

Displaying invisible strings Default value: *false*.

language

SourceCodeEditEnums.Language

Language syntax. Selected from the *SourceCodeEditEnums.Language* enumeration (Table 4). Default value: *SourceCodeEditEnums.Language.JAVASCRIPT*.

theme

SourceCodeEditEnums.Theme

Editor theme Selected from the *SourceCodeEditEnums.Theme* enumeration (Table 5). Default value: *SourceCodeEditEnums.Theme.CRIMSON_EDITOR*.

showLineNumbers

Boolean

Display string numbers. Default value: *true*.

showGutter

Boolean

Set the gap between columns. Default value: *true*.

highlightActiveLine

Boolean

Highlighting the active line. Default value: *true*.

highlightGutterLine

Boolean

Highlighting of the inter-column space line. Default value: *true*.

Table 4. Language syntax of the source code editor (*SourceCodeEditEnums.Language*)

Enumeration member	Programming language
JAVASCRIPT	JavaScript
CSHARP	C#
LESS	LESS
CSS	CSS
SQL	SQL
HTML	HTML

Table 5. - Source code editor themes (*SourceCodeEditEnums.Theme*)

Enumeration member	Subject
SQLSERVER	SQL editor subject
CRIMSON_EDITOR	Crimson editor subject

Use case

Add a mixin to the *mixins* property to use it in a control:

```
// Connecting a mixin.  
mixins: {  
    SourceCodeEditMixin: "Terrasoft.SourceCodeEditMixin"  
},
```

Mixin functionality gets a value by calling the *getSourceCodeValue()* abstract *getter* method. It returns the string for editing. The *getter* method should be implemented each time a mixin is used:

```
// Implementing a field string value.
getSourceCodeValue: function () {
    // The "getValue()" method is implemented in the "Terrasoft.BaseEdit" base class.
    return this.getValue();
},
```

After the editing is completed, the mixin will call the `setSourceCodeValue()` abstract *setter* method to save the result. The *setter* method should be implemented each time a mixin is used:

```
// Method implementation for setting up a result string.
setSourceCodeValue: function (value) {
    // The "setValue()" method is implemented in the "Terrasoft.BaseEdit" base class.
    this.setValue(value);
},
```

Call the `openSourceCodeBox()` method to open the source code editing window. The method is called in the main class instance context. For example, when the `onSourceButtonClick` method of the component is called.

```
// Implementing the process of calling a source editor window method.
onSourceButtonClick: function () {
    this.mixins.SourceCodeEditMixin
        .openSourceCodeBox.call(this);
},
```

After the work with the primary class instance is finished, it is deleted from the memory. The `SourceCodeEditMixin` requires freeing certain resources. To do this, the `destroySourceCode` method is called in the main class instance context.

```
onDestroy: function () {
    this.mixins.SourceCodeEditMixin
        .destroySourceCode.apply(this, arguments);
    this.callParent(arguments);
}
```

Below is the complete source code:

```
// Adding a mixin module to dependencies.
define("SomeControl", ["SomeControlResources", "SourceCodeEditMixin"],
    function (resources) {
        Ext.define("Terrasoft.controls.SomeControl", {
            extend: "Terrasoft.BaseEdit",
            alternateClassName: "Terrasoft.SomeControl",

            // Connecting a mixin.
            mixins: {
                SourceCodeEditMixin: "Terrasoft.SourceCodeEditMixin"
            },

            // Method implementation for obtaining a string value.
            getSourceCodeValue: function () {
                // The "getValue()" method is implemented in the "Terrasoft.BaseEdit" base class.
                return this.getValue();
            },

            // Method implementation for setting up the result string.
            setSourceCodeValue: function (value) {
                // The "setValue()" method is implemented in the "Terrasoft.BaseEdit" base class.
                this.setValue(value);
            }

            // Implementing the process of calling a source editor window method.
```

```
onSourceButtonClick: function () {
    this.mixins.SourceCodeEditMixin
        .openSourceCodeBox.call(this);
},
// Implementing a call to clear mixin resources.
onDestroy: function () {
    this.mixins.SourceCodeEditMixin
        .destroySourceCode.apply(this, arguments);
    this.callParent(arguments);
}
});
});
```

Interface controls

Contents

- **Basic interface controls**
- **Interface control tools**

Basic interface controls

Contents

- **Controls. Introduction**
- **Main menu**
- **Sections**
- **Record edit page**
- **Details**
- **Mini-page**
- **Modal windows**
- **Communication panel**
- **Command line**
- **Action dashboard**
- **Dashboard widgets**
- **The [Timeline] tab**
- **The [Connected entity profile] control**

Controls. Introduction

Beginner Easy **Medium** Advanced

Controls are objects used to create an interface between the user and a Creatio application. They are displayed in navigation panels, dialog boxes and toolbars. Controls include buttons, checkboxes, radio buttons, input fields etc.

All controls are inherited from the *Terrasoft.Component* class (*Terrasoft.controls.Component* in full), which in turn is inherited from *Terrasoft.BaseObject*. Because the *Bindable* mixin is declared in the (*Component*) parent class, it is possible to bind control properties to desired view model properties, methods, or attributes.

For correct operation, the necessary events are declared in the control element. Each element contains events inherited from the *Terrasoft.Component* class:

- *added* – triggered after the component is added to the container.
- *afterrender* – triggered after the component has been rendered and its HTML representation gets in the DOM.
- *afterrerender* – triggered after the component has been rendered and its HTML representation is updated in the DOM.

- *beforererender* – triggered before the component has been rendered and its HTML representation gets in the DOM.
- *destroy* – triggered before the control is deleted.
- *destroyed* – triggered after the control is deleted.
- *init* – triggered when component initialization is complete.

Learn more about *Terrasoft.Component* class events in the "[\('JavaScript API for platform core' in the on-line documentation\)](#)" article.

Controls may subscribe to browser events and determine their own events.

The control is defined in the *diff* modification array of the module where it must be located.

```
// The diff modification array of the module
diff: [
    // Insert operation.
    "operation": "insert",
    // Control parent element.
    "parentName": "CardContentContainer",
    // Name of the parent element property with which the operation is performed.
    "propertyName": "items",
    // Control name.
    "name": "ExampleButton",
    // Control value.
    "values": {
        // Control type.
        "itemType": "Terrasoft.ViewItemType.BUTTON",
        // Control caption.
        "caption": "ExampleButton",
        // Binding the control event to a function.
        "click": {"bindTo": "onExampleButtonClick"},
        // Control style.
        "style": Terrasoft.controls.ButtonEnums.style.GREEN
    }
]
```

The appearance of the control is determined by the (*template <tpl>*) template. Element view is generated according to a specified template during the rendering of the control in the page view.

Controls do not have any business logic. The logic is determined in the module where the control is added.

The control has the *styles* and *selectors* attributes, which are determined in the *Terrasoft.Component* parent class. These attributes provide style customization flexibility.

Main menu

Beginner

Easy

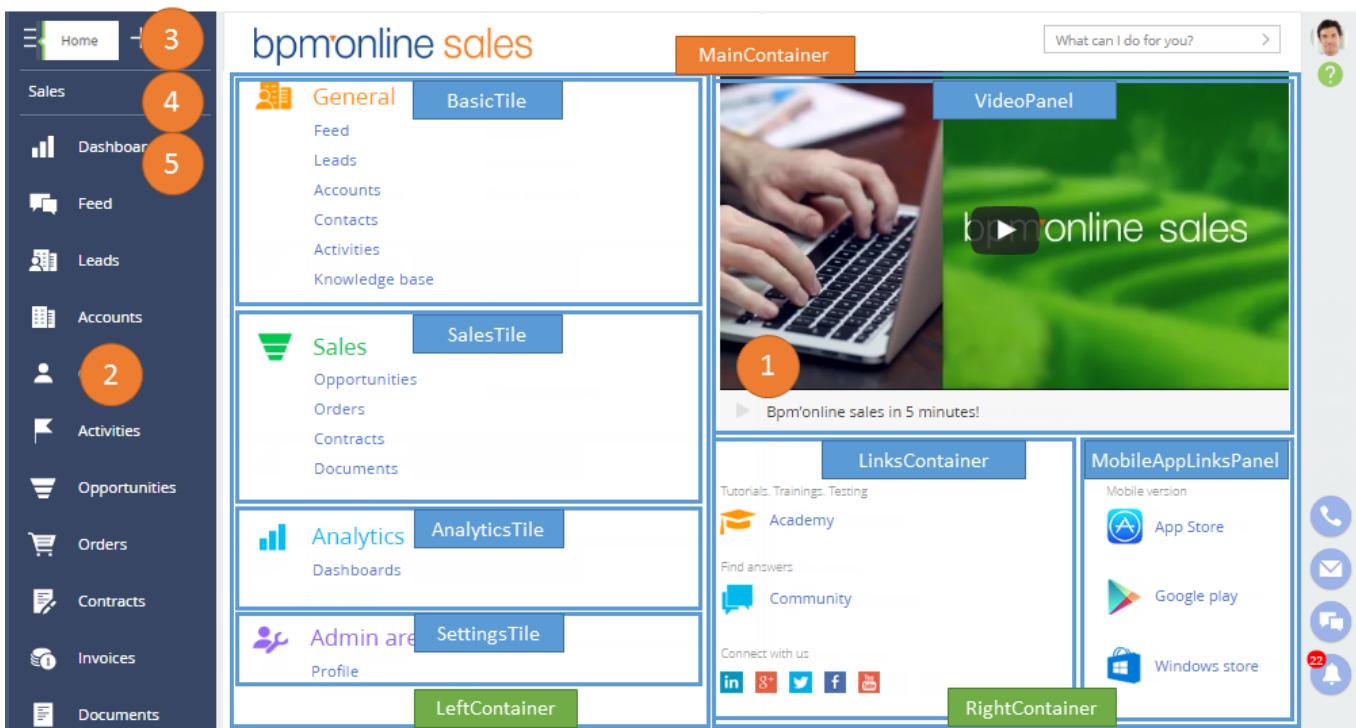
Medium

Advanced

Overview

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

Fig. 1. Main menu



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

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

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

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

Sections

Contents

- **Introduction**
- **Section lists**
- **Section analytics**
- **Section actions**
- **Filters**
- **Tags**

Sections

Beginner

Easy

Medium

Advanced

Overview

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

Fig. 1. The [Contacts] section interface elements

The screenshot shows the 'Contacts' section interface. The sidebar on the left has 'Sales' selected, with 'Accounts' highlighted. The main area shows a list of contacts:

- Contact 1:** Alex Wilson (Account: Alpha Business)
- Contact 2:** Alice Phillips (Account: Streamline Development)
- Contact 3:** Barber Andrew (Account: Infocom)
- Contact 4:** Brenda Lynn (Account: Parsons & Co)

Numbered callouts point to specific UI elements:

- Action button container with 'NEW CONTACT' and 'ACTIONS' dropdown.
- Drop-down list for actions.
- Filter and tag icons.
- Grid data view showing contact details.
- Avatar of Barber Andrew.
- Name of Barber Andrew.
- Job title and contact information for Barber Andrew.

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

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

The screenshot shows the 'Contacts' section interface in the 'Analytics' view. The sidebar on the left has 'Sales' selected, with 'Contacts' highlighted. The main area displays analytics cards and a chart:

- Number of employees:** 13
- New employees:**
 - Richards Sarah (Head of department, 6/26/2011)
 - Peter Moore (Head of department, 6/11/2011)
- Number of contacts:** 63
- Contacts by decision-making role:** Influencer (1)

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

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

- Action buttons container with a section action button (1) and a drop-down list (2)

- Filters container with filters (3) and tags (4)
- Section list data view container (GridView) with action buttons to edit (5), copy (6), and delete (7) the current record
- Section analytics data view container (AnalyticsGridView)

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

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

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

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

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

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

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

Section lists

Beginner

Easy

Medium

Advanced

Overview

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

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

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

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

Contact Name	Account	Job Title	Business phone	Mobile phone	Email
John Best	Our company	Head of department	3030	+44 20 5-549-222	john_best_business@yahoo.com
Murphy Valerie	Our company	Head of department	3090	+44 782 245 8357	valerie.murphy1980@gmail.com
Peter Moore	Our company	Head of department	3040	+44 782 335 8812	peter.moore@yahoo.com
Richards Sarah	Our company	Head of department	3020	+44 782 257 7722	sarah.richards.work@gmail.com

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

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

Contact name	Account	Job title	Business phone	Mobile phone	Email
Murphy Valerie	Our company	Head of department	3090	+44 782 245 8357	valerie.murphy1980@gmail.com
Peter Moore	Our company	Head of department	3040	+44 782 335 8812	peter.moore@yahoo.com
Richards Sarah	Our company	Head of department	3020	+44 782 257 7722	sarah.richards.work@gmail.com
John Best	Our company	Head of department	3030	+44 20 5-549-222	john_best_business@yahoo.com

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

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

Fig. 3. Contact object schema in the objects designer

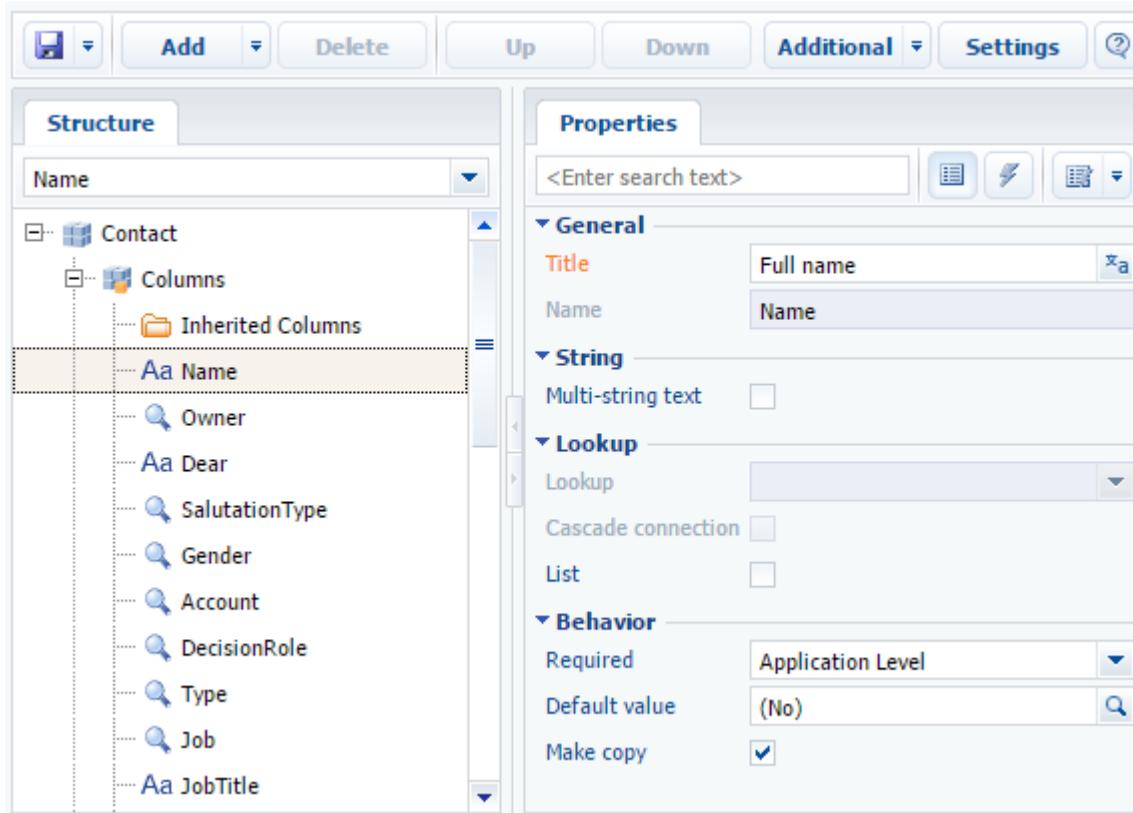


Fig. 4. Contact table

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

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

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

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

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

added columns display in the list. More information about the object wizard can be found in the "[Workspace of the Object Designer](#)".

Section analytics

[Beginner](#)

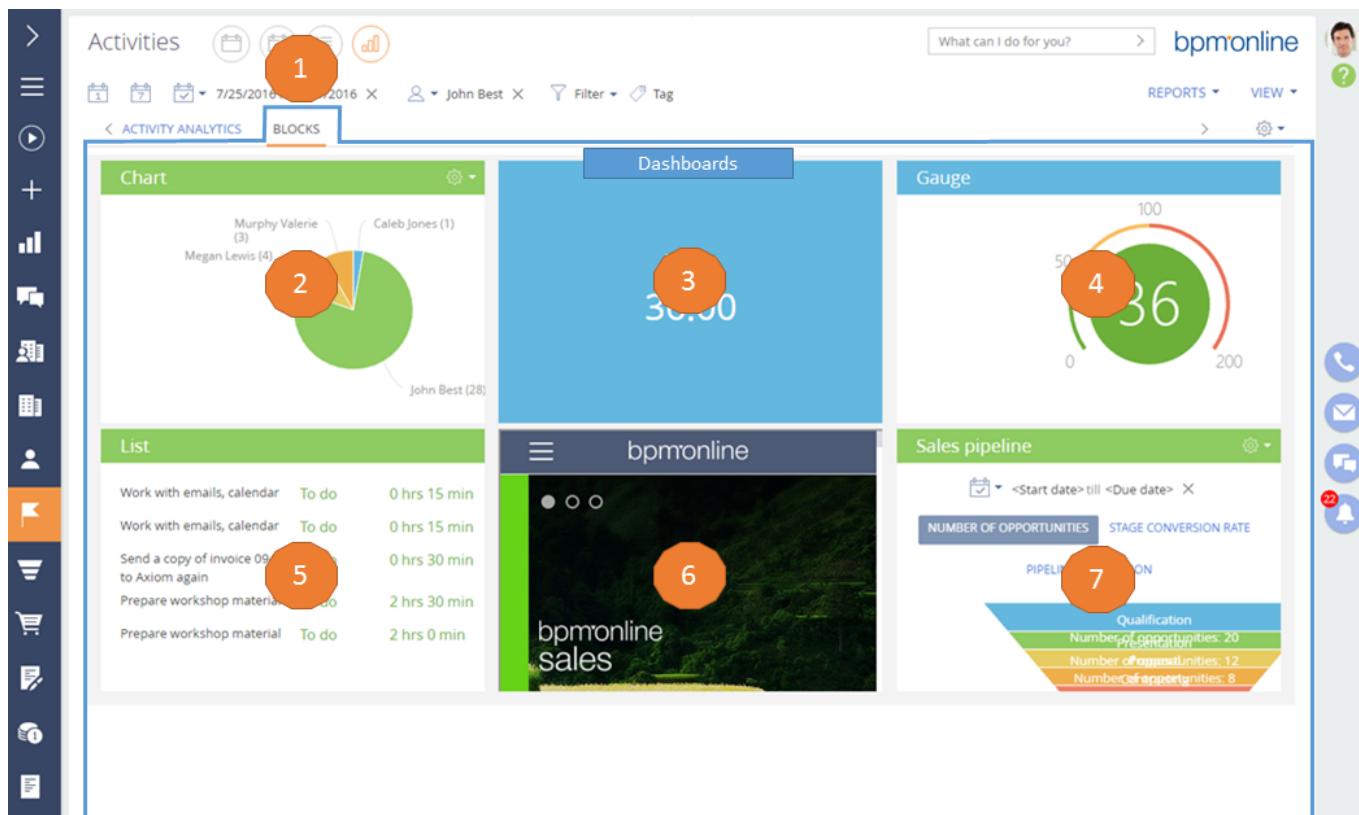
[Easy](#)

[Medium](#)

[Advanced](#)

Overview

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



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

The Creatio application uses the following dashboard blocks (Fig. 1):

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

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

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

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

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

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

The widget dashboard component displays additional custom widgets.

For more information on customizing analytics, please refer to the "["Dashboard components setup"](#) article.

Section actions

Beginner

Easy

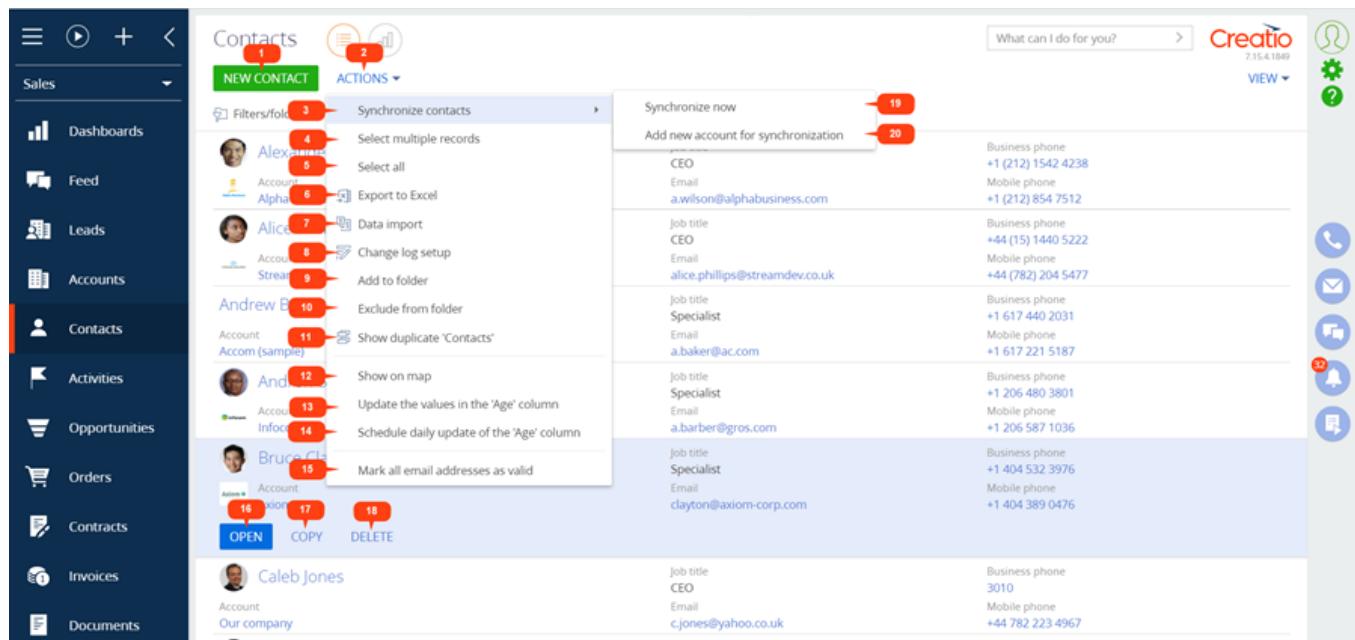
Medium

Advanced

Introduction

The section actions are functional elements representing sets of operations with one or several section records. You can call actions using different buttons, placed both in the container of the current section actions or in the container of the active record (Fig. 1).

Fig. 1. – Actions of the [Contacts] section



The section actions are available (Fig. 1):

- under the call action buttons (1), (16)...(18).
- under the drop-down menu buttons (2), (3).
- under the call action buttons of the menu list (15), (19)...(20).

There are two types of section actions in Creatio: standard and additional.

Standard section actions include:

- [New contact] (1) – opens a pop up window for adding and saving a new section record.
- [Open] (16) – opens the edit page of a selected section record.
- [Copy] (17) – opens a section edit page, copies the data of the selected record and creates a new record upon clicking [Save].
- [Delete] (18) – deletes the current record.
- [Select multiple records] (4) – enables selecting several records in the section list.
- [Select all] (5) – enables selecting all records in the section list.
- [Export to Excel] (6) – exports all list records of the current section in a separate *.xlsx file.
- [Data import] (7) – imports data in Creatio from an *.xlsx file. You can learn more about importing data in the "["Excel data import"](#) article.
- [Change log setup] (8) – opens the page of log management and enables selecting the section columns that should be logged when a record is changed.

Additional actions implement the functionality depending on the section business logic. The following actions are additional for the [Contacts] section in Creatio:

- [Synchronize now] (19) – enables synchronizing Creatio with Google contacts. The action is only performed for records that are specially tagged as per the synchronization settings. You can learn more about the synchronization functionality in the "[How to synchronize Creatio with Google contacts](#)" article.
- [Add new account for synchronization] (20) – synchronization with Google for Creatio Cloud. You can learn more about the synchronization setup in the "[How to set up synchronization from Google for Creatio Cloud](#)" article.
- [Add to folder] (9) – opens a pop up window for selecting a folder to place the current record in.
- [Exclude from folder] (10) – excludes the current record from all folders it belongs to.
- [Show duplicate 'Contacts'] (11) – opens an additional page containing all possible duplicates of the contacts. Creatio adds records to this page automatically after performing the duplicate search.
- [Show on map] (12) – enables displaying the location of selected contacts on the map. The action opens a map window with the selected contacts located. If the address fields of the selected contacts are not populated, the action will not be performed. If the addresses of some contacts are not populated or are populated incorrectly, the window will display the corresponding information.
- [Update the values in the 'Age' column] (13) – updates the [Age] column value of the current contact page.
- [Schedule daily update of the 'Age' column] (14) – opens a pop up window to set up the time for the daily update of the contact age.
- [Mark all email addresses as valid] (15) – sets the [Valid] checkbox for the email address of the selected contact record.

You can create custom actions in Creatio. You can learn more about adding custom actions in the "[Adding an action to the list](#)" block of articles.

Filters

[Beginner](#)

[Easy](#)

[Medium](#)

[Advanced](#)

Overview

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

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

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

The screenshot shows the Creatio interface with the 'Activities' section selected. On the left, there's a sidebar with navigation links: Sales, Dashboards, Feed, Leads, Accounts, Contacts, Activities (which is highlighted in orange), and Opportunities. Above the main content area, there are filter management elements: a 'Quick filter' dashboard with four numbered boxes (1, 2, 3, 4) pointing to specific filter fields, and a 'Standard filter' tab. The main content area displays a list of activities:

Activity Type	Owner	End Date	Account	Status	Category
client material	John Best	7/29/2016 5:00 PM	XT Group	Not started	Paper work
Prepare quotation	John Best	7/29/2016 9:00 PM	Axiom	Not started	To do
Prepare specifications	John Best	7/26/2016 6:30 PM	Clearsoft	Not started	Paper work
Meeting with client	John Best	7/26/2016 8:30 PM	Streamline Development	Completed	Meeting

On the right side, there are icons for a profile picture, a question mark, and a red notification bubble with the number '22'. A search bar at the top right says 'What can I do for you? >' and a 'VIEW' dropdown is also present.

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

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

More detailed information about the filters is available in the "[Quick filter](#)" article.

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

Fig. 2. [Contacts] section filter

John Best	Business phone 3030
Murphy Valerie	Mobile phone +44 20 5-549-222
Peter Moore	Business phone 3090
Sarah Richards	Mobile phone +44 782 245 8357
Peter Moore	Business phone 3040
Sarah Richards	Mobile phone +44 782 257 7722

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

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

Fig. 3. Advanced [Contacts] section filter

John Best	Business phone 3030
Murphy Valerie	Mobile phone +44 20 5-549-222
Peter Moore	Business phone 3040

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

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

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

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

Fig. 4. Filter folder of the [Contacts] section

The screenshot shows the 'Contacts' section of the Creatio application. On the left is a sidebar with navigation links: Sales, Dashboards, Feed, Leads, Accounts, Contacts (which is highlighted in orange), Activities, and Opportunities. The main area has tabs for 'NEW FOLDER', 'NEW CONTACT', and 'ACTIONS'. A 'Filter folders' button is visible. A modal dialog box is open, listing filter categories: Favorites, All, Contact persons, Customers, Employees, Final authority, and Influencers. To the right of the dialog, there is a search bar ('What can I do for you?'), a user profile icon, and a 'bpmonline' logo. Below the search bar, there are filters for 'Account: Our company', 'Job title: Head', and 'Tag'. The main content area displays three contact records:

Contact	Account	Job title	Business phone
John Best	Our company	Head of department	3030
Murphy Valerie	Our company	Head of department	3090
Peter Moore	Our company	Head of department	3040

On the far right, there are icons for phone, email, and messaging.

More detailed information about filter folders is available in the "[Folders](#)" article.

In Creatio you can add user filters. More information about user quick filters is available in the "[Adding quick filter block to a section](#)" article.

Tags

[Beginner](#)

Easy

Medium

Advanced

Overview

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

Fig. 1 The [Contacts] section tags

The screenshot shows the Sales Creatio interface with the 'Contacts' section selected. A modal window titled 'Tags' is open, allowing users to add or remove tags from a contact record. The contact list includes four entries: John Best, Murphy Valerie, Peter Moore, and Richards Sarah, each with their respective details like job title, email, and phone numbers.

Contact	Job Title	Email	Business Phone	Mobile Phone
John Best	Head of department	john_best_business@yahoo.com	3030	+44 20 5-549-222
Murphy Valerie	Head of department	valerie.murphy1980@gmail.com	3090	+44 782 245 8357
Peter Moore	Head of department	peter.moore@yahoo.com	3040	+44 782 335 8812
Richards Sarah	Head of department	sarah.richards.work@gmail.com	3020	+44 782 257 7722

Records are tagged manually. Each section of Sales Creatio has a separate list of tags.

Sales Creatio contains the following tag types:

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

More information about creating and configuring tags can be found in the "[Working with tags](#)" article.

Record edit page

Beginner

Easy

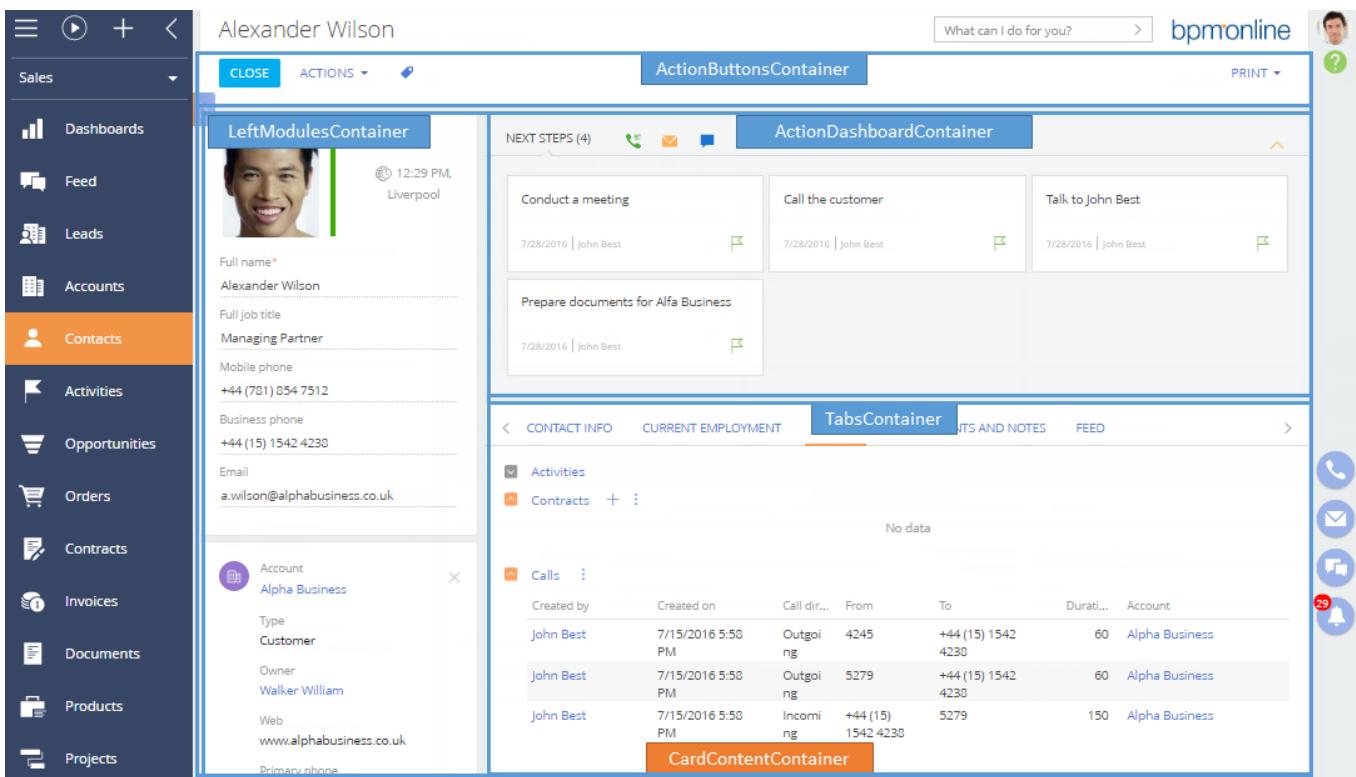
Medium

Advanced

Overview

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

Fig. 1 The [Contacts] section edit page interface



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

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

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

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

Details

Contents

- **Introduction**
- **Details**

Details

Beginner

Easy

Medium

Advanced

Overview

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

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

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

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

The screenshot shows the 'CONTACT INFO' tab selected in the top navigation bar. Below it, two details are displayed:

- Detail with edit fields:** This section contains various input fields for contact communication options. A circled '1' indicates where a new field can be added. The fields include:
 - Business phone: 3030
 - Skype: John Best
 - Email: john.best.business@gmail.com
 - Do not use SMS: checked
 - Do not use mail: checked
 - Twitter: John Best
 - Email: john_best_business@yahoo.com
 - Do not use fax: checked
- Detail with adding page:** This section is for entering addresses. It includes fields for Address type, Address, City, Country, and Index.

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

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

The screenshot shows the 'CONTACTS' tab selected in the left sidebar. A specific detail for 'John Best / Contact address' is open. The page includes a sidebar with navigation links like Sales, Dashboards, Feed, Leads, Accounts, and Contacts. The main area displays the following form fields:

Address type	Business
Country	United States
City	Indiana
Address	12 Bernice St.

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

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

The screenshot shows the 'PRODUCTS' tab selected in the top navigation bar. A detail titled 'Detail with editable list' is displayed, showing a list of products in an order. The table includes columns for Product, Price, Quantity, Unit of measure, Discount, %, and Total. The last column contains icons for editing and deleting items.

Product	Price	Quantity	Unit of measure	Discount, %	Total
Graphics Card MSI GTX 980Ti 6GD5	1,049.98	1.000	pieces	0.00	1,049.98
Graphics Card ASUS TURBO-GTX960-OC...	423.55	1.000	pieces	0.00	
Graphics Card ASUS GTX750TI-OC-2GD5	294.69	1.000	pieces	0.00	294.69

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

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

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

Name
Asus R9280X-DC2T-3GD5
Batery Back-up System APC Back-UPS ES 700VA (BE700G-RS)
Batery Back-up System APC Back-UPS Pro 900VA (BR900G-RS)
Batery Back-up System APC Smart-UPS C 1500VA LCD (SMC1500I)
Gamepad Logitech F310 Gamepad
Gamepad Logitech Wireless Gamepad F710
Graphics Card ASUS GTX750TI-2GD5
Graphics Card ASUS GTX750TI-OC-2GD5
Graphics Card ASUS TURBO-GTX960-OC-2GD5
Graphics Card MSI GTX 980Ti 6GD5

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

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

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

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

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

Details

Beginner

Easy

Medium

Advanced

Introduction

Details are used to display supplemental data for a primary section object. The section details are displayed on the section edit page tabs in the tabs area.

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

- Details with edit fields.
- Details with adding page.
- Details with editable list.
- Details with selection from lookup.

More information about details of different types can be found in the "**Details**" article.

The detail creation

A custom detail must be registered so that the detail wizard could work with it. To register a detail, add a record with detail caption, detail schema identifier *DetailSchemaUid* (from the *Uid* column in the *SysSchema* table) and detail object schema identifier *EntitySchemaUid* (from the *Uid* column in the *SysSchema* table) to the *SysDetail* table.

More information about creating details of different types can be found in the "**Work with details**" article.

The base schema of the BaseDetailV2 detail

All detail schemas must be inherited from the *BaseDetailV2* base schema. The base logic of data initialization and communication with the edit page are implemented in the schema.

The base schema class has the following properties:

BaseDetailV2 messages

The massages are used for communication between the detail and the edit page. A full list of messages, their broadcast mode and direction are given in the table 1.

Table 1. The messages of the base detail

Name	Mode	Direction	Description
<i>GetCardState</i>	Address	Publication	Returns a state of the edit page.
<i>SaveRecord</i>	Address	Publication	Tells the edit page to save the data.
<i>DetailChanged</i>	Address	Publication	Tells the edit page about the changes of the detail data.
<i>UpdateDetail</i>	Address	Subscription	A subscription to the edit page update.
<i>OpenCard</i>	Address	Publication	Opens edit page.

The message modes are defined in the *Terrasoft.core.enums.MessageMode* enumeration and message direction is defined in the *Terrasoft.core.enums.MessageDirectionType*. More information about them can be found in the “**JavaScript API for platform core (on-line documentation)**” article.

BaseDetailV2 attributes

The *attributes* property contains the attributes of detail view model. The attributes that are defined in base detail class are given in the table 2.

Table 2. The attributes of the base detail

CanAdd

BOOLEAN

Indicates the possibility to add data.

CanEdit

BOOLEAN

Indicates the possibility to edit data.

CanDelete

BOOLEAN

Indicates the possibility to delete data.

Collection

COLLECTION

Detail data collection.

Filter

CUSTOM_OBJECT

Detail filter. Used for filtering detail data.

DetailColumnName

STRING

The column name where the filtering is performed.

MasterRecordId

GUID

The key value of the parent record.

IsDetailCollapsed

BOOLEAN

Indicates if the detail is collapsed.

DefaultValues

CUSTOM_OBJECT

The default value of the model columns.

Caption

STRING

The detail caption.

The available attribute data types are represented by the *Terrasoft.DataValueType* enumeration. More information about them can be found in the “**JavaScript API for platform core (on-line documentation)**” article.

BaseDetailV2 methods

The methods defined in base detail class are given in the table 3.

Table 3. The methods of the base detail

`init`

Initializes the detail page.

Parameters:

- `{Function} callback` – callback function;
- `{Object} scope` – the context of the method execution.

`initProfile`

Initializes the schema profile. Default value is `Terrasoft.emptyFn`.

`initDefaultCaption`

Sets the default caption of the detail.

`initDetailOptions`

Initializes the list view data collection.

`subscribeSandboxEvents`

It is subscribed to the messages necessary for the work of the detail.

`getUpdateDetailSandboxTags`

Generates the array of tags for the `UpdateDetail` message.

`updateDetail`

Updates the detail according to the parameters passed. Default value is `Terrasoft.emptyFn`.

Parameters:

- `{Object} config` – configuration object that contains the properties of the detail.

`initData`

Initializes the list view data collection.

Parameters:

- `{Function} callback` – callback function;
- `{Object} scope` – the context of the method execution.

`getEditPageName`

Returns the name of the edit page depending on the record type at editing or on selected record type for adding.

`onDetailCollapsedChanged`

The handler of collapsing or expanding of the detail.

Parameters:

- `{Boolean} isCollapsed` – the attribute of the collapsed/expanded detail.

`getToolsVisible`

Returns the collapse value of the detail.

`getDetailInfo`

Publishes a message to get information about the detail.

BaseDetailV2 array of modifications

In the `diff` modifications array of the base detail, only a base container for detail view is defined:

```
diff: /**SCHEMA_DIFF*/ [
```

```
// Base container for detail view.
{
    "operation": "insert",
    "name": "Detail",
    "values": { ... }
}
]/**SCHEMA_DIFF*/
```

The “BaseGridDetailV2” base “detail with list” class

The class is a *BaseDetailV2* inheritor. All details with lists must be the *BaseGridDetailV2* inheritors. The list base logic (import, filtering, adding, deleting and editing the detail records) is implemented in the *BaseGridDetailV2* schema.

More information about creating custom details can be found in the **“Adding a detail with an editable list”** article.

BaseGridDetailV2 messages

Main *BaseGridDetailV2* messages are given in table 4.

Table 4. The messages of the detail with a list

Name	Mode	Direction	Description
<i>getCardInfo</i>	Address	Subscription	Returns information about the edit page: its default values, type column name and type column value.
<i>CardSaved</i>	Broadcasting	Subscription	Handles a message of saving the edit page.
<i>UpdateFilter</i>	Broadcasting	Subscription	Refreshes filters in the detail.
<i>GetColumnsValues</i>	Address	Publication	Receives the column values of the edit page model.

The message modes are defined in the “Terrasoft.core.enums.MessageMode” enumeration and message direction in the “Terrasoft.core.enums.MessageDirectionType” enumeration. More information about them can be found in the **“JavaScript API for platform core (on-line documentation)”** article.

BaseGridDetailV2 attributes

Main *BaseGridDetailV2* attributes are given in table 5.

Table 5. The attributes of the detail with a list

ActiveRow
GUID

The value of the primary column of the active record in the list.

IsGridEmpty
BOOLEAN

Indicates that the list is empty.

MultiSelect
BOOLEAN

Indicates if multiple selection is permitted.

SelectedRows
COLLECTION

An array of selected values.

RowCount
INTEGER

Number of rows in the list.

IsPageable
BOOLEAN

Indicates if the page-by-page loading is enabled.

SortColumnIndex
INTEGER

Index of the sorting column.

CardState
TEXT

Opening mode for the record edit page.

EditPageUID
GUID

A unique identifier of the edit page.

ToolsButtonMenu
COLLECTION

The collection of the functional button's drop-down list.

DetailFilters
COLLECTION

Collection of the detail filters.

IsDetailWizardAvailable
BOOLEAN

Indicates if the detail wizard is available.

The available attribute data types are represented by the *Terrasoft.DataType* enumeration. More information about them can be found in the “**JavaScript API for platform core (on-line documentation)**” article.

BaseGridDetailV2 mixins

Main *BaseGridDetailV2* mixins are given in the table 6.

Table 6. The mixins of the detail with a list

GridUtilities
Terrasoft.GridUtilities

Mixin for the list.

WizardUtilities
Terrasoft.WizardUtilities

Mixin for the detail wizard.

More information about the *GridUtilities* mixin is given below.

BaseGridDetailV2 methods

Main *BaseGridDetailV2* methods are given in table 7.

Table 7. The methods of the base detail with a list

init

Overrides the *BaseDetailV2* method. Calls the parent's *init* method logic, registers the messages, initializes the

filters.

Parameters:

- *{Function}* `callback` – callback function;
- *{Object}* `scope` – the context of the method execution.

`initData`

The override of the `BaseDetailV2` method. Calls the parent's `initData` method logic, initializes the data collection of the list view.

Parameters:

- *{Function}* `callback` – callback function;
- *{Object}* `scope` – the context of the method execution.

`loadGridData`

Executes the load of the list data.

`initGridData`

Executes the initialization of the default values for working with the list.

`getGridData`

Returns list collection.

`getFilters`

Returns the detail filter collection.

`getActiveRow`

Returns the identifier of the selected record in the list.

`addRecord`

Adds the new record to the list. Saves the edit page, if needed.

Parameters:

- *{String}* `editPageUid` – identifier of typed edit page.

`copyRecord`

Copies the record and opens the edit page.

Parameters:

- *{String}* `editPageUid` – identifier of typed edit page.

`editRecord`

Opens edit page of the selected record.

Parameters:

- *{Object}* `record` – record model for editing.

`subscribeSandboxEvents`

It is subscribed to the messages necessary for the detail operation.

`updateDetail`

The override of the `BaseDetailV2` method. Calls the parent's `updateDetail` method logic, updates the detail.

Parameters:

- *{Object}* `config` – configuration object that contains the properties of the detail.

`openCard`

Opens edit page.

Parameters:

- *{String}* *operation* – operation type (creating/modifying);
- *{String}* *typeColumnValue* – the value of record typing column;
- *{String}* *recordId* – record identifier.

onCardSaved

Handles the save event of the edit page where the detail is located.

addToolsButtonMenuItems

Adds elements to the collection of the functional button drop-down list.

Parameters:

- *{Terrasoft.BaseViewModelCollection}* *toolsButtonMenu* – the collection of the functional button drop-down list.

initDetailFilterCollection

Initializes the detail filter.

setFilter

Sets the detail filter value.

Parameters:

- *{String}* *key* – filter type;
- *{Object}* *value* – filter value.

loadQuickFilter

Loads the quick filter.

Parameters:

- *{Object}* *config* – parameters of the filters module load.

destroy

Clears the data, exports the detail.

BaseGridDetailV2 array of modifications

In the *diff* modifications array of the base detail, only a base container for detail view is defined:

```
diff: /**SCHEMA_DIFF*/ [
  {
    // Element for displaying the list.
    "operation": "insert",
    "name": "DataGrid",
    "parentName": "Detail",
    "propertyName": "items",
    "values": {
      "itemType": Terrasoft.ViewItemType.GRID,
      ...
    }
  },
  {
    // List reloading button.
    "operation": "insert",
    "parentName": "Detail",
    "propertyName": "items",
    "name": "loadMore",
    "values": {
      "itemType": Terrasoft.ViewItemType.BUTTON,
      ...
    }
  }
]
```

```

        }
    },
{
    // Record adding button.
    "operation": "insert",
    "name": "AddRecordButton",
    "parentName": "Detail",
    "propertyName": "tools",
    "values": {
        "itemType": Terrasoft.ViewItemType.BUTTON,
        ...
    }
},
{
    // Typed record adding button.
    "operation": "insert",
    "name": "AddTypedRecordButton",
    "parentName": "Detail",
    "propertyName": "tools",
    "values": {
        "itemType": Terrasoft.ViewItemType.BUTTON,
        ...
    }
},
{
    // Detail menu.
    "operation": "insert",
    "name": "ToolsButton",
    "parentName": "Detail",
    "propertyName": "tools",
    "values": {
        "itemType": Terrasoft.ViewItemType.BUTTON,
        ...
    }
}
] /**SCHEMA_DIFF*/

```

The GridUtilitiesV2 mixin

GridUtilitiesV2 is a mixin that implements the logic of the “list” control. Features that are implemented in the *Terrasoft.configuration.mixins.GridUtilities* class:

1. Message subscription
2. Data load.
3. Working with the list:
 - selection of the records (the search of the active records)
 - adding, deleting and modifying the records
 - setting up the filters
 - sorting
 - exporting to the file
 - checking the access permissions to the list records

GridUtilitiesV2 methods

Main GridUtilitiesV2 methods are given in table 8.

Table 8. The methods of the base detail with a list

init

Subscribes to the events.

destroy

Deletes event subscriptions.

`loadGridData`

Executes the load of the list data.

`beforeLoadGridData`

Prepares the view model to the data load.

`afterLoadGridData`

Prepares the view model after the data load.

`onGridDataLoaded`

A handler of the data load event. Executes when the server returns the data.

Parameters:

- `{Object} response` — the result of fetching the data from the database.

`addItemsToGridData`

Adds a collection of new elements to the list collection.

Parameters:

- `{Object} dataCollection` — collection of new elements;
- `{Object} options` — adding parameters.

`reloadGridData`

Reloads the list.

`initQueryOptions`

Initializes the settings of the query instance, such as pagination and hierarchy.

Parameters:

- `{Terrasoft.EntitySchemaQuery} esq` — in this query the necessary settings will be initialized.

`initQuerySorting`

Initializes the sorting column.

Parameters:

- `{Terrasoft.EntitySchemaQuery} esq` — in this query the necessary settings will be initialized.

`prepareResponseCollection`

Modifies the data collection before loading it to the list.

Parameters:

- `{Object} collection` — list elements collection.

`getFilters`

Returns filters that are applied to current schema. It is overridden in the inheritors.

`exportToFile`

Exports the contents of the list into a file.

`sortGrid`

Performs list sorting.

Parameters:

- `{String} tag` — a key that shows how to sort the list.

`deleteRecords`

Initiates the deletion of the selected records.

`checkCanDelete`

Checks the ability to delete a record.

Parameters:

- `{Array} items` – the identifiers of the selected records;
- `{Function} callback` – callback function;
- `{Object} scope` – the context of the method execution.

`onDeleteAccept`

Performs the deletion after the confirmation of the user.

`getSelectedItems`

Returns the selected records in the list.

`removeGridRecords`

Removes the deleted records from the list.

Parameters:

- `{Array} records` – deleted records.

A detail with editable list

A detail with editable list enables editing records directly in the list without going to the record editing page. To make a detail list editable, you need to modify its schema the following way:

1. Add the dependencies from the `ConfigurationGrid`, `ConfigurationGridGenerator` and `ConfigurationGridUtilities` modules.
2. Connect the `ConfigurationGridUtilites` and `OrderUtilities` mixins.
3. Set the `IsEditable` attribute to “true”.
4. Add a configuration object in the modification array where the properties will be set and the methods that handle the detail list events will be bound.

The development case of creating a detail with an editable list can be found in the [“Adding a detail with an editable list”](#) article.

The “ConfigurationGrid” module

The `ConfigurationGrid` module contains the implementation of the “Configuration Grid” control. The `Terrasoft.ConfigurationGrid` class is the inheritor of the `Terrasoft.Grid` class. Main `Terrasoft.ConfigurationGrid` methods are given in table 9.

Table 9. Configuration grid methods

`init`

Initializes a component. Subscribes to the events.

`activateRow`

Selects the string and adds edit elements.

Parameters:

- `{String|Number} id` – the identifier of the list string.

`deactivateRow`

Removes selection of a string and removes edit elements.

Parameters:

- `{String|Number} id` – the identifier of the list string.

`formatCellContent`

Formats the data of a string cell.

Parameters:

- *{Object} cell* — the cell;
- *{Object} data* — the data;
- *{Object} column* — the cell configuration.

onUpdateItem

The handler of the record update event.

Parameters:

- *{Terrasoft.BaseViewModel} item* — collection element.

onDestroy

Destroys the list and its components.

ConfigurationGridGenerator module

The *Terrasoft.ConfigurationGridGenerator* generates list configuration and is an inheritor of the *Terrasoft.ViewGenerator* class. The methods that are implemented in the *Terrasoft.ConfigurationGridGenerator* class are given in table 10.

Table 10. The methods of the configuration grid generator

addLinks

Overridden method of the *Terrasoft.ViewGenerator* class. No links will be added to the editable list.

generateGridCellValue

Overridden method of the *Terrasoft.ViewGenerator* class. Generates a value configuration in the cell.

Parameters:

- *{Object} config* — column configuration.

The ConfigurationGridUtilities module

The *Terrasoft.ConfigurationGridUtilities* class contains methods that initialize a view model of the list string, process the active record actions and handle “hotkeys”.

The main properties of the *Terrasoft.ConfigurationGridUtilities* class are given in table 11 and its methods are provided in table 12.

Table 11. The *Terrasoft.ConfigurationGridUtilities* class properties

currentActiveColumnName

String

The name of the currently selected column.

columnsConfig

Object

Column configuration.

systemColumns

Array

Collection of the system column names.

Table 12. The *Terrasoft.ConfigurationGridUtilities* class methods

onActiveRowAction

Handles clicking an action of the active record.

Parameters:

- *{String}* buttonTag — a tag of the selected action;
- *{String}* primaryColumnName — active record identifier.

saveRowChanges

Saves the record.

Parameters:

- *{Object}* row — list string;
- *{Function}* [callback] — callback function;
- *{Object}* scope — the callback function context.

activeRowSaved

Handles the result of saving the record.

Parameters:

- *{Object}* row — list string;
- *{Function}* [callback] — callback function;
- *{Object}* scope — the callback function context.

initActiveRowKeyMap

Initializes the subscription to the button events in the active string.

Parameters:

- *{Array}* keyMap — events description.

getCellControlsConfig

Returns the configuration of the list cell edit items.

Parameters:

- *{Terrasoft.EntitySchemaColumn}* entitySchemaColumn — the column of the list cell.

copyRow

Copies and adds a record to the list.

Parameters:

- *{String}* recordId — the identifier of a copying record.

initEditableGridRowViewModel

Initializes the classes of the collection elements of the edited list.

Parameters:

- *{Function}* [callback] — callback function;
- *{Object}* scope — the callback function context.

Mini-page

Beginner

Easy

Medium

Advanced

Overview

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

Fig. 1 Contact mini-page

Web	Primary phone	Type
www.xtg.au	+61 2 6247 5656	Customer
148 Bunda St, Canberra ACT 2601, Australia, 10351	City	Australia
www.globalventure.com.ny	+1 212 721 1810	Customer
412 Pine Street, New York, NY	City	United States
www.feature-it.com	+1 212 735 2537	Partner
85 46th Street	City	United States

Fig. 2 Account mini-page

Web	Primary phone	Type
www.xtg.au	+61 2 6247 5656	Customer
148 Bunda St, Canberra ACT 2601, Australia, 10351	City	Australia
www.globalventure.com.ny	+1 212 721 1810	Customer
412 Pine Street, New York, NY	City	United States
www.feature-it.com	+1 212 735 2537	Partner
85 46th Street	City	United States

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

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

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

Modal windows

[Beginner](#)

[Easy](#)

[Medium](#)

[Advanced](#)

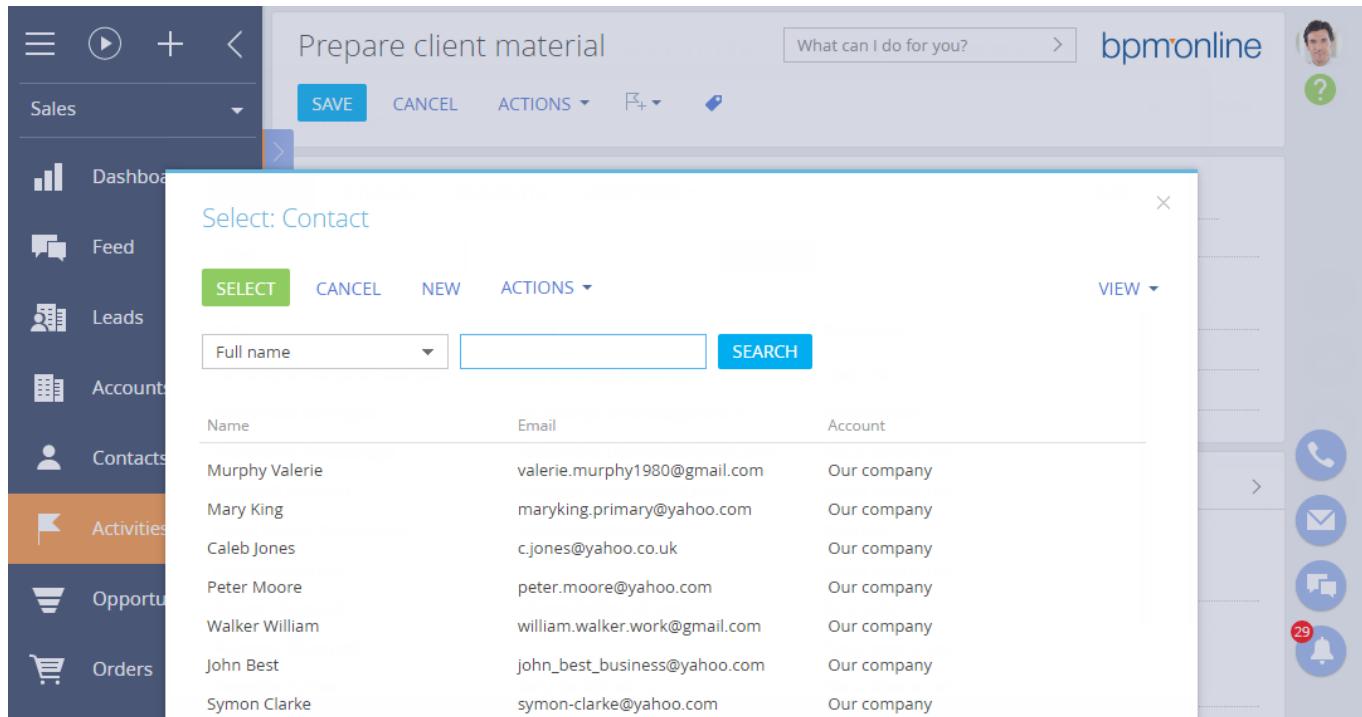
Overview

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

The modal windows are used to display and select data from various lookups, for example, when selecting an activity

assignee from the contact lookup (fig.1).

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



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

Communication panel

Beginner

Easy

Medium

Advanced

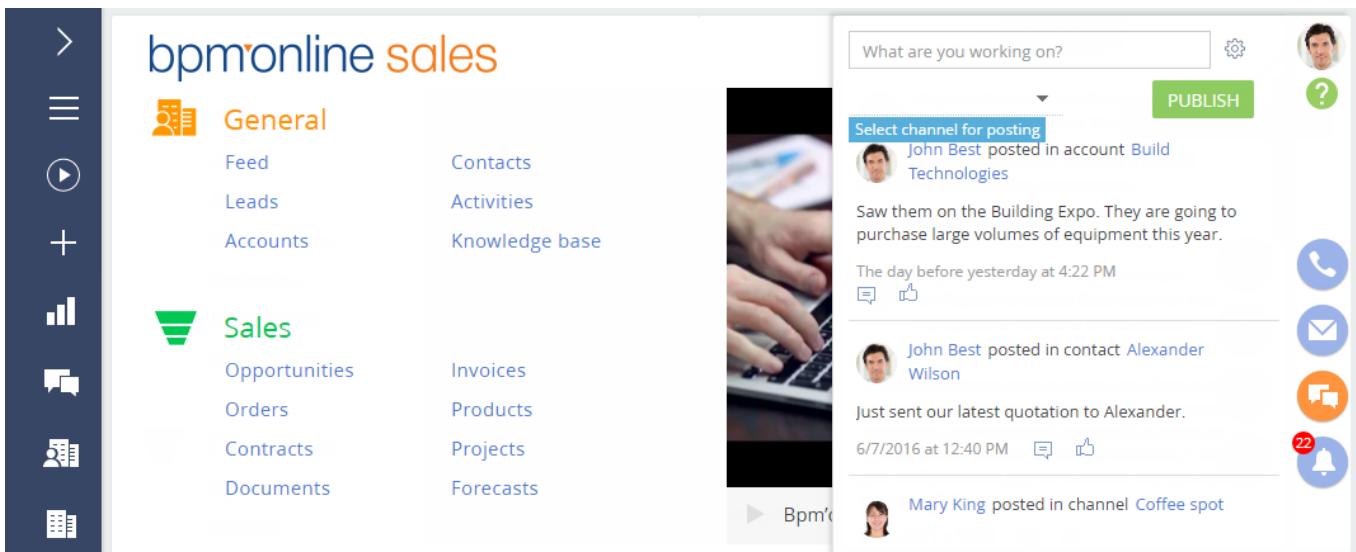
Overview

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

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

- Calls (1) — enables you to accept and make calls directly in the application. Read more about the possibilities of telephony in the "[Phone integration](#)" article.
- Email (2) — designed to work with email. Features of configuration and integration with email services are described in the "[Working with emails](#)" article.
- Feed (3) — displays messages of the [Feed] section and is used to view messages in subscribed channels and to add new posts and comments. More information can be found in the "[The \[Feed\] section](#)" article.
- The notification center displays notifications about various events in the system. It is described in detail in the "[Notification center](#)" article.

Fig. 1 Communication panel



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

Command line

Beginner

Easy

Medium

Advanced

Overview

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

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

Fig. 1 Command line

A screenshot of the command line interface. On the left is a sidebar with navigation links: Sales, Dashboards, Feed, Leads, Accounts, Contacts (which is highlighted in orange), Activities, and Opportunities. The main area has a header with 'mainHeaderContainer' and 'CONTACTS'. Below it is a search bar with 'Find Lead' and a dropdown arrow. A 'NEW CONTACT' button and an 'ACTIONS' dropdown are also present. The main content area shows a contact card for 'John Best' with fields for Account (Our company), job title (Head of department), and Email (john_best_business@ym). To the right is a list of suggestions starting with 'Find Lead Actual Works, Edith Roberts' and continuing with various lead names and their details.

The features of the command line are:

- Navigation – "Go to..." a section
- Search for records – "Search ..." for contacts, accounts or records

- Creating records — "Create..." a record
- Start business process — "Start process ..."
- Create custom commands with the "Create custom command".

Read more about the possibilities of the command line in the "[Command line](#)" article.

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

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

Action dashboard

Beginner

Easy

Medium

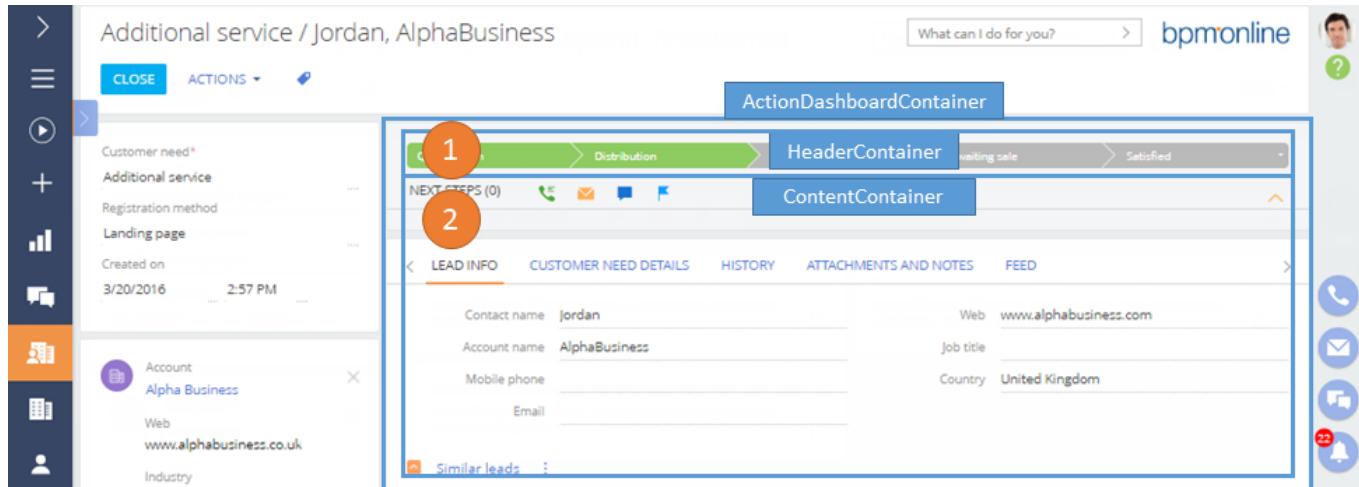
Advanced

Overview

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

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

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



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

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

Dashboard widgets

Contents

- **Introduction**
- **Charts**
- **Metrics**

- **Gauge**
- **Lists**
- **Web-page**
- **Sales pipeline**

Dashboard widgets

Beginner

Easy

Medium

Advanced

General information

Dashboard widgets (analytic elements) are used for data analysis of sections. Go to the “Dashboards” view of the required section to work with its analytics. Use the [Dashboards] section to work with the entirety of Creatio section data analytics.

To learn more about Creatio dashboard widgets, please refer to the “[Section analytics](#)” article.

Data storage structure of dashboards

The dashboards section is a user-defined set of tab elements. The mechanism for working with dashboards is implemented with the help of the *DashboardManager* dashboard client manager and the *DashboardManagerItem* element client manager, which represents the tabs. The *SysDashboard* object is responsible for dashboards in the system. The *SysDashboard* object properties are described in table 1.

Table 1. SysDashboard object properties

Name	Header	Type	Details
Caption	Header	String	This information is displayed in the tab header.
Position	Position	Number	If a position is not specified, the elements are displayed in alphabetical order.
Section	Section	Lookup	System section.
ViewConfig	Element (widget) view configuration	Array	[// Element type (Terrasoft.ViewItemType). itemType: "4", // Element name. name: "SomeInvoiceChart", // View configuration. layout: { columns: 4, rows: 4, colspan: 4, rowspan: 4 } }, {...}]
Items	Element (widget) module configuration	JSON Object	{ // The name of the element for which the module // settings are defined. "SomeInvoiceChart": {

```
// Name of the "DashboardItem" view element.
"widgetType": "Chart",

// Parameters required to display data for a
// particular "DashboardItem" element.

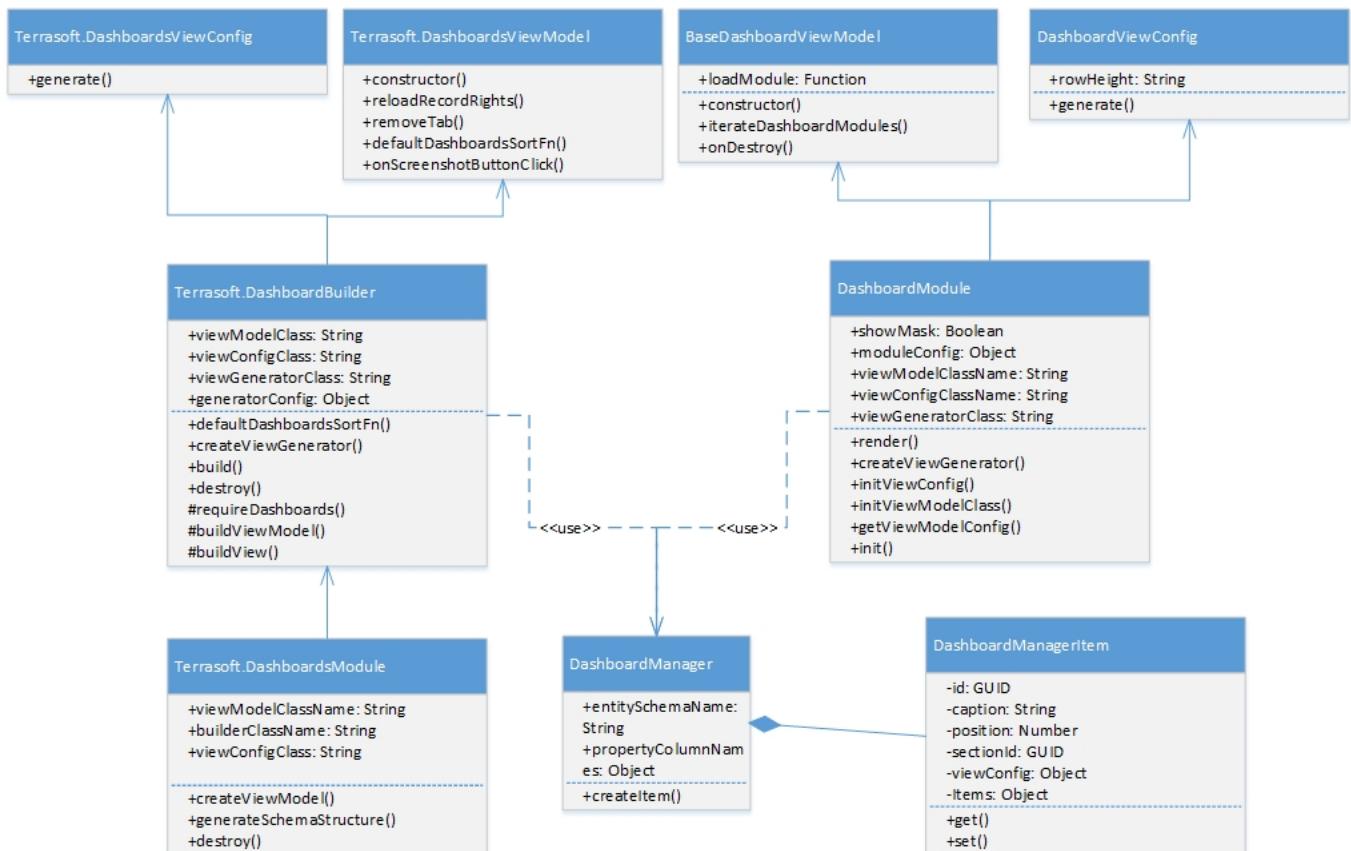
"parameters": {

    "caption": "some caption",
    ...
},
{
...
}
}
```

Implementing functionality in the dashboards view mode

The hierarchy of classes that implement functionality in the dashboards view mode is displayed on Fig. 1.

Fig. 1. The hierarchy of classes that implement functionality in the dashboards view mode



The **SectionDashboardModule** module:

- The **SectionDashboardBuilder** encapsulates the view generation logic and view model class for the [Dashboards] section module.
- SectionDashboardsViewModel** – the model class of the [Dashboards] section view model.
- SectionDashboardsModule** – [Dashboards] section class module.

The **DashboardModule** module:

- DashboardViewConfig** – a class that generates the view configuration for the dashboards page view module.

- *BaseDashboardViewModel* – a base class for the dashboards page view model.
- *DashboardModule* – a class that contains functionality for working with dashboard modules.

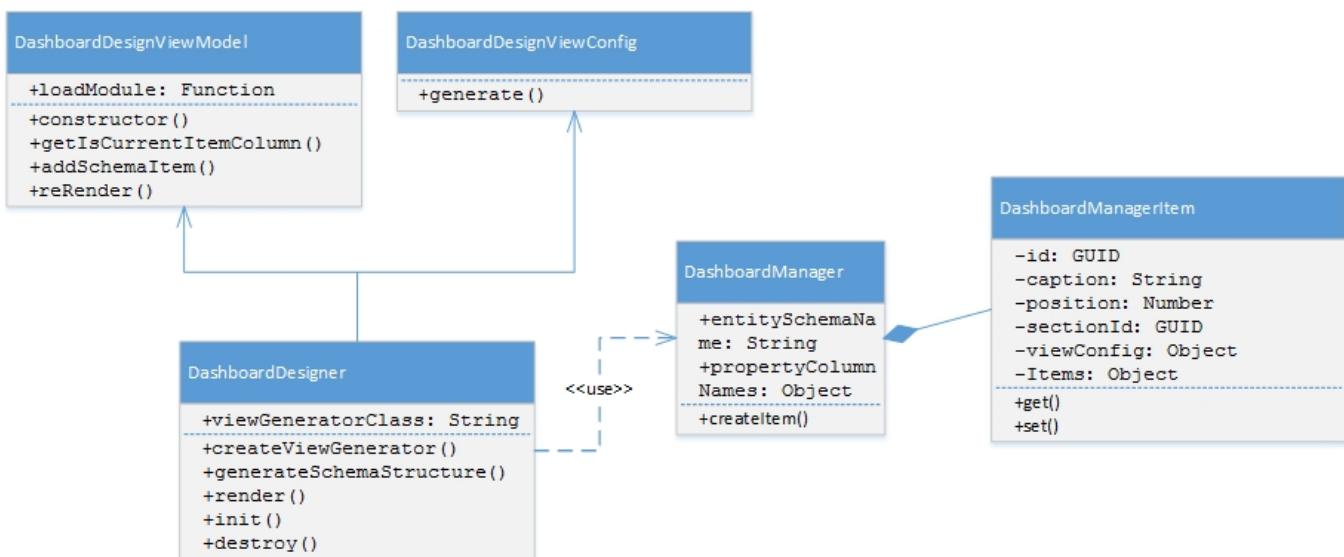
The *DashboardBuilder* module:

- *DashboardsViewConfig* – a class that generates a dashboards module view configuration.
- *BaseDashboardsViewModel* – a base class of the dashboards section view model.
- *DashboardBuilder* – a class for dashboards module construction.

Implementing functionality in the dashboards view mode

The hierarchy of classes that implement the functionality in the dashboards view mode is displayed in Fig. 2.

Fig. 2. The hierarchy of classes that implement the functionality in the dashboards view mode



The *DashboardDesigner* module:

- *DashboardDesignerViewConfig* – a class that generates the view configuration for the dashboards designer module.
- *DashboardDesignerViewModel* – a class of the dashboards designer view model.
- *DashboardDesigner* – dashboard visual module class.

Base classes that implement widget functionality

BaseDashboardsViewModel – a base class of the dashboards section view model. To use this class, register the following messages in the module:

- *GetHistoryState* (publish; ptp);
- *ReplaceHistoryState* (publish; broadcast);
- *HistoryStateChanged* (subscribe; broadcast);
- *GetWidgetParameters* (subscribe; ptp);
- *PushWidgetParameters* (subscribe; ptp) – if the parameters are drawn from modules (useCustomParameterMethods = true).

BaseWidgetDesigner – base widget settings view schema. Main methods:

- *GetWidgetConfig()* – returns the current widget settings object.
- *GetWidgetConfigMessage()* – returns the name of the message used for getting widget module settings.
- *GetWidgetModuleName()* – returns the name of the widget module.
- *GetWidgetRefreshMessage()* – returns the name of the widget update message.
- *getWidgetModulePropertiesTranslator()* – returns the connecting object of widget module properties and widget module settings.

BaseAggregationWidgetDesigner – contains methods for working with aggregate columns and aggregation types.

DashboardEnums – contains an enumeration of widget properties.

Terrasoft.DashboardEnums.WidgetType – contains the widget view mode and design mode configuration of the dashboards. The configuration is defined by the following properties:

- *moduleName* – widget module name.
- *ConfigurationMessage* – the name of the module settings receiving message.
- *ResultMessage* – the name of the message that returns widget designer module settings.
- *StateConfig (stateObj)* – widget designer schema name.

Charts

Beginner

Easy

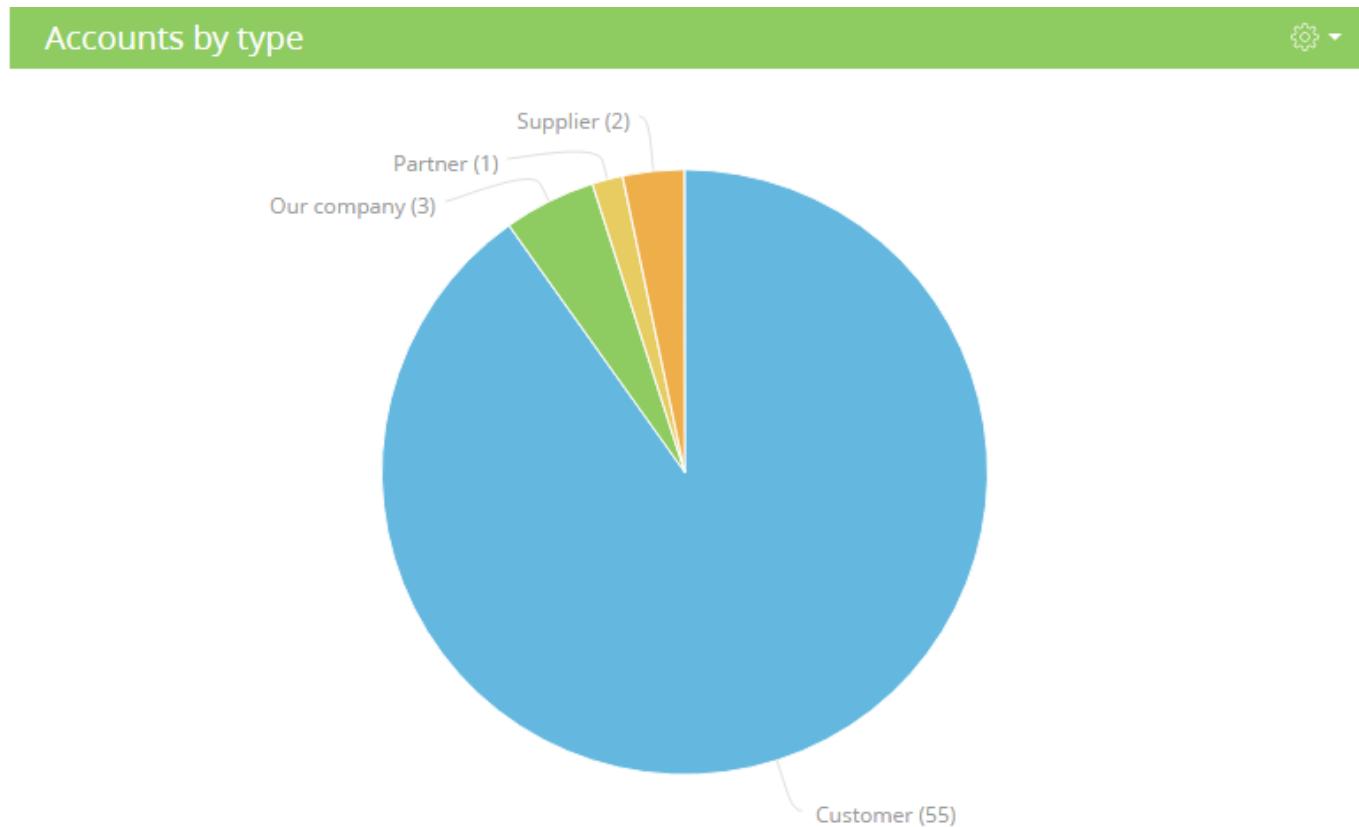
Medium

Advanced

General information

Charts display multiple system records in the form of diagrams of different types. For example, you can display a pie chart of accounts distributed by type. Charts display information in the form of different diagram types or in a data list form. Learn more about charts in the “[The “Chart” dashboard component](#)” article. Chart settings are described in the [How to set up a “Chart” dashboard component](#) article.

The “Charts” dashboard



Charts functionality implementation classes

ChartViewModel – chart view model.

ChartViewConfig – generates the chart view model.

ChartModule – a module designed to work with charts.

ChartDesigner – view model schema of a chart.

ChartModuleHelper – generates a query using the *Terrasoft.EntitySchemaQuery* object.

ChartDrillDownProvider – contains methods for working with the “Show data” function (used for working with chart series).

Chart setup parameters

To configure a chart, you need to add the JSON configuration object with the chart properties to the widget module configuration. The widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “**Dashboard widgets**” article.

Set the “Chart” value to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible chart parameters are listed in table 1.

Table 1. Chart setup parameters

Name	Type	Details
<i>seriesConfig</i>	<i>object</i>	The settings of an embedded chart in a series.
<i>orderBy</i>	<i>string</i>	Sorting field.
<i>orderDirection</i>	<i>string</i>	Sorting direction.
<i>caption</i>	<i>string</i>	Chart header.
<i>sectionId</i>	<i>string</i>	Section id.
<i>xAxisDefaultCaption</i>	<i>string</i>	Default X-axis header.
<i>yAxisDefaultCaption</i>	<i>string</i>	Default Y-axis header.
<i>primaryColumnName</i>	<i>string</i>	Name of initial column. The “id” column is the default one.
<i>yAxisConfig</i>	<i>object</i>	Array of the Y-axis name settings.
<i>schemaName</i>	<i>string</i>	Chart object.
<i>sectionBindingColumn</i>	<i>string</i>	Section link column.
<i>func</i>	<i>string</i>	Aggregate function.
<i>type</i>	<i>string</i>	Chart type.
<i>XAxisCaption</i>	<i>string</i>	X-axis caption.
<i>YAxisCaption</i>	<i>string</i>	Y-axis caption.
<i>xAxisColumn</i>	<i>string</i>	The X-axis grouping column.
<i>yAxisColumn</i>	<i>string</i>	The Y-axis grouping column.
<i>styleColor</i>	<i>string</i>	Chart color.
<i>filterData</i>	<i>object</i>	Filter settings.

Metrics

Beginner Easy **Medium** Advanced

The “Metric” dashboard (Fig. 1) displays the number (or date) received by inquiring system data, for example, a total number of company’s employees. Learn more about analytics in the “[The “Metric” dashboard component](#)” article. Analytics settings are described in the “[Setting up the “Metric” dashboard component](#)” article.

(Fig. 1). A “metric” dashboard



Functionality implementation classes of the “Metric” dashboard

IndicatorViewModel – metric view model.

IndicatorViewConfig – generates the configuration of the metric view model.

IndicatorModule – a module designed to work with metrics.

IndicatorDesigner – view model schema of the metric edit page.

Metric settings

To configure a metric, you need to add the JSON configuration object with Metric properties to the widget module configuration. The widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “[Dashboard widgets](#)” article.

Set the “Metric” value to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible metric parameters are listed in table 1.

Table 1. Metric settings

Name	Type	Details
<i>caption</i>	<i>string</i>	Metric header
<i>sectionId</i>	<i>string</i>	Section id.
<i>entitySchemaName:</i>	<i>string</i>	Metric object.
<i>sectionBindingColumn</i>	<i>string</i>	Section link column.
<i>columnName</i>	<i>string</i>	Name of aggregating column.
<i>format</i>	<i>object</i>	Metric format.
<i>filterData</i>	<i>object</i>	Filter settings.
<i>aggregationType</i>	<i>number</i>	Type of aggregating function.
<i>style</i>	<i>string</i>	Metric color.

Gauge

Beginner

Easy

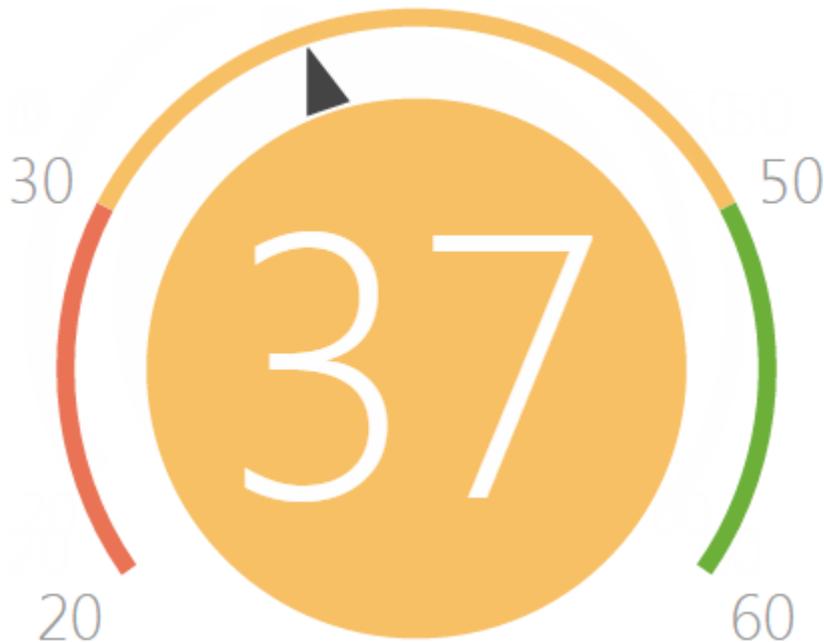
Medium

Advanced

General information

A “gauge” dashboard element displays aggregate data from multiple system records in the form of a dial with green, yellow and red areas on its scale. For example, you may use this dashboard to display a number of performed activities and compare it to a desired rate.

(Fig. 1). A gauge dashboard



Gauge functionality implementation classes

GaugeViewModel – gauge view model.

GaugeViewConfig – generates the gauge view model.

GaugeModule – module designed to work with gauges.

GaugeChart – implements a gauge chart component.

GaugeDesigner – view model schema of a gauge.

Gauge settings

To configure a gauge, you need to add the JSON configuration object with the gauge properties to the widget module configuration. The widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “**Dashboard widgets**” article.

Set the “Gauge” value to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible gauge parameters are listed in table 1.

Table 1. Gauge settings

Name	Type	Details
<i>caption</i>	<i>string</i>	Gauge header.
<i>sectionId</i>	<i>string</i>	Section id.
<i>entitySchemaName:</i>	<i>string</i>	Gauge object.
<i>sectionBindingColumn</i>	<i>string</i>	Section link column.

Lists

Beginner

Easy

Medium

Advanced

Introduction

A list displays multiple system records in a unified visual form. Lists enable you to limit the number of records displayed to create such dashboards as the “Top ten most productive managers by the number of closed deals”, for example. Learn more about the lists in the [The “List” dashboard component](#) article. List settings are described in the [How to set up the “List” dashboard component](#) article.

Fig. 1. A “list” dashboard

Top 3 managers by productivity	
James Rodrick	70
Stephanie Lowe	68
Andrew Morrison	61

List functional classes

DashboardGridViewModel – list view model.

DashboardGridViewConfig – generates list view configuration.

DashboardGridModule – module designed to work with lists.

DashboardGridDesigner – list editing page schema.

List settings

To configure a list, you need to add the JSON configuration object with list properties to the widget module configuration. The widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the [“Dashboard widgets”](#) article.

Set the “*DashboardGrid*” value of to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible list parameters are listed in table 1.

Table 1. List settings

Name	Type	Details
<i>caption</i>	<i>string</i>	List header.
<i>sectionBindingColumn</i>	<i>string</i>	Section link column.
<i>filterData</i>	<i>object</i>	Filter settings.
<i>sectionId</i>	<i>string</i>	Section id.
<i>entitySchemaName:</i>	<i>string</i>	List object.
<i>style</i>	<i>string</i>	List color.
<i>orderDirection</i>	<i>number</i>	Sorting options (1 - ascending, 2 - descending).
<i>orderColumn</i>	<i>string</i>	List sorting column.
<i>rowCount</i>	<i>number</i>	The number of rows to display.
<i>gridConfig</i>	<i>object</i>	List configuration.

Web-page

Beginner

Easy

Medium

Advanced

The “web-page” dashboard is used to display web-pages on the dashboard panel. It may be an online currency calculator, your corporate website, etc. Web-page dashboards are described in the “[Setting up the “Web-page” dashboard](#)” article.

Web-page functionality implementation classes

WebPageViewModel – web-page view model.

WebPageViewConfig – generates the web-page view model configuration.

WebPageModule – module used to work with web-pages.

WebPageDesigner – web-page widget view schema.

Web-page settings parameters

To configure a web-page, you need to add the JSON configuration object with web-page properties to the widget module configuration. Widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “**Dashboard widgets**” article.

Set the “WebPage” value to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible web-page parameters are listed in table 1.

Table 1. Web-page settings

Name	Type	Details
<i>caption</i>	<i>string</i>	Web-page widget title.
<i>sectionId</i>	<i>string</i>	Section id.
<i>url</i>	<i>string</i>	Web-page link.
<i>style</i>	<i>string</i>	Web-page widget CSS-styles

Sales pipeline

Beginner Easy **Medium** Advanced

The “Sales pipeline” dashboard is used to analyze sales dynamics by stages. Learn more about the sales pipeline dashboard settings in the “[The “Sales pipeline” dashboard tile](#)” article.

Sales pipeline functionality implementation classes

OpportunityFunnelChart – a class inherited from *Chart*.

Sales pipeline settings

To configure a sales pipeline, you need to add the JSON configuration object with sales pipeline properties to the widget module configuration. Widget module configuration is defined by the *Items* property of the *SysDashboard* object. Learn more about the *SysDashboard* object and its properties in the “**Dashboard widgets**” article.

Set the “OpportunityFunnel” value to the *widgetType* property in the JSON configuration object with widget settings. In addition, assign the *parameters* property to the object with necessary parameters. Possible sales pipeline parameters are listed in table 1.

Table 1. Sales pipeline settings

Name	Type	Details
<i>caption</i>	<i>string</i>	Sales pipeline header.
<i>sectionId</i>	<i>string</i>	Section id.
<i>defPeriod</i>	<i>string</i>	Pipeline period (last week by default).
<i>sectionBindingColumn</i>	<i>string</i>	Section link column.

<i>type</i>	<i>string</i>	Chart type (“funnel”).
<i>filterData</i>	<i>object</i>	Filter settings.

The [Timeline] tab

Contents

- **Introduction**
- **Creating the [Timeline] tab tiles bound to custom section**

The [Timeline] tab

Beginner Easy Medium **Advanced**

Introduction

Starting from version 7.12.0 you can use the [Timeline] tab for quick analysis of customer cooperation, opportunity, case, etc. history in Creatio. This tab is available by default in the [Contacts], [Accounts], [Leads], [Opportunities] and [Cases] sections.

The database tables

The following tables are provided in the database for setting up the timeline:

- *TimelinePageSetting* – for setting up sections and their tiles (table 1).
- *TimelineTileSetting* – for setting up all existing and custom timeline tiles (table 2).
- *SysTimelineTileSettingLcz* – for localizing tile names (see “**Localization tables**”).

Table 1. – TimelinePageSetting table primary columns

Column	Details
<i>Id</i>	Record identifier.
<i>Key</i>	Key – the name of section page schema. For example, AccountPageV2, ContactPageV2, etc.
<i>Data</i>	Section timeline setup in JSON format.

Table 2. – TimelineTileSetting table primary columns

Column	Details
<i>Id</i>	Record identifier.
<i>Name</i>	Tile caption that will be displayed in the filter menu. It must have plural form, for example, “Tasks”. Localization is performed via the <i>SysTimelineTileSettingLcz</i> table. If this field is not populated, the tile caption will be derived from the entity or type schema name.
<i>Data</i>	Section timeline setup in JSON format (table 3).
<i>Image</i>	The tile icon that will be displayed in the filter menu and on the left side of the tile on the [Timeline] tab.

Table 3. – Timeline tile configuration parameters in JSON format.

Column	Details	If required	Example
<i>entityConfigKey</i>	Tile key. It should match the <i>Id</i> in the <i>TimelineTileSetting</i> table of the corresponding existing tile that should be displayed for the entity.	No	706f803d-6a30-4bcd-88e8-36a0e722ea41
<i>entitySchemaName:</i>	Name of the entity object schema.	Yes	Activity

<i>referenceColumnName</i>	Name of the object column that will be used for selecting records.	Yes	Account
<i>masterRecordColumnName</i>	Name of the parent record column that will be used for selecting records.	Yes	Id
<i>typeColumnName</i>	Name of the type column .	No	Type
<i>typeColumnValue</i>	Value of the type column.	Should only be applied when <i>typeColumnName</i> is indicated.	fbeoacdc-cfco-df11-boof-001d60e938c6
<i>viewModelClassName</i>	The view model class name of the existing tile.	No. If the value is not populated, the <i>BaseTimelineItemViewModel</i> base class will apply.	Terrasoft.ActivityTimelineItemViewModel
<i>viewClassName</i>	Name of the existing tile view class.	No. If the value is not populated, the <i>BaseTimelineItemView</i> base class will apply.	Terrasoft.ActivityTimelineItemView
<i>orderColumn</i>	Column for sorting.	Yes	StartDate
<i>authorColumnName</i>	Column for the author.	Yes	Owner
<i>captionColumnName</i>	Column for the caption.	Yes, if the <i>messageColumnName</i> column is not indicated.	Title
<i>messageColumnName</i>	Column for messages.	Yes, if the <i>captionColumnName</i> column is not indicated.	DetailedResult
<i>caption</i>	Tile caption that will be displayed in the filter menu. It must have plural form, for example, "Tasks". It is used for setting a tile caption that would differ from the one indicated in the <i>Name</i> field of the corresponding tile setting in <i>TimelinePageSetting</i> .	No	My Activity
<i>columns</i>	Setup array for additional tile columns.	No	
<i>columnName</i>	Path to the entity object column.	Yes	Result
<i>columnAlias</i>	Column alias in the tile model view.	Yes	ResultMessage
<i>isSearchEnabled</i>	Indicates the capability of text search according to the column value (for text columns only).	No	true

Adding the [Timeline] tab to the section

Adding the [Timeline] tab tiles to the new section is described in the "[Creating the \[Timeline\] tab tiles bound to custom section](#)" article.

To add the [Timeline] tab to the section page and display records thereon:

1. Add a new record to the *TimelinePageSetting* table.
2. Populate the corresponding columns (table 1). Indicate the section page schema name in the *Key* column. For example, if you need to add a tab to the {Accounts} section, the *Key* column value will be "AccountPageV2". The *Data* column contains the configuration of timeline tiles that are displayed on the indicated section tab in JSON format (table 3).

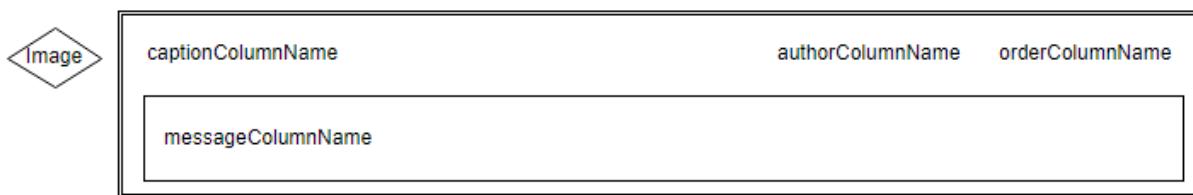
The [Timeline] tab will not be displayed on the section record edit page if the tile configuration in the *Data* column is not available or if there exist errors (for example, syntax error) in the configuration.

Usage of base tile

To start using the timeline in a section, perform base tile configuration (fig.1). The base tile compound elements:

- icon
- caption
- author
- date (sorting)
- message

Fig. 1. – The base tile element location



Example

Add the [Contract] tile to the [Accounts] section page. Sorting should be performed according to the *StartDate* column; the caption values, author and tile messages should be derived from the *Number*, *Owner* and *Notes* columns correspondingly.

Case implementation

1. Add a new record (or update an existing record) in the *TimelinePageSetting* table.
2. Set the "AccountPageV2" value for the *Key* column and populate the *Data* column with the following JSON object:

```
[  
  {  
    "entityConfigKey": "0ef5bd15-f3d3-4673-8af7-f2e61bc44cf0",  
    "entitySchemaName": "Contract",  
    "referenceColumnName": "Order",  
    "orderColumnName": "StartDate",  
    "authorColumnName": "Owner",  
    "captionColumnName": "Number",  
    "messageColumnName": "Notes",  
    "caption": "My Contracts",  
    "masterRecordColumnName": "Id"  
  }  
]
```

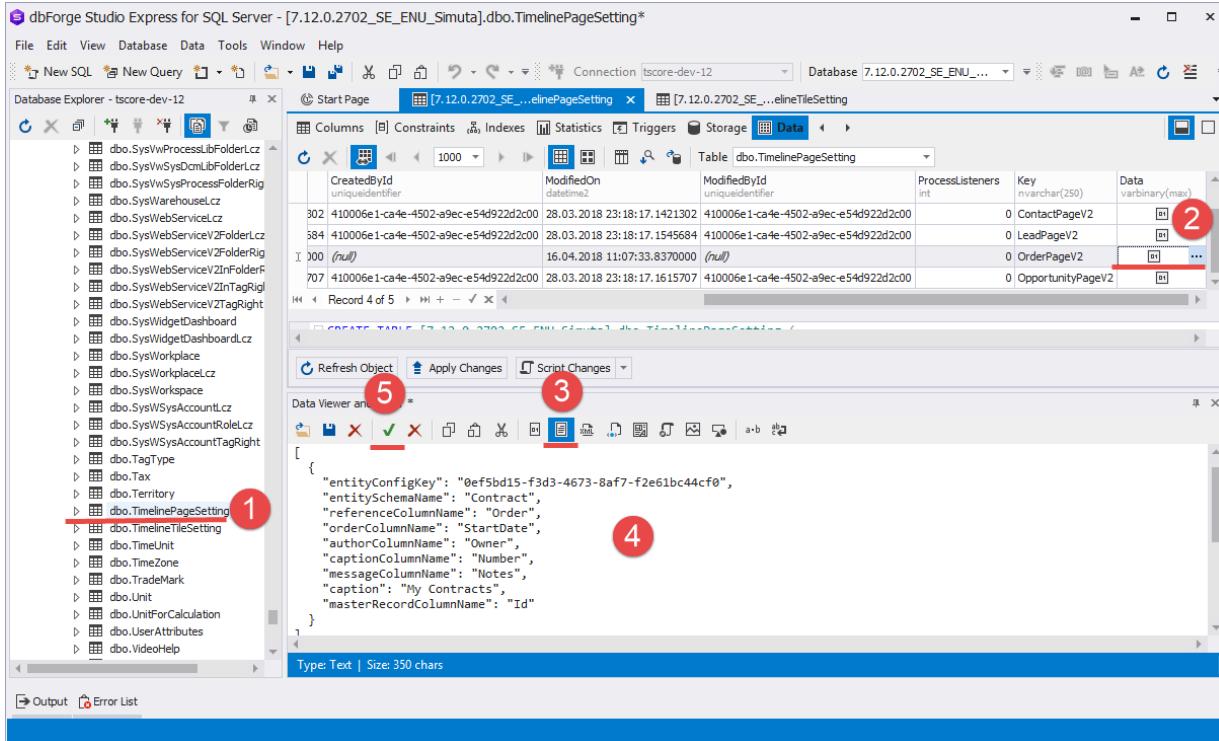
The [Orders] base tile is used in the following case. This tile has a record in the *TimelineTileSettings* table with the *0ef5bd15-f3d3-4673-8af7-f2e61bc44cf0* Id.

As the data in the *Data* column are stored in the varbinary(max) form, use specific editor (such as dbForge Studio Express for SQL Server) to modify them (Fig. 2). To do this:

1. Select a table.
2. Select the necessary column of the record and click the edit button.
3. Enter the text data display mode in the data editor.
4. Add necessary data.
5. Save the changes in the data editor.

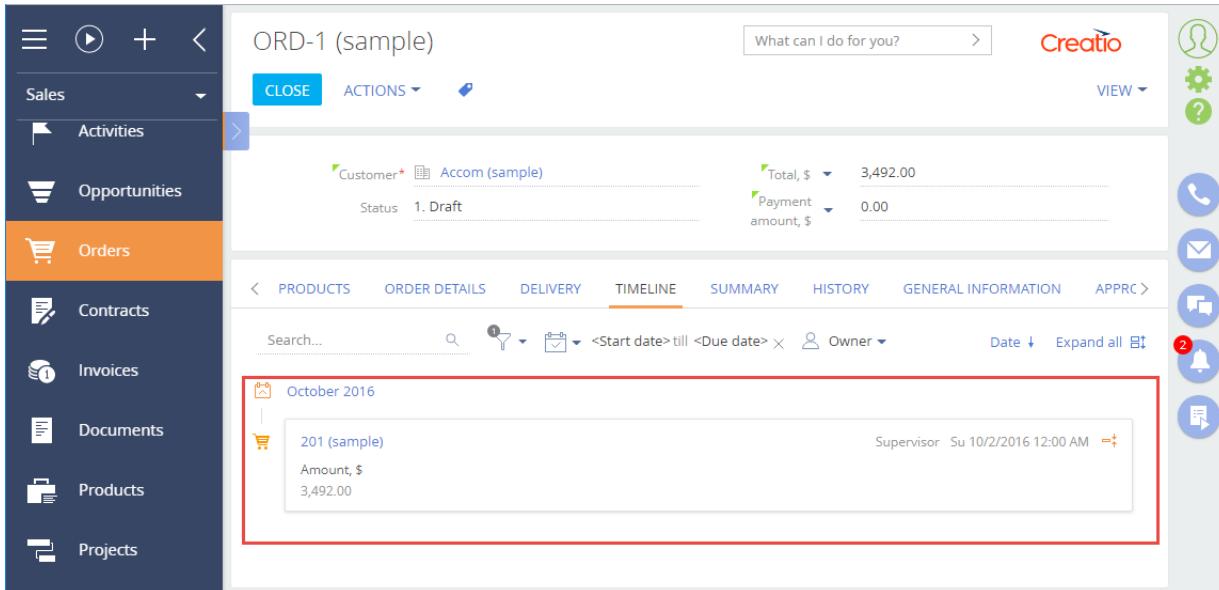
Creatio developer guide

Fig. 2. Editing data via the dbForge Studio Express for SQL Server



The result of the base tile usage on the [Timeline] tab in the [Accounts] section is shown in fig.3

Fig. 3. – The [Timeline] tab in the [Accounts] section.



See also

- [Creating the \[Timeline\] tab tiles bound to custom section](#)

Creating the [Timeline] tab tiles bound to custom section

Beginner

Easy

Medium

Advanced

Introduction

Starting from version 7.12.0 you can use the [Timeline] tab for quick analysis of customer cooperation, opportunity, case, etc. history in Creatio. This tab is available by default in the [Contacts], [Accounts], [Leads], [Opportunities] and [Cases] sections. General information about this tab functions is provided in the "[The \[Timeline\] tab](#)" article.

You can set up a tile using the *TimelinePageSetting* table settings as shown in the example with the base tile (see "[The \[Timeline\] tab](#)"). In such a case you will use the following for your tile:

- the default icon
- the *BaseTimelineItemView* and *BaseTimelineItemViewModel* base view and view model modules
- author field
- tile caption field
- message field

You can use one and the same tile for different sections if needed. However, we recommend to use the *TimelineTileSetting* table and set up your tiles for different sections.

The *TimelineTileSetting* table contains tile configurations that already exist in Creatio. The section, however, will only display the tiles indicated in the *TimelinePageSetting* table for this particular section.

For example, the *TimelineTileSetting* contains three pre-configured tiles: *Tasks*, *Leads* and *Calls*. The *TimelinePageSetting* table contains the *Tasks* and *Calls* tiles that are pre-configured for usage in the [Accounts] section, and only the *Calls* tile that is pre-configured for usage in the [Contacts] section. The *Leads* tile in this case will not be displayed in any section.

It is considered a good practice to differentiate the tile settings. We recommend to use the *TimelineTileSetting* table for setting up tiles, and the *TimelinePageSetting* – for adding the tiles to section timeline.

If you need to add the existing *Files* tile to a section, the "entitySchemaName" property of the *TimelinePageSetting* table should contain the entity schema name for files in the corresponding section configuration (for example, *AccountFile*, *ContactFile*, etc.). The object schema name (the "entitySchemaName" property) of the *TimelineTileSetting* table should always look as follows: "##ReferenceSchemaName##File".

To add a new pre-configured tile:

1. Add a new section (if needed).
2. Add a module schema to your custom package and determine the tile view class, bound to the new section. The class should be the inheritor of *BaseTimelineItemView*.
3. Add a module schema to your custom package and determine the tile view model class, bound to the new section. The class should be the inheritor of *BaseTimelineItemViewModel*.
4. Add a record with the tile view settings bound to the new section into the *TimelineTileSetting* database table.
5. In the *TimelinePageSetting* table add or edit the record enabling the tile display on the [Timeline] tab in the necessary section.

Case description

Display the tiles bound to the [Books] custom section on the [Timeline] tab of the [Accounts] section. The tiles should contain:

- icon
- name
- author
- book record date
- price
- ISBN number

A short book description should also be displayed when you deploy the tile.

Source code

You can download the package with the [Book] section implementation using the following [link](#). You can download the package with the tile module schema implementation using the following [link](#).

You should also edit the database tables to implement the case (see steps 4 and 5).

Case implementation algorithm

1. Adding a new [Books] section

Use the archive containing the needed function package to add the new [Books] section. Install the package via the marketplace application installation function from the *.zip-archive (see "**Transferring changes using packages export and import**").

You can also add the section via [section wizard](#).

The [Books] section will be available in the [General] workplace after you install the package (Fig.1).

Fig. 1. The [Books] section

The screenshot shows the Creatio application interface with the 'Books' section selected. The top navigation bar includes icons for search, filters, and a 'What can I do for you?' input field. On the right, there's a vertical sidebar with user profile and system-related icons. The main content area displays two book records in a table format:

JavaScript: The Definitive Guide: Activate Your Web Pages			
Author	Publisher	ISBN	Price
David Flanagan	Apress	978-0596805524	33.89

Pro C# 7: With .NET and .NET Core			
Author	Publisher	ISBN	Price
Andrew Troelsen	Apress	978-1484230176	56.99

You will also see a detail displaying the linked records from the [Books] section on the [Books] tab of the [Accounts] section record edit page (Fig.2).

Fig. 2. The [Books] detail in the [Accounts] section

The screenshot shows the Creatio application interface. On the left, there's a sidebar with fields for 'Name*' (Apress), 'Type' (Owner, Supervisor, Web), and a progress bar at 10% with an 'Enrich data' button. The main area has tabs for 'NEXT STEPS (0)', 'ACCOUNT INFO', 'BOOKS' (selected), 'CONTACTS AND STRUCTURE', 'TIMELINE', and 'CONNECT'. Under 'BOOKS', there's a list of books with columns for Name, ISBN, Author, and Publisher. Two books are listed: 'JavaScript: The Definitive Guide: Activate Your Web Pages' by David Flanagan (ISBN 978-0596805524) and 'Pro C# 7: With .NET and .NET Core' by Andrew Troelsen (ISBN 978-1484230176).

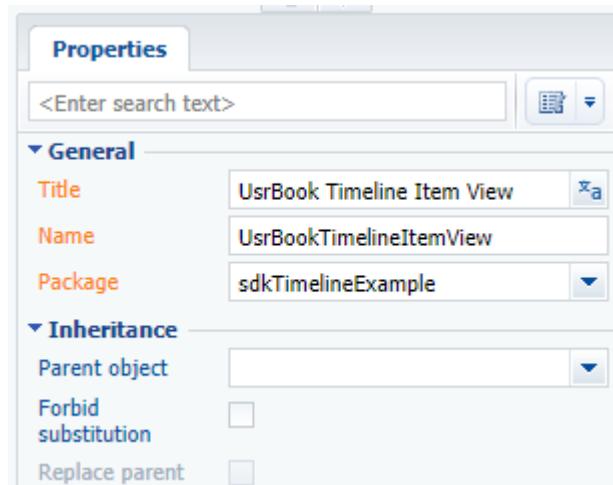
2. Adding a tile view module

Add a client module schema containing dependencies from the Timeline package to the custom package (see "Creating a custom client module schema").

For the created module schema specify (Fig. 3):

- [Name] – "UsrBookTimelineItemView"
- [Title] – "UsrBook Timeline Item View"

Fig. 3. Tile view module schema properties



Add the following module source code to the [Source code] tab of the schema:

```
// Defining the module and its dependencies.
define("UsrBookTimelineItemView", ["UsrBookTimelineItemViewResources",
"BaseTimelineItemView"], function() {
    // Defining the tile view class.
    Ext.define("Terrasoft.configuration.UsrBookTimelineItemView", {
        extend: "Terrasoft.BaseTimelineItemView",
        alternateClassName: "Terrasoft.UsrBookTimelineItemView",
        // Method returning the [UsrISBN] additional tile field configuration.
        getUsrISBNViewConfig: function() {
```

```
        return {
            // Field name.
            "name": "UsrISBN",
            // Field type - label.
            "itemType": Terrasoft.ViewItemType.LABEL,
            // Caption.
            "caption": {
                "bindTo": "UsrISBN"
            },
            // Visibility.
            "visible": {
                // Binding to the tile linked entity column.
                "bindTo": "UsrISBN",
                // Visibility setup.
                "bindConfig": {
                    // A field is visible if its value is populated.
                    "converter": "checkIsNotEmpty"
                }
            },
            // CSS field styles.
            "classes": {
                "labelClass": ["timeline-text-light"]
            }
        };
    },
    // Method returning the [UsrPrice] additional tile field configuration.
    getUsrPriceViewConfig: function() {
        return {
            "name": "UsrPrice",
            "itemType": Terrasoft.ViewItemType.LABEL,
            "caption": {
                "bindTo": "UsrPrice"
            },
            "visible": {
                "bindTo": "UsrPrice",
                "bindConfig": {
                    "converter": "checkIsNotEmpty"
                }
            },
            "classes": {
                "labelClass": ["timeline-item-subject-label"]
            }
        };
    },
    // Redefined method returning the [Message] tile field configuration.
    getMessageViewConfig: function() {
        // Receiving standard settings.
        var config = this.callParent(arguments);
        // Visibility setup. Visible if the tile is deployed.
        config.visible = {
            "bindTo": "IsExpanded"
        };
        return config;
    },
    // Redefined method returning general tile configuration.
    getBodyViewConfig: function() {
        // Receiving standard settings.
        var bodyConfig = this.callParent(arguments);
        // Adding additional field configurations.
        bodyConfig.items.unshift(this.getUsrISBNViewConfig());
        bodyConfig.items.unshift(this.getUsrPriceViewConfig());
        return bodyConfig;
    }
};
```

});
});

Here you can define the configuration of the [UsrISBN] and [UsrPrice] fields that are additionally displayed on the tab. The standard configuration is defined in the *BaseTimelineItemView* module.

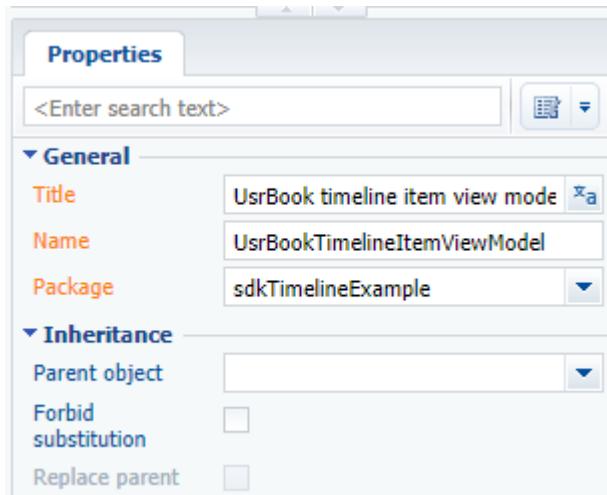
3. Adding a tile view model module

Add a client module schema containing dependencies from the Timeline package to the custom package (see ["Creating a custom client module schema"](#)).

For the created module schema specify (Fig. 3):

- [Name] – "UsrBookTimelineItemViewModel"
 - [Title] – "UsrBook timeline item view model"

Fig. 4. Module schema properties of the tile view model



Add the following module source code to the [Source code] tab of the schema:

```
define("UsrBookTimelineItemViewModel", ["UsrBookTimelineItemViewModelResources",
"BaseTimelineItemViewModel"],
function() {
    Ext.define("Terrasoft.configuration.UsrBookTimelineItemViewModel", {
        alternateClassName: "Terrasoft.UsrBookTimelineItemViewModel",
        extend: "Terrasoft.BaseTimelineItemViewModel"
    });
});
```

You define the *Terrasoft.configuration.UsrBookTimelineItemViewModel* class here. Since this class is defined as the inheritor of *Terrasoft.BaseTimelineItemViewModel*, it enables using all functions of the base class.

4. Adding the record with tile view settings to the TimelineTileSetting table

The TimelineTileSetting table is used to set up the timeline tile properties. The purpose of primary columns of this table is provided in table 2 of the "[The \[Timeline\] tab](#)" article.

Add a new record to the TimelineTileSetting table. You can add a new record via the following SQL query:

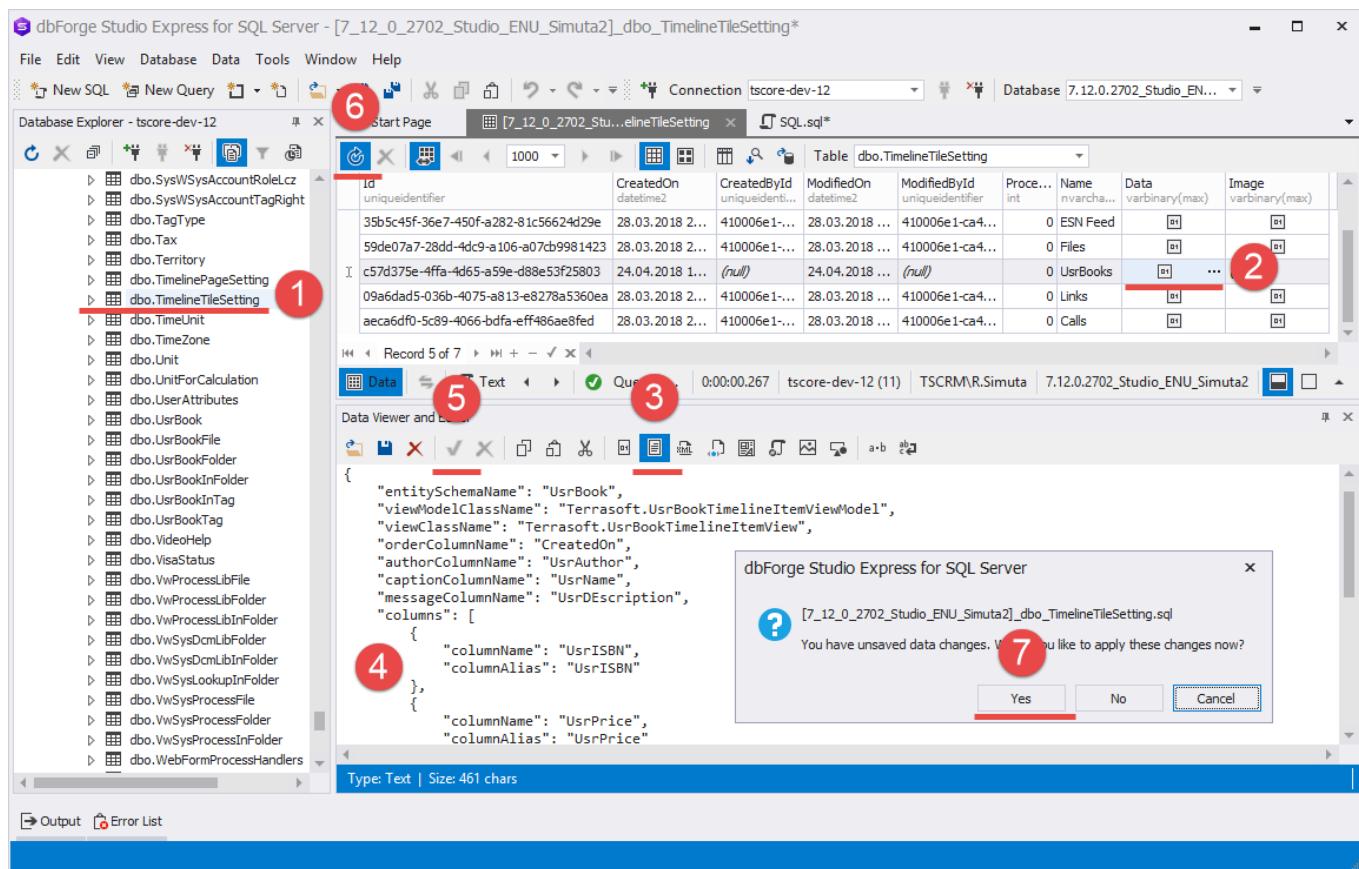
```
INSERT INTO TimelineTileSetting (CreatedOn, CreatedById, ModifiedOn, ModifiedById, Name, Data, Image)
VALUES (GETUTCDATE(), NULL, GETUTCDATE(), NULL, 'UserBooks!', NULL, NULL);
```

¹Significant in the Date and Month and significant in the month of birth and in sex.

2. Select the necessary record column and click the edit button.
3. Enter the text data display mode in the data editor.
4. Add necessary data.
5. Save the changes in the data editor.
6. Click the data update button.
7. Click OK in the popped up checkout window to apply the modifications.

This method is only good for the development environments deployed on-site. Since the modifications are implemented directly in the database, they are not bound to any package. That is why the modifications will not be implemented in the database if the package with the view models and the tile view models is installed into another application. For correct transfer of the developed functions you need to bind the SQL-scripts that implement the corresponding modifications in the database when installing the package.

Fig. 5. Editing data via the dbForge Studio Express for SQL Server



Add the following configuration object to the Data column using the above mentioned algorithm:

```
{
    "entitySchemaName": "UsrBook",
    "viewModelClassName": "Terrasoft.UsrBookTimelineItemViewModel",
    "viewClassName": "Terrasoft.UsrBookTimelineItemView",
    "orderColumnName": "CreatedOn",
    "authorColumnName": "UsrAuthor",
    "captionColumnName": "UsrName",
    "messageColumnName": "UsrDEscription",
    "columns": [
        {
            "columnName": "UsrISBN",
            "columnAlias": "UsrISBN"
        },
        {
            "columnName": "UsrPrice",
            "columnAlias": "UsrPrice"
        }
    ]
}
```

```
        "columnName": "UsrPrice",
        "columnAlias": "UsrPrice"
    }
]
}
```

You need to indicate the additional field array whose display is configured in the *UsrBookTimelineItemView* view model in addition to the primary fields inherited from the base tile (see step 2).

Add SVG-format data to the *Image* column to display the icon that corresponds to the section icon (Fig.6).

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 52 52" enable-background="new 0 0 52 52">
<path d="M46.072,31.384c-0.011-0.026-0.025-0.048-0.039-0.073c-0.036-0.064-0.077-0.125-0.123-0.182
    c-0.018-0.022-0.034-0.044-0.053-0.064c-0.034-0.036-0.068-0.07-0.105-0.104c-0.062-0.055-0.431-0.3-0.819-0.559
    c-1.958-1.307-7.465-4.978-9.424-6.284c-0.388-0.258-0.703-0.845-0.703-1.312v3.938c0-0.401-0.19-0.777-0.512-1.017
    c-0.322-0.239-0.739-0.311-1.122-0.193l15.015,8.254c-0.446,0.136-1.154,0.097-1.583-0.0861-1.094-0.467
    c-0.428-0.184-0.414-0.442,0.031-0.578l15.213-4.646c0.668-0.204,1.045-0.911,0.841-1.58s-0.912-1.047-1.58-0.841l7.507,5.961

C7.454,5.982,7.429,5.994,7.403,6.005c7.338,6.031,7.276,6.062,7.217,6.097c7.205,6.104,7.191,6.108,7.178,6.116
    c-0.015,0.01-0.026,0.025-0.041,0.035c7.081,6.191,7.03,6.236,6.982,6.284c-0.02,0.021-0.041,0.039-0.06,0.062

C6.864,6.412,6.813,6.485,6.77,6.562c6.716,6.659,6.683,6.748,6.658,6.838c6.651,6.864,6.648,6.89,6.642,6.916

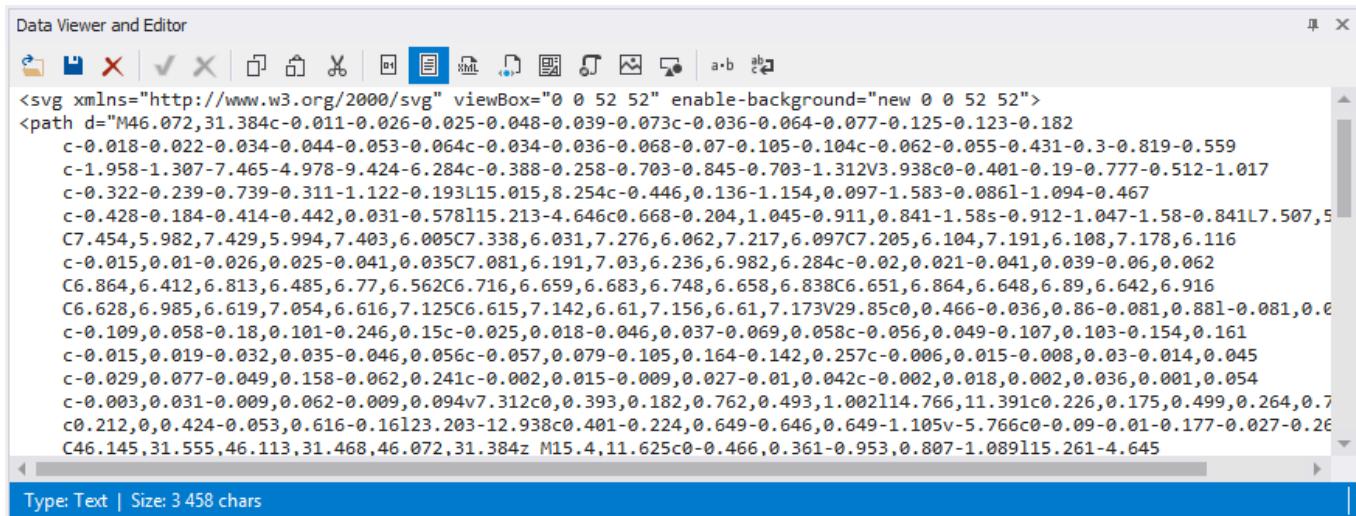
C6.628,6.985,6.619,7.054,6.616,7.125c6.615,7.142,6.61,7.156,6.61,7.173v29.85c0,0.466-0.036,0.86-0.081,0.881-0.081,0.036
    c-0.109,0.058-0.18,0.101-0.246,0.15c-0.025,0.018-0.046,0.037-0.069,0.058c-0.056,0.049-0.107,0.103-0.154,0.161
    c-0.015,0.019-0.032,0.035-0.046,0.056c-0.057,0.079-0.105,0.164-0.142,0.257c-0.006,0.015-0.008,0.03-0.014,0.045
    c-0.029,0.077-0.049,0.158-0.062,0.241c-0.002,0.015-0.009,0.027-0.01,0.042c-0.002,0.018,0.002,0.036,0.001,0.054
    c-0.003,0.031-0.009,0.062-
0.009,0.094v7.312c0,0.393,0.182,0.762,0.493,1.002l14.766,11.391c0.226,0.175,0.499,0.264,0.773,0.264
    c0.212,0,0.424-0.053,0.616-0.16123.203-12.938c0.401-0.224,0.649-0.646,0.649-1.105v-5.766c0-0.09-0.01-0.177-0.027-0.261
    C46.145,31.555,46.113,31.468,46.072,31.384z M15.4,11.625c0-0.466,0.361-0.953,0.807-1.089l15.261-4.645
    c0.446-0.136,0.807,0.132,0.807,0.598v14.63c0,0.467-0.314,0.635-0.702,0.376l-1.127-0.752c-0.361-0.24-0.819-0.278-1.216-0.104
    l-13.059,5.805c-0.426,0.189-0.771-0.034-0.771-0.501c15.4,25.943,15.4,11.625,15.4,11.625z M28.851,23.579
    c0.425-0.189,1.085-0.134,1.473,0.125l11.43,7.62c0.388,0.259,0.368,0.644-0.045,0.861-18.404,9.662
    c-0.412,0.216-1.047,0.163-1.418-0.1211-11.789-9.001c-0.371-0.283-0.326-0.665,0.1-0.854l28.851,23.579z M9.142,9.932
    c0-0.466,0.348-0.695,0.776-0.512l12.174,0.929c0.429,0.183,0.776,0.708,0.776,1.175v2.158c-1.57-0.068-2.894-0.916-3.727-1.61
    L9.142,9.932l9.142,9.932z
M9.142,13.152c0.931,0.671,2.22,1.323,3.727,1.372v7.633c-1.57-0.066-2.894-0.915-3.727-1.609
    C9.142,20.548,9.142,13.152,9.142,13.152z
```

```

M9.142,21.627c0.931,0.671,2.22,1.323,3.727,1.372v3.992c0,0.466-0.35,0.985-0.782,1.16
  1-2.163,0.876c-0.432,0.175-0.782-0.061-0.782-0.527v21.627z
M43.666,36.101c0,0.467-0.33,1.027-0.737,1.255l22.578,48.702
  c-0.407,0.228-1.036,0.18-1.405-0.104l8.897,39.127c-0.369-0.284-0.668-0.893-0.668-
1.358v-2.444c0-0.466,0.3-0.614,0.671-0.332
  112.764,9.748c0.225,0.171,0.496,0.26,0.768,0.26c0.201,0,0.403-0.048,0.588-
0.146l19.899-10.447
  c0.413-0.217,0.747-0.015,0.747,0.452v36.101z" style="fill:#6c91de;"/>
<path d="M33.81,34.064c0.072,0.049,0.155,0.073,0.239,0.073c0.072,0,0.145-0.018,0.209-
0.055l4.505-2.575
  c0.126-0.072,0.207-0.204,0.212-0.349c0.006-0.146-0.063-0.283-0.183-0.365l-9.011-
6.192c-0.118-0.08-0.268-0.097-0.399-0.042
  1-5.157,2.123c-0.143,0.059-0.243,0.191-0.259,0.346c-
0.017,0.154,0.053,0.304,0.181,0.392l33.81,34.064z M29.492,25.426
  18.269,5.682l-3.692,2.111-8.803-6.052l29.492,25.426z" style="fill:#6c91de;"/>
</svg>

```

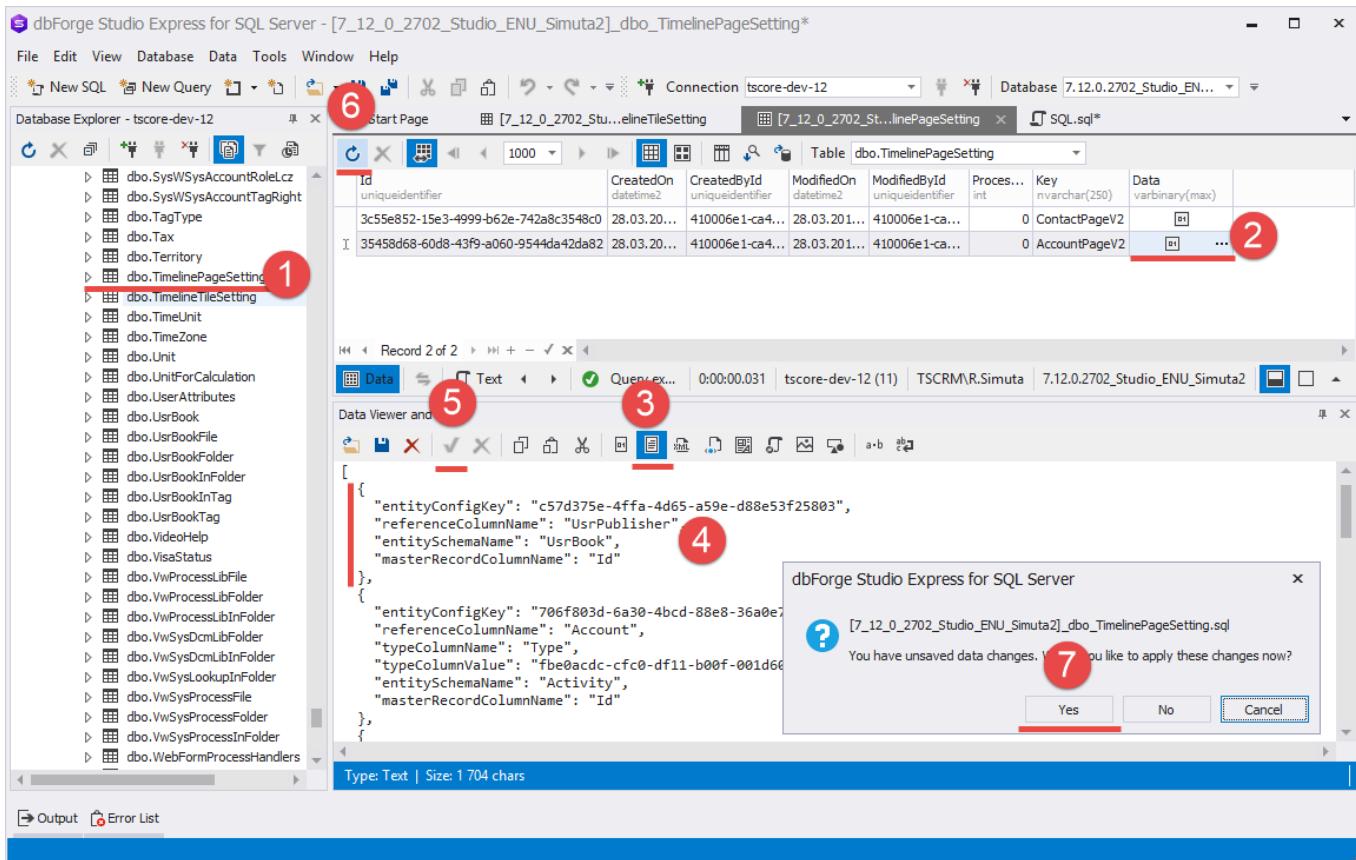
Fig. 6. Editing the Image column data via the dbForge Studio Express for SQL Server



5. Editing the record that enables the tile display on the [Timeline] tab of the account page in the TimelinePageSettings table.

For the [Accounts] section there already exists a record in the *TimelineTileSettings* table with settings of tiles bound to other sections. This is the record containing the "AccountPageV2" value in the *Key* column (Fig.7).

Fig. 7. Editing the Image column data via the dbForge Studio Express for SQL Server



Since there are several tiles used on the [Accounts] section page timeline, the array of configuration objects enabling the corresponding tile is stored in the Data column.

Using the algorithm mentioned in step 4, change the configuration object array by adding a new record to it.

```
[
  {
    "entityConfigKey": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
    "referenceColumnName": "UsrPublisher",
    "entitySchemaName": "UsrBook",
    "masterRecordColumnName": "Id"
  },
  ...
]
```

Here the "entityConfigKey" property: "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx" should contain the TimelineTileSettings table record identifier created on step 4. In our case it is the "c57d375e-4ffa-4d65-a59e-d88e53f25803" value (Fig.5 and Fig.7).

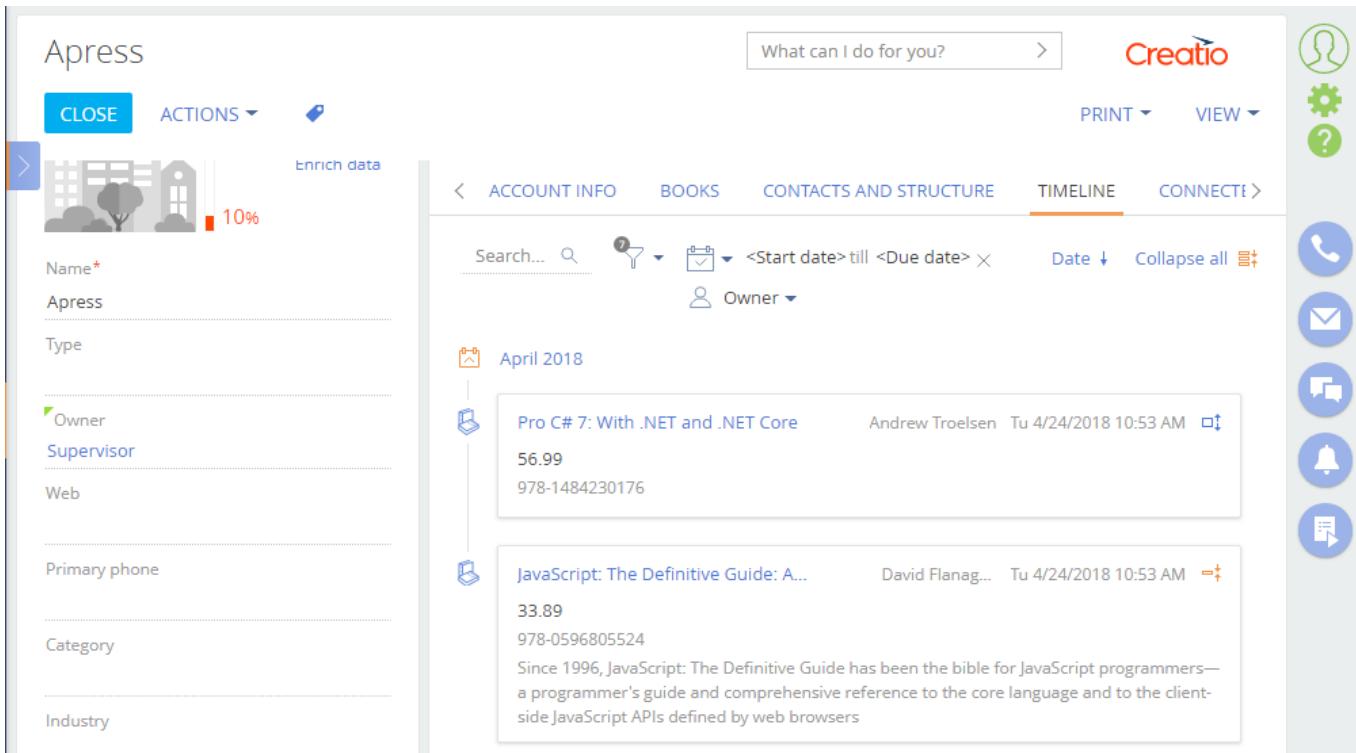
The "entityConfigKey" should be obligatory indicated. It should match the *Id* column value of the record containing settings of the necessary tile in the *TimelineTileSettings* table.

Since the identifiers of the added records are generated at random, the generated identifier in your database will be different from the one we have in our case, when you repeat step 4.

Be careful when modifying the *Data* column value. Incorrect modifications can disrupt the operation of all existing timeline tiles in a section.

As a result of case implementation you will have the tiles bound to the [Book] custom section displayed on the [Timeline] tab of the [Accounts] section page. These tiles contain all the fields we described in our case conditions. The short book description will only be displayed when you deploy the tile (Fig.8).

Fig. 8. Case result



The [Connected entity profile] control

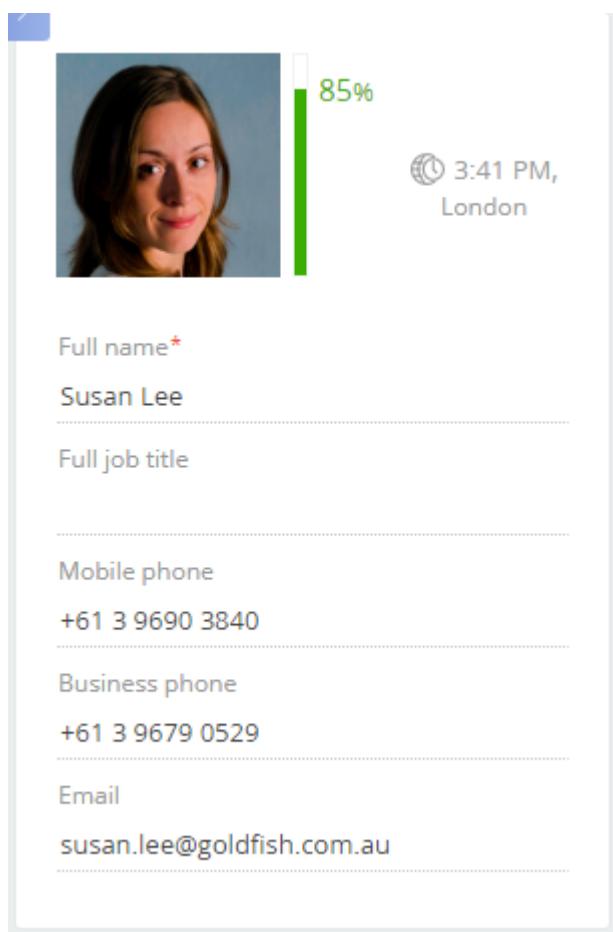
Beginner Easy Medium **Advanced**

General Information

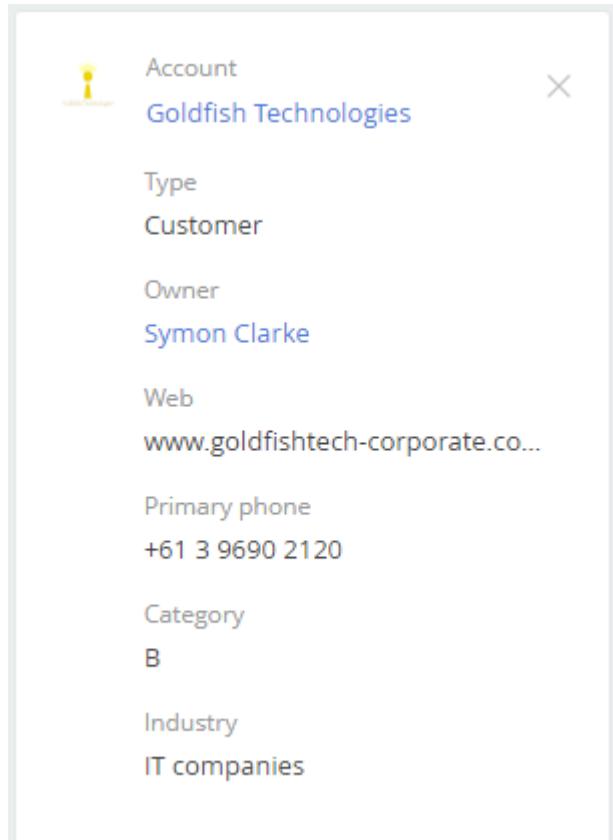
The [Connected entity profile] control (the *Profile* class) is a configuration module (information block) which is populated with information about the connected entity when the page is loading. This element is used in the system as a connected record profile on the section entry editing page. For example, if you open the [Contact] editing page in the [Profile] element (Fig. 1), the contact profile and the associated account information will be displayed (communication via the [Account] column of the [Contact] object, Fig. 2). Learn more about record profiles and connected records in the “[Record pages](#)”.

The parent class for *Profile* is *BaseProfileSchema* – a basic schema for creating any related record profiles in the system.

(Fig. 1). [Contact] object profile



(Fig. 2). Connected contact profile of the related [Account] entity



The parent class for *Profile* is *BaseProfileSchema* – a basic schema for creating any related record profiles in the system.

All profiles are inherited from *BaseProfileSchema*.

The *BaseProfileSchema* schema implements the ability to display any set of fields of a related entity, as well as any number of different modules.

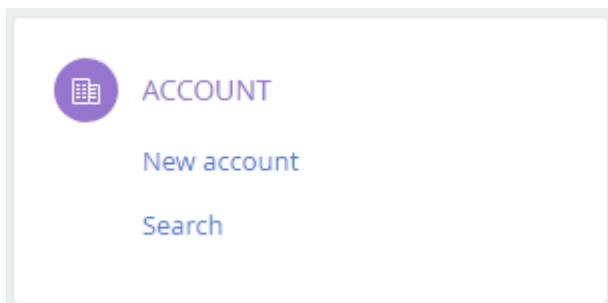
A view is described by the *diff* property (similar to the editing page description process). While embedding a module to the editing page, specify the parameter *masterColumnName* in the module attributes. The parameter stores the name of the column used to connect the profile to the main editing page diagram. *Profile* will download the data based on this column value.

At the initialization stage, the profile object sends a message to *GetColumnInfo* to obtain additional information about its connected column (filters, header, etc.). Then it requests the connected column value, and if it is full, the data for this record is initialized. When you clear the field or change the value, the data in the profile object is reinitialized.

If the profile is empty, i.e. the entry in the link field is not selected, then the name of the field through which the connection is made and the two actions are displayed in it (Fig. 3).

- [Add account] – create a new entry in the link field lookup.
- [Select] – select an existing entry from the list.

(Fig. 3). An empty connected entity profile



When you select an existing related entity, all business logic (defined on the editing page and connected to the referring entity attribute) is superimposed on the lookup. Filtering, query column settings, etc. are retained.

If the attribute is removed from the layout of the editing page, all logic will be lost along with it.

The BaseProfileSchema

The interaction interface is implemented by the standard messages mechanism. The following messages are used:

- *OpenCard* – opens a page for adding an entry using the standard mechanism.
- *UpdateCardProperty* – updates the value of the editing page attribute.
- *GetColumnInfo* – returns communication field information.
- *GetColumnsValues* – returns the values of the requested editing page columns.
- *GetEntityColumnChanges* – subscription to edit the editing page data.
- *CardModuleResponse* – the result of adding a new record through the profile.

The visual content of a profile (buttons, links, modules) is defined in the *diff* modification array.

Profile configuration case

```
define("ContactProfileSchema", ["ProfileSchemaMixin"],  
function () {  
    return {  
        // Name of object schema.  
        entitySchemaName: "Contact",  
        // Mixins.  
        mixins: {  
            // Mixin with functions for obtaining icons and profile pictures.  
            ProfileSchemaMixin: "Terrasoft.ProfileSchemaMixin"  
        }  
    };  
});
```

```
},
// The diff modification array.
diff: /**SCHEMA_DIFF*/ [
{
    // Inserting.
    "operation": "insert",
    // Entity name.
    "name": "Account",
    // The name of the parent element in which to insert.
    "parentName": "ProfileContentContainer",
    // The property of the parent element with which the operation is
performed.
    "propertyName": "items",
    // The values of the inserted item.
    "values": {
        // Binding the Account property to the Contact object value.
        "bindTo": "Account",
        // Layout configuration. Element positioning.
        "layout": {
            "column": 5,
            "row": 1,
            "colSpan": 19
        }
    }
}
// ...Other modification array configuration objects.
] /**SCHEMA_DIFF*/
};

});
```

An example of embedding a profile to an editing page.

```
//Defining the editing page schema and its dependencies.
define("ContactPageV2", ["BaseFiltersGenerateModule", "BusinessRuleModule",
"ContactPageV2Resources",
    "ConfigurationConstants", "ContactCareer",
"DuplicatesSearchUtilitiesV2"],
function (BaseFiltersGenerateModule, BusinessRuleModule, resources,
ConfigurationConstants, ContactCareer) {
    return {
        entitySchemaName: "Contact",
        // Modules used .
        modules: /**SCHEMA_MODULES*/{
            // Account profile module.
            "AccountProfile": {
                // Profile configuration.
                "config": {
                    // Schema name.
                    "schemaName": "AccountProfileSchema",
                    // A characteristic indicating that circuit configuration is
initialized.
                    "isSchemaConfigInitialized": true,
                    // A characteristic indicating that HistoryState is not used.
                    "useHistoryState": false,
                    // Profile parameters.
                    "parameters": {
                        // View model configuration.
                        "viewModelConfig": {
                            // The name of the connected entity column.
                            masterColumnName: "Account"
                        }
                    }
                }
            }
        }
    }
}
```

```
        }
    }
}/**SCHEMA_MODULES*/,
// Modification array.
diff: /**SCHEMA_DIFF*/ [
{
    "operation": "insert",
    "parentName": "LeftModulesContainer",
    "propertyName": "items",
    // Profile name.
    "name": "AccountProfile",
    // Values.
    "values": {
        // Element type - module.
        "itemType": Terrasoft.ViewItemType.MODULE
    }
}
]/**SCHEMA_DIFF*/
);
});
```

The BaseMultipleProfileSchema profile schema

In addition to the *BaseProfileSchema* base profile schema, there are additional schemas that implement specific functionality of user profiles.

The profile described by the *BaseMultipleProfileSchema* schema can contain any number of profiles and switch between them by using the logic of selected values from several directories. The main difference from the base profile is the ability to embed other profiles to the current profile. In this case, the built-in profiles can communicate with each other via messages. Otherwise, the way it works with the editing page is similar to that of the base profile.

The *BaseMultipleProfileSchema* profiles must be inherited from the *BaseRelatedProfileSchema* base profile schema, which can be dependent or embedded to other profiles.

Example of the BaseMultipleProfileSchema client profile module.

```
// Defining a profile.
define("ClientProfileSchema", ["ProfileSchemaMixin"],
function () {
    return {
        // Mixins.
        mixins: {
            ProfileSchemaMixin: "Terrasoft.ProfileSchemaMixin"
        },
        // Attributes.
        attributes: {
            // Contact profile visibility.
            "IsVisibleContactProfile": {
                dataValueType: this.Terrasoft.DataValueType.BOOLEAN,
                value: true
            }
        },
        // Methods.
        methods: {
            // Date-marker. Needed for automatic tests.
            getProfileModuleContainerDataMarker: function () {
                return "client-profile-module-container";
            },
            // Returns the header.
            getBlankSlateHeaderCaption: function () {
                return this.get("Resources.Strings.Client");
            },
            // Returns the warning icon.
```

```
getWarningIcon: function () {
    return this.getImageUrlByResourceKey("Resources.Images.WarningIcon");
},
// Checks a warning display.
getIsVisibleWarning: function () {
    var masterColumnNames = this.get("MasterColumnNames");
    if (!masterColumnNames) {
        return false;
    }
    var masterColumnValues = masterColumnNames.filter(function
(columnName) {
        var value = this.get(columnName);
        return !this.Ext.isEmpty(value);
    }, this);
    return masterColumnValues.length > 1;
},
// The event handler of the profile column change.
onProfileColumnChanged: function () {
    this.set("IsVisibleContactProfile", !this.getIsVisibleWarning());
    return this.callParent(arguments);
},
// The column change event handler.
onColumnChanged: function () {
    this.callParent(arguments);
    this.set("IsVisibleContactProfile", !this.getIsVisibleWarning());
},
// The [Clear] button hint.
getClearButtonHint: function () {
    var clearActionCaption =
this.get("Resources.Strings.ClearButtonCaption");
    var masterColumnCaption = this.get("Resources.Strings.Client");
    return this.Ext.String.format("{0} {1}", clearActionCaption,
masterColumnCaption);
}
},
// Schema modules.
modules: /**SCHEMA_MODULES*/{
    // Built-in client profile of the account.
    "AccountClientProfile": {
        // Profile configuration.
        "config": {
            "schemaName": "ClientAccountProfileSchema",
            "isSchemaConfigInitialized": true,
            "useHistoryState": false,
            "parameters": {
                "viewModelConfig": {
                    masterColumnName: "Account"
                }
            }
        }
    },
    // Client contact embedded profile.
    "ContactClientProfile": {
        "config": {
            "schemaName": "ClientContactProfileSchema",
            "isSchemaConfigInitialized": true,
            "useHistoryState": false,
            "parameters": {
                "viewModelConfig": {
                    masterColumnName: "Contact"
                }
            }
        }
    }
}
```

```
        }
    }
}/**SCHEMA_MODULES*/,
// Profile view modifications.
diff: /**SCHEMA_DIFF*/ [
{
    "operation": "remove",
    "name": "ProfileIcon"
},
{
    "operation": "remove",
    "name": "ProfileHeaderContainer"
},
{
    "operation": "insert",
    "name": "ClientProfilesContainer",
    "parentName": "ProfileContentContainer",
    "propertyName": "items",
    "values": {
        "itemType": this.Terrasoft.ViewItemType.CONTAINER,
        "items": [],
        "layout": {
            "column": 0,
            "row": 0,
            "colSpan": 24,
            "rowSpan": 24
        }
    }
},
{
    "operation": "insert",
    "name": "WarningIcon",
    "parentName": "ClientProfilesContainer",
    "propertyName": "items",
    index: 0,
    "values": {
        "getSrcMethod": "getWarningIcon",
        "readonly": true,
        "generator": "ImageCustomGeneratorV2.generateSimpleCustomImage",
        "visible": { "bindTo": "getIsVisibleWarning" },
        "classes": {
            "wrapClass": ["warning-icon"]
        },
        "hint": { "bindTo": "Resources.Strings.WarningMessage" }
    }
},
{
    "operation": "insert",
    "parentName": "ClientProfilesContainer",
    "propertyName": "items",
    "name": "AccountClientProfile",
    "values": {
        "itemType": this.Terrasoft.ViewItemType MODULE
    }
},
{
    "operation": "insert",
    "parentName": "ClientProfilesContainer",
    "propertyName": "items",
    "name": "ContactClientProfile",
    "values": {
        "itemType": this.Terrasoft.ViewItemType MODULE,
```

```
        "visible": { "bindTo": "IsVisibleContactProfile" }
    }
}
]/**SCHEMA_DIFF*/
);
);
)
```

Example of a built-in BaseRelatedProfileSchema client profile

```
define("ClientContactProfileSchema", ["ProfileSchemaMixin"],
function () {
    return {
        // Schema object name.
        entitySchemaName: "Contact",
        // Mixins.
        mixins: {
            ProfileSchemaMixin: "Terrasoft.ProfileSchemaMixin"
        },
        // Methods.
        methods: {
            getProfileHeaderCaption: function () {
                return this.get("Resources.Strings.ProfileHeaderCaption");
            }
        },
        // Modifications array.
        diff: /**SCHEMA_DIFF*/ [
            {
                "operation": "insert",
                "name": "Account",
                "parentName": "ProfileContentContainer",
                "propertyName": "items",
                "values": {
                    "bindTo": "Account",
                    "enabled": false,
                    "layout": {
                        "column": 5,
                        "row": 1,
                        "colSpan": 19
                    }
                }
            },
            {
                "operation": "insert",
                "name": "Job",
                "parentName": "ProfileContentContainer",
                "propertyName": "items",
                "values": {
                    "bindTo": "Job",
                    "enabled": false,
                    "layout": {
                        "column": 5,
                        "row": 2,
                        "colSpan": 19
                    }
                }
            },
            {
                "operation": "insert",
                "name": "Type",
                "parentName": "ProfileContentContainer",
                "propertyName": "items",
                "values": {
                    "bindTo": "Type"
                }
            }
        ]
    }
});
```

```
        "values": {
            "bindTo": "Type",
            "enabled": false,
            "layout": {
                "column": 5,
                "row": 3,
                "colSpan": 19
            }
        }
    },
{
    "operation": "insert",
    "name": "MobilePhone",
    "parentName": "ProfileContentContainer",
    "propertyName": "items",
    "values": {
        "bindTo": "MobilePhone",
        "enabled": false,
        "layout": {
            "column": 5,
            "row": 4,
            "colSpan": 19
        }
    }
},
{
    "operation": "insert",
    "name": "Phone",
    "parentName": "ProfileContentContainer",
    "propertyName": "items",
    "values": {
        "bindTo": "Phone",
        "enabled": false,
        "layout": {
            "column": 5,
            "row": 5,
            "colSpan": 19
        }
    }
},
{
    "operation": "insert",
    "name": "Email",
    "parentName": "ProfileContentContainer",
    "propertyName": "items",
    "values": {
        "bindTo": "Email",
        "enabled": false,
        "layout": {
            "column": 5,
            "row": 6,
            "colSpan": 19
        }
    }
}
]/**SCHEMA_DIFF*/
};

});
```

An example of embedding a client profile module to the editing page.

```
// Editing page modules.
```

```
modules: /**SCHEMA_MODULES*/ {
    // Module name.
    "ClientProfile": {
        // Configuration.
        "config": {
            // A characteristic indicating that circuit configuration is
            initialized.
            "isSchemaConfigInitialized": true,
            // A characteristic indicating that HistoryState is not used.
            "useHistoryState": false,
            // Schema name.
            "schemaName": "ClientProfileSchema",
            // Parameters.
            "parameters": {
                // View Model Configuration.
                "viewModelConfig": {
                    // Connected entity column name.
                    "masterColumnName": "Client"
                }
            }
        }
    }
}/**SCHEMA_MODULES*/,
// The diff modification array.
diff: /**SCHEMA_DIFF*/ [
{
    "operation": "insert",
    // Profile name.
    "name": "ClientProfile",
    "parentName": "LeftModulesContainer",
    "propertyName": "items",
    // Values.
    "values": {
        // Element type - module.
        "itemType": Terrasoft.ViewItemType.MODULE
    }
}
]/**SCHEMA_DIFF*/
```

Interface control tools

Contents

- **Locking edit page fields**
- **Feature Toggle. Mechanism of enabling and disabling functions**

Locking edit page fields

Beginner Easy Medium **Advanced**

Introduction

During the development of the Creatio custom functions, you may need to lock all fields and details on an edit page under certain conditions. The mechanism of locking edit page fields lets you implement page business logic without creating additional business rules.

The locking mechanism is available in version 7.11.1 and up.

You can disable the locking of edit page fields using the *CompleteCardLockout* option on the Feature Toggle page (see “**Feature Toggle. Mechanism of enabling and disabling functions**”). The Feature Toggle page is available via the following URL: [..../o/Nui/ViewModule.aspx#BaseSchemaModuleV2/FeaturesPage](#). For example,

<https://myserver.com/CreatioWebApp/o/Nui/ViewModule.aspx#BaseSchemaModuleV2/FeaturesPage>

As a result of applying the locking mechanism on an edit page, all fields and details will become locked. If the field has a binding for the *enabled* property in the *diff* array element or the business rule, the mechanism will not lock this field. Locking a detail will hide buttons and menu items for performing operations with the detail records. A locked detail with an editable list will still feature an ability to access the object page, however, all fields on it will be locked.

The locking mechanism is intended for locking details with a regular list and an editable list. To ensure the correct operation of the mechanism for details with editable fields, create a replacement schema for this detail and control the availability of fields using the *IsEnabled* attribute.

To enable the locking mechanism, set the source code of the edit page to *false* for the *IsModelItemsEnabled* model attribute:

```
this.set("IsModelItemsEnabled", false);
```

Alternatively, set the default value for the attribute:

```
"IsModelItemsEnabled": {
    dataType: Terrasoft.DataValueType.BOOLEAN,
    value: true,
    dependencies: [
        columns: ["PaymentStatus"],
        methodName: "setCardLockoutStatus"
    ]
}
```

Additionally, to operate the locking mechanism on a specific edit page in the *diff* array of this page, specify the *DisableControlsGenerator* generator for the containers in which you want to lock fields. Therefore, to lock all fields of an edit page, specify the global *CardContentWrapper* container:

```
diff: /**SCHEMA_DIFF*/ [
{
    "operation": "merge",
    "name": "CardContentWrapper",
    "values": {
        "generator": "DisableControlsGenerator.generatePartial"
    }
}
]/**SCHEMA_DIFF*/
```

In versions 7.13.0 and below, an attempt to specify a generator value will trigger an error when opening the edit page in the Section Wizard.

To fix the error:

- Find out the name of the group containing all employees. To do this, use the following DB query.

```
select Name from SysAdminUnit
```

- Run the script provided below on your database. Note that the *@allEmployeeGroupName* field must be filled in with the name of the group containing all employees of your organization.

```
DECLARE @allEmployeeGroupName nvarchar(max) = 'All employees';
DECLARE @featureName nvarchar(max) = 'PageDesignerCustomGeneratorFix';
DECLARE @featureStatus bit = 1;

IF (NOT EXISTS (SELECT NULL FROM Feature WHERE Code = @featureName))
BEGIN
    INSERT INTO Feature (Name, Description, Code, ProcessListeners)
    VALUES (@featureName, @featureName, @featureName, 0)
END
IF (EXISTS (SELECT NULL FROM AdminUnitFeatureState
            WHERE FeatureId = (SELECT Id FROM Feature WHERE Code = @featureName) AND
                  SysAdminUnitId = (SELECT Id FROM SysAdminUnit WHERE Name =
@allEmployeeGroupName)) )
```

```

BEGIN
    UPDATE AdminUnitFeatureState SET FeatureState = @featureStatus WHERE FeatureId =
    (SELECT Id FROM Feature WHERE Code = @featureName) AND
                                SysAdminUnitId = (SELECT Id FROM SysAdminUnit WHERE
Name = @allEmployeeGroupName)
END
ELSE
BEGIN
    INSERT INTO AdminUnitFeatureState (ProcessListeners, SysAdminUnitId,
FeatureState, FeatureId) VALUES
    (
        0,
        (SELECT Id FROM SysAdminUnit WHERE Name =
@allEmployeeGroupName),
        @featureStatus,
        (SELECT Id FROM Feature WHERE Code = @featureName)
    )
END

```

The script introduces an extra feature into the system, *PageDesignerCustomGeneratorFix*, and enables it for the “All employees” user group.

- Examine the *generateCustomItem()* method in the *ViewModelSchemaDesignerViewGenerator* module. The method must look like this:

```

generateCustomItem: function(config) {
    if (Terrasoft.Features.getIsEnabled("PageDesignerCustomGeneratorFix")) {
        if (config) {
            delete config.generator;
        }
        return this.generateStandardItem(config);
    } else {
        return this.generateLabel({
            caption: config.name
        });
    }
}

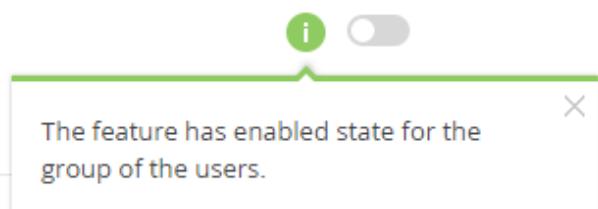
```

If the *generateCustomItem()* method is different from the one specified above, you must replace the *ViewModelSchemaDesignerViewGenerator* class and change the *generateCustomItem()* method.

- On the newly added **FeaturesPage**, check the value of the *PageDesignerCustomGeneratorFix* setting. When the mouse pointer hovers on , a notification [The feature has enabled state for the group of the users] must pop up.

Fig. 1. – Notification popup

PageDesignerCustomGeneratorFix



Locking exceptions

It is possible to disable locking for some fields and details. To do this, override the *getDisableExclusionsDetailSchemaNames()* and *getDisableExclusionsColumnTags()* methods. These methods return lists of fields and details that should not be blocked by the mechanism. The implementation of methods is available below:

```

getDisableExclusionsColumnTags: function() {
    return ["SomeField"];
}

```

```
getDisableExclusionsDetailSchemaNames: function() {
    return ["SomeDetailV2"]
}
```

More complex exception logic can be implemented by overriding the `isModelItemEnabled()` method for fields and the `isDetailEnabled()` method for details. These methods are called for each field and detail. They receive the name and return the availability signal of the field or detail. The implementation of methods is available below:

```
isModelItemEnabled: function(fieldName) {
    var condition = this.get("SomeConditionAttribute");
    if (fieldName === "ExampleField" || condition)) {
        return true;
    }
    return this.callParent(arguments);
}

isDetailEnabled: function(detailName) {
    if (detailName === "ExampleDetail") {
        var exampleDate = this.get("Date");
        var dateNow = new Date(this.Ext.Date.now());
        var condition = this.Ext.Date.isDate(exampleDate) && exampleDate >= dateNow;
        return condition;
    }
    return this.callParent(arguments);
}
```

Feature Toggle. Mechanism of enabling and disabling functions

Beginner

Easy

Medium

Advanced

Introduction

Feature toggle is a software development technique that provides support for connecting additional functionality in a running application. This allows to use continuous integration, keep the application working and hide the functionality that is under development process.

The main idea is that there is a block of additional functionality (often not fully implemented) in the source code and conditional operator that defines if the functionality connected.

Mechanism of enabling and disabling functions

The FeaturesPage page is used to add, enable and disable functions. The page address is:

[Application address]/0/Nui/ViewModule.aspx#BaseSchemaModuleV2/FeaturesPage

Example:

<http://mycreatio.com/0/Nui/ViewModule.aspx#BaseSchemaModuleV2/FeaturesPage>

To add new functions specify its code, name and description and click the [Create feature] button (Fig. 1).

Fig. 1. Interface of adding new feature

Features

Create feature

Feature code* UsrNewFeature

Feature name New feature

Feature description Some feature description

CREATE FEATURE

SAVE CHANGES

Use corresponding checkbox to enable or disable new features (Fig. 2.1). To apply changes click the [Save changes] button (Fig. 2.2).

Fig. 2. Enable/disable feature



Storing the functionality data in the database

A list of functionality available for enabling/disabling is stored in the *Feature* table of the application database. Table is empty by default. Main *Feature* table fields are given in table 1.

Table 1. Main *Feature* table fields

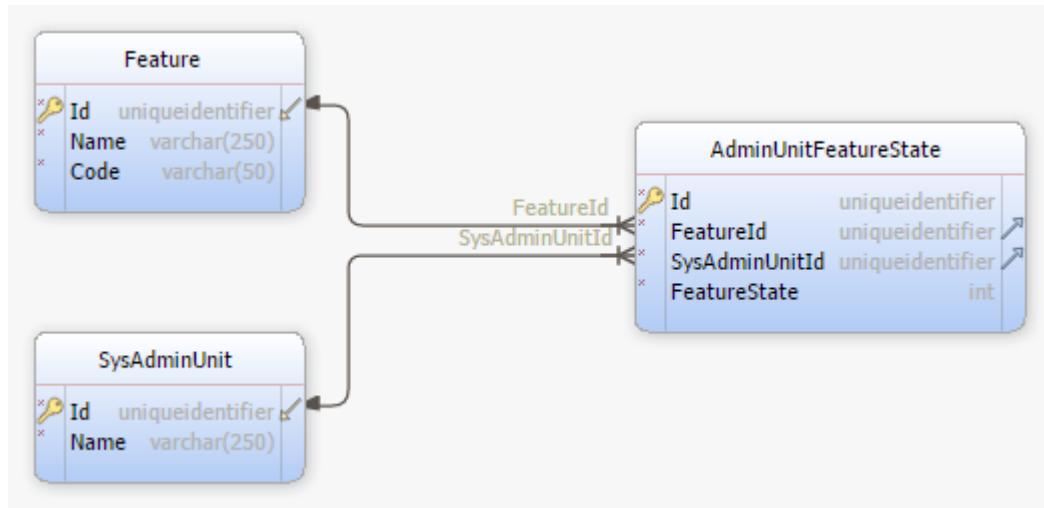
Name	Type	Description
<i>Id</i>	<i>uniqueidentifier</i>	Unique Id of the record
<i>Name</i>	<i>varchar(250)</i>	Functionality name.
<i>Code</i>	<i>varchar(50)</i>	Functionality code.

Information about functionality state (enabled/disabled) stored in the *FeatureState* field of the *AdminUnitFeatureState* table (Fig.1). The *AdminUnitFeatureState* table binds the *Feature* and *SysAdminUnit* tables where users and system user groups are defined. Main *AdminUnitFeatureState* table fields are given in table 2.

Table 2. Main *AdminUnitFeatureState* table fields

Name	Type	Description
<i>Id</i>	<i>uniqueidentifier</i>	Unique Id of the record
<i>FeatureId</i>	<i>uniqueidentifier</i>	Unique Id of the functionality record.
<i>SysAdminUnitId</i>	<i>uniqueidentifier</i>	Unique Id of the user record.
<i>FeatureState</i>	<i>int</i>	Functionality state. 1 – enabled, 0 – disabled.

Fig. 1 Diagram of table relationships



Defining the new functionality in the source code.

To implement the new functionality to the source code it should be defined in the block of the conditional operator that will check the state of the functionality connection (FeatureState).

Client side JavaScript

A conditional template for defining additional functionality in the source code:

```

// The method defining the additional functionality.
someMethod: function() {
    // Functionality connection check.
    if (Terrasoft.Features.getIsEnabled("functionality code")) {
        // Implementation of additional functionality.
        ...
    }
    // Method Implementation
    ...
}
    
```

The `getIsFeatureEnabled` method is implemented in the `BaseSchemaViewModel` base schema view model. Therefore, the `Terrasoft.Features.getIsEnabled` method can be replaced with `this.getIsFeatureEnabled("functionality code")`.

Refresh the browser page after connecting the new functionality to enable it in the client code and load it in the browser.

Server side C#

A set of extending methods of the [UserConnection](#) class was implemented to use the *Feature toggle* in the source code schemas on the server side in the `Terrasoft.Configuration.FeatureUtilities` class. A list of the extended methods is given in the Table 3. The `FeatureState` functionality states are enumerated in the same class.

Table 2. Main methods of the `DataManager` class

Methods.	Parameters	Description
<code>int GetFeatureState(this UserConnection source, string code)</code>	<code>code</code> – functionality code.	Returns functionality state.
<code>Dictionary <string, int> GetFeatureStates(this UserConnection</code>	No.	Returns the state of all functionality.

<code>source)</code>		
<code>void SetFeatureState(</code>	<code>code – functionality code;</code>	Returns functionality state.
<code>this UserConnection</code>	<code>state – functionality state (0/1);</code>	
<code>userConnection,</code>	<code>forAllUsers – a flag of enabling the</code>	
<code>string code, int state,</code>	<code>functionality for all users.</code>	
<code>bool forAllUsers =</code>		
<code>false)</code>		
<code>void CreateFeature(</code>	<code>code – functionality code;</code>	Creates new functionality.
<code>this UserConnection</code>	<code>name – functionality name;</code>	
<code>source, string code,</code>	<code>Description – functionality description.</code>	
<code>string name, string</code>		
<code>description)</code>		
<code>bool</code>	<code>code – functionality code.</code>	Checks if the functionality connected.
<code>GetIsFeatureEnabled(</code>		
<code>this UserConnection</code>		
<code>source, string code)</code>		

A conditional template for defining additional functionality in the source code:

```
...
// A namespace in which the ability to switch additional
// functionality is defined.
using Terrasoft.Configuration;

...
// The method in which additional functionality will be defined.
public void AnyMethod() {
    // Check if functionality is enabled.
    if (UserConnection.GetIsFeatureEnabled("functionality code")) {
        // Implementation of additional functionality.
    }
    // Method implementation.
    ...
}
```

Setting the value of functionality state is executed by call of the SetFeatureState method:

```
UserConnection.SetFeatureState("functionality code", FeatureState);
```

User services and tools

Contents

- **Report setup**
- **Phone integration**
- **Creatio marketing**
- **Sync Engine synchronization mechanism**
- **Scheduler setup**
- **Self-service Portal**
- **Machine learning service**
- **Sending emails**
- **Contact data enrichment from emails**
- **Static content bundling service**

Report setup

Contents

- **Introduction**
- **Setting up reports in Creatio**
- **Setting up the report**
- **Setting up the report with an image**

The "Report setup" section

Beginner

Easy

Medium

Advanced

Introduction

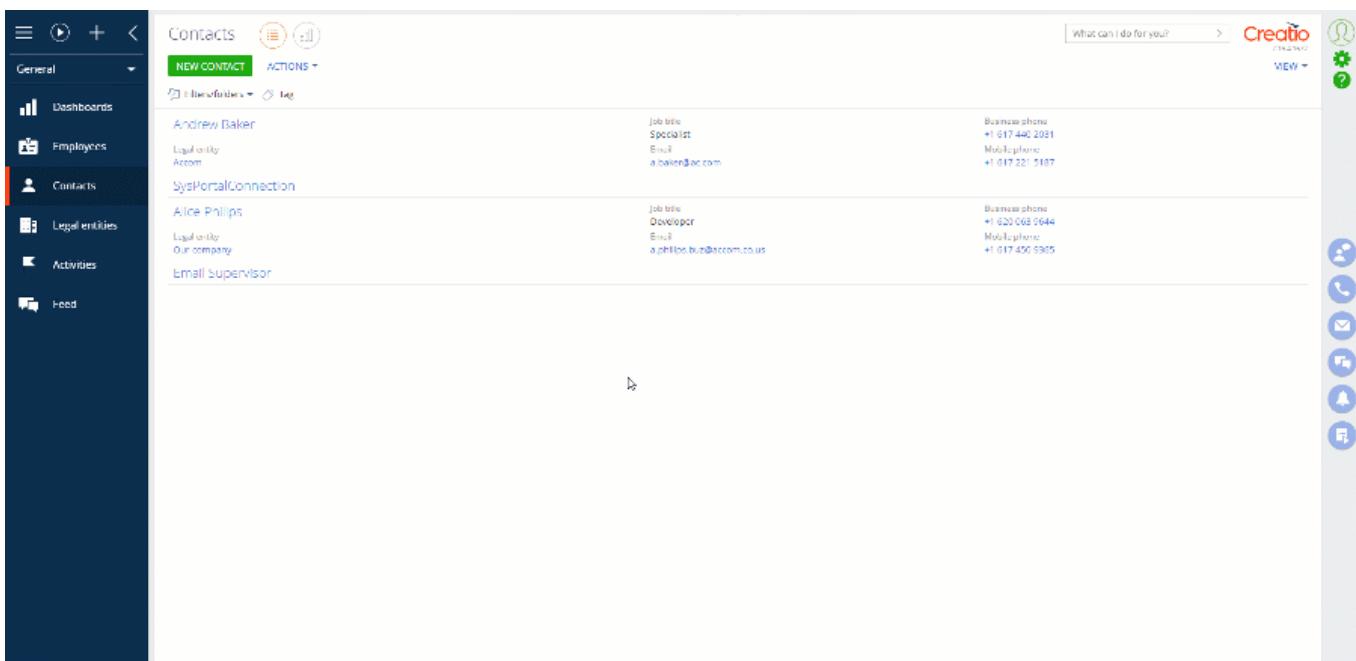
The **[Report setup]** section enables users to create reports using the Creatio tools and configure such reports using the FastReport Designer.

You can set up custom reports using Creatio version 7.15.3 and up.

To open the **[Report setup]** section:

1. Open the System Designer by clicking .
2. In the **[System setup]** block, select the **[Report setup]** link (Fig. 1).

Fig. 1. – Opening the [Report setup] section



The report setup section page (Fig. 2) will open. The page contains the following elements:

- The **[Close]** button – closes the report setup section.
- The **[New report]** button – selects the type of report to add (*FastReport* or *MS Word*).
- The **[Delete]** button – removes the selected report from the section list. The button appears when you select an existing report.
- The **[Search]** string – performs the search of a report by name.

Fig. 2. – The [Report setup] section list

The screenshot shows the 'Report setup' page in Creatio. At the top, there's a 'CLOSE' button and a 'NEW REPORT ▾' button which is currently active, showing a dropdown menu with 'FastReport' and 'MS Word' options. Below this is a table listing various reports:

Section	Type
Account Info	FastReport
Contract	MS Word
Data entry compliance (sample)	FastReport
Invoice	MS Word
Invoice for contract	MS Word
Noteworthy events for contact (sample)	FastReport
Opportunity summary	MS Word
Order	MS Word
Quotation	MS Word

Selecting the **[FastReport]** option, a report setup page (Fig. 3) will open. The page contains several functional areas with the tools to create and set up a report.

Fig. 3. – The [Report setup] page

The screenshot shows the 'New report' setup page. It has several sections:

- 1. Top Bar:** Includes 'New report' buttons for 'APPLY' and 'CANCEL'.
- 2. Data Sources:** A large area titled 'Specify data sources for the report' containing JSON code for data source definitions.
- 3. Notes:** A section titled 'Note' with instructions for setting up a report, numbered 1 through 6.
- 4. File Imports:** Sections for 'Download file with data sources to design a report in the FastReport Designer' (with a 'DOWNLOAD FILE' button) and 'Import a file with the report template' (with an 'UPLOAD FILE' button).

Toolbar

The Toolbar (Fig. 3, 1) has the following buttons:

- **[Apply]** – saves the created report. The button appears after you click any field on the page.
- **[Cancel]** – closes the report setup without saving the changes.

The setup area of display parameters

The setup area of display parameters (Fig. 3, 2) includes the following elements:

- The **[Report title]** field – sets the name for the report. Depending on your settings, the report name will display in the menu of the **[Print]** button in a section or on a record page, as well as in the **[Reports]** button drop-down list of a section in the dashboard view.
- The **[Section]** field – a drop-down list of sections that you can select for generating your report.
- The **[Show on the section list review]** checkbox – determines whether the report displays in the drop-down list of the **[Print]** button in a section.
- The **[Show on the section record page]** checkbox – determines whether the report displays in the drop-down list of the **[Print]** button on a page of a section record.
- The **[Show in the section analytics view]** checkbox – determines whether the report displays in the drop-down list of the **[Reports]** button in the section dashboard view. In this case, you can use a custom page of additional filtering. Set the filtering parameters using a schema. Specify this schema in the **[Filter page]** field. To set the filtering parameters, select a report (click the **[Reports]** button in the section dashboard view and select the report from the drop-down list).
- The **[Filter page]** field – a drop-down list of pages with filters. The field only appears if you select the **[Show in the section analytics view]** checkbox. You can select the *SimpleReportFilterPage* standard filtering page or create a custom filtering page and specify the *BaseReportFilterPage* schema as its parent schema.

Standard filters are implemented in the *SimpleReportFilterPage* filtering page schema. After you select a report from the drop-down list of the **[Reports]** button in the section dashboard view, you will see a page where you can specify the following report parameters:

- **[Selected records].**
- **[Filtered records in list].**
- **[All records in list].**

The information [Note] area

The **[Note]** area (Fig. 3, 3) contains a short instruction on how to set up the reports.

Working area

Use the working area of the report setup page (Fig. 3, 4) to set up your report. The report setup working area contains the flowing elements:

- The **[Specify data sources for the report]** block – use it to enter the sources of data for your report and set localizable strings in the *json* format.
- The **[Download file with data sources to design a report in the Fast Report Designer]** – use it to download the report template (a **.frx* file). To do this, click the **[Download file]** button.
- The **[Import a file with the report template]** block – use it to upload the template to Creatio after you set it up using FastReport. To do this, click the **[Upload file]** button.

Learn more about setting up reports in the "**Setting up reports in Creatio**" article.

See also:

- [Setting up reports in Creatio](#)
- [Setting up the report](#)
- [Setting up the report with an image](#)

Setting up reports in Creatio

Beginner

Easy

Medium

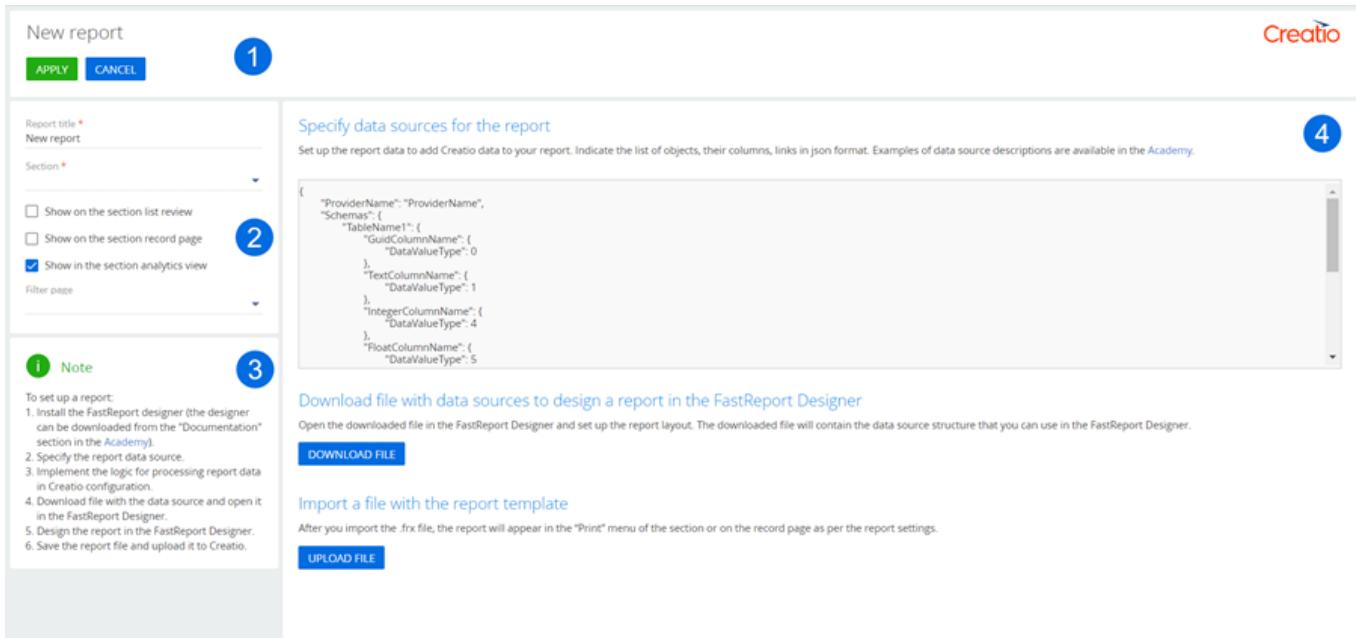
Advanced

Introduction

You can set up custom reports using Creatio version 7.15.3 and up.

The **[Report setup]** section enables users to create reports using Creatio tools and configure these reports via the FastReport designer. You can learn more about the section and find the interface description (Fig. 1) in the "**The "Report setup" section**" article.

Fig. 1. – The report setup page



General algorithm of creating a report:

1. Install the FastReport Designer (you only perform it once).
2. Create a report in the **[Report setup]** section.
3. Specify the data sources for the report.
4. Create a report data provider that will implement processing of data logic.
5. Download the file with data sources and set it up in the FastReport Designer.
6. Upload the configured report template to Creatio.

Installing the FastReport Designer

You will need the following components to work with the Report Designer:

1. Windows OS.
2. 64-bit Microsoft .Net Framework 4.7.2.

To install the FastReport Designer, use the following [link](#) and download the zip archive.

Creating a new report

To create a new report:

1. Open the System Designer by clicking . In the **[System setup]** block, click the **[Report setup]** link.
2. Click **[New report] → [FastReport]**.
3. In the parameter setup area (Fig. 1, 2) specify the report title, the section for the report, the display parameters.

Specifying the report data sources

In the **[Specify data sources for the report]** block of the working area (Fig. 1, 4), specify the list of objects, their columns and connections that will be used to receive data. Specify the localizable strings if needed. Use the JSON format. Example of providing a data source:

```
{
    // Name of the data provider class.
    "ProviderName": "YourProviderName",
    // Table structure for the report template.
    "Schemas": {
        // Name of the database table or virtual tables, whose columns need to be
        ...
    }
}
```

```
added to the report.
    "TableName1": {
        // Name of the column that needs to be added to the report.
        "ColumnName1": {
            // Column data type.
            "DataValueType": DataValueType1
        },
        "ColumnName2": {
            "DataValueType": DataValueType2
        }
    },
    "TableName2": {
        "ColumnName1": {
            "DataValueType": DataValueType1
        },
        "ColumnName2": {
            "DataValueType": DataValueType2
        }
    },
    // Report localizable strings.
    "LocalizableStrings": {
        // Name of the report localizable string.
        "LocalizedString1": {
            // Data type of the localizable string.
            "DataValueType": 1
        },
        "LocalizedString2": {
            "DataValueType": 1
        }
    }
}
}
```

The *DataValueType* parameter contains a value from the [Terrasoft.core.enums.DataValueType](#) enumeration.

Click **[Apply]** in the toolbar (Fig. 1, 1) to save the data.

Creating a report data provider

The report data provider is a custom class written in C#. To create the provider:

1. In the custom development package, create a **[Source code]** type schema.
2. Create a service class in the schema source code. Use the *Terrasoft.Configuration* namespace or any of its embedded namespaces. Mark the class with the *[DefaultBinding]* attribute containing the necessary parameters. The service class must be the inheritor of *Terrasoft.Configuration.Reporting.IFastReportDataSourceDataProvider*.
3. Add the *GetLocalizableStrings(UserConnection)* method implementation to the class. The method implements localization of the report fields.
4. Add the *ExtractFilterFromParameters(UserConnection, Guid, IReadOnlyDictionary)* method implementation to the class. This method is responsible for adding the interface filters.
5. Add the *GetData(UserConnection, IReadOnlyDictionary)* method implementation to the class. This method must return a *Task<ReportDataDictionary>* type value. Describe the logic of receiving the report data in the method.
6. Publish the source code schema.

Example of implementing the report data processing logic:

```
namespace Terrasoft.Configuration
{
    using System.Collections.Generic;
    using System.Threading.Tasks;
    using Terrasoft.Configuration.Reporting.FastReport;
```

```
using Terrasoft.Core;
using Terrasoft.Core.Factories;

// Name of the data provider class for the report, whose logic needs to be
implemented.
[DefaultBinding(typeof(IFastReportDataSourceDataProvider), Name =
"YourProviderName")]
public class YourProviderName : IFastReportDataSourceDataProvider
{
    // The code for implementing the logic of getting data for the report.

    // Localization of the report strings.
    private IEnumerable<IReadOnlyDictionary<string, object>>
GetLocalizableStrings(UserConnection userConnection) {
        var localizableStrings = _localizableStringNames.ToDictionary(
            x => x,
            x => (object)(new LocalizableString(userConnection.ResourceStorage,
_resourceManagerName, $"LocalizableStrings.{x}.Value")).Value);
        return new[] { localizableStrings };
    }

    // Adding the interface filters.
    private IEntitySchemaQueryFilterItem
ExtractFilterFromParameters(UserConnection userConnection, Guid entitySchemaUID,
    IReadOnlyDictionary<string, object> parameters) {
        var managerItem =
userConnection.EntitySchemaManager.GetItemById(entitySchemaUID);
        return parameters.ExtractEsqFilterFromReportParameters(userConnection,
managerItem.Name) ?? throw new Exception();
    }

    // Adding data to the report.
    public Task<ReportDataDictionary> GetData(UserConnection userConnection,
IReadOnlyDictionary<string, object> parameters) {
    }
}
}
```

Setting up templates in the FastReport Designer

Download the file with data sources. Click the **[Download file]** button in the **[Download file with data sources to design a report in the FastReport Designer]** block (Fig. 1, 4) of the working area. The file must have the **.frx* extension.

Double click the downloaded file to open it in FastReport and configure the template layout. Learn more about configuring the template in the [FastReport documentation](#).

The file saves the structure of the data source implemented in the **[Report setup]** section.

The FastReport Designer is a third party application. The template preview function is not available.

Uploading the configured template to Creatio

Click the **[Upload template]** button in the **[Import a file with the report template]** block (Fig. 1, 4) of the working area to upload the prepared template to Creatio. After you upload the template, you can generate a report in the section dashboard view or on a record page. You can specify this in the parameter setup area (Fig. 1, 2). The generated report will be saved in the *pdf* format.

The **[Print]** and **[Reports]** buttons display in corresponding sections and on record pages if there is at least one report configured and published for a specific section.

Multilingual interface elements in reports

The [Translations] section in the System Designer enables setting the values of interface elements for a multilingual report. To find the previously localized strings of the report, use the *Configuration:SchemaName* key (e.g., *Configuration:UsrContactDataSourceCode*). You can find a report field using the following key: *Configuration:SchemaName:FieldName* (e.g., *Configuration:UsrContactDataSourceCode:LocalizableStrings.ReportTitle.Value*).

If the [Show on the section list review] and [Show on the section record page] checkboxes are selected, translate the report title. You can find the report title using the following key: *Configuration:SchemaName:Caption* (e.g., *Configuration:UsrContactDataSourceCode:Caption*). If the [Show in the section analytics view] checkbox (Fig. 1, 2) is selected, translate the report title. You can find the report title using the following key: *Data:SysModuleAnalyticsReport.Caption:FieldIdentifier* (e.g., *Data:SysModuleAnalyticsReport.Caption:d52e8b78-772b-77ee-3394-bdb3616d859a*).

Learn more about working with the [Translations] section in the "[How to translate the interface and system elements in Creatio](#)" article.

Transferring the package to another development environment

To transfer the package with the report to another environment, go to the [Configuration] section -> the [Data] tab and bind the data of the following elements:

- *FastReportTemplate_ReportName* – the report template. To bind it, use the template Id from the [dbo.FastReportTemplate] database table.
- *FastReportDataSource_ReportName* – the source of the report data. To bind it, use the source Id from the [dbo.FastReportDataSource] database table.
- *SysModuleReport_ReportName* – the report. To bind it, use the report Id from the [dbo.SysModuleReport] database table.

You can view the record Id in the database table even if you do not have access to the database. To do this, display the Id system column in the window of binding data to packages.

See also:

- [Setting up reports in Creatio](#)
- [Setting up the report](#)
- [Setting up the report with an image](#)

Setting up the report

Beginner Easy **Medium** Advanced

Case description

You can set up custom reports using Creatio version 7.15.3 and up.

Create a "Contact Data" base report that would display the following information about the contacts:

- [Full name];
- [Birthday];
- [Gender];
- [Account].

Source code

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

Case implementation algorithm

1. Set up the report display parameters

Specify the following values (Fig. 2) for the created report in the parameter setup area (Fig. 1, 2):

- [Report title] – "Contact Data";
- [Section] – "Contacts";
- [Show in section] – select the checkbox;

- [Show in section] – select the checkbox;
- [Show in the section analytics view] – select the checkbox;
- [Filter page] – SimpleReportFilterPage.

SimpleReportFilterPage – a client schema that implements standard simple filters.

Fig. 1. – The report setup page

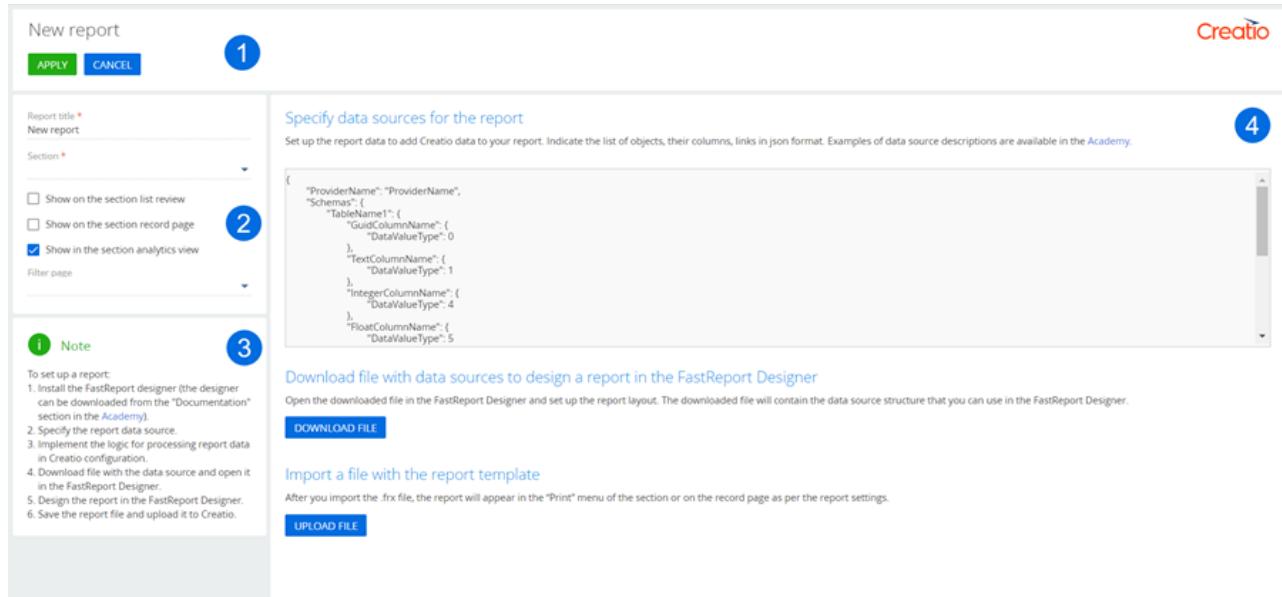
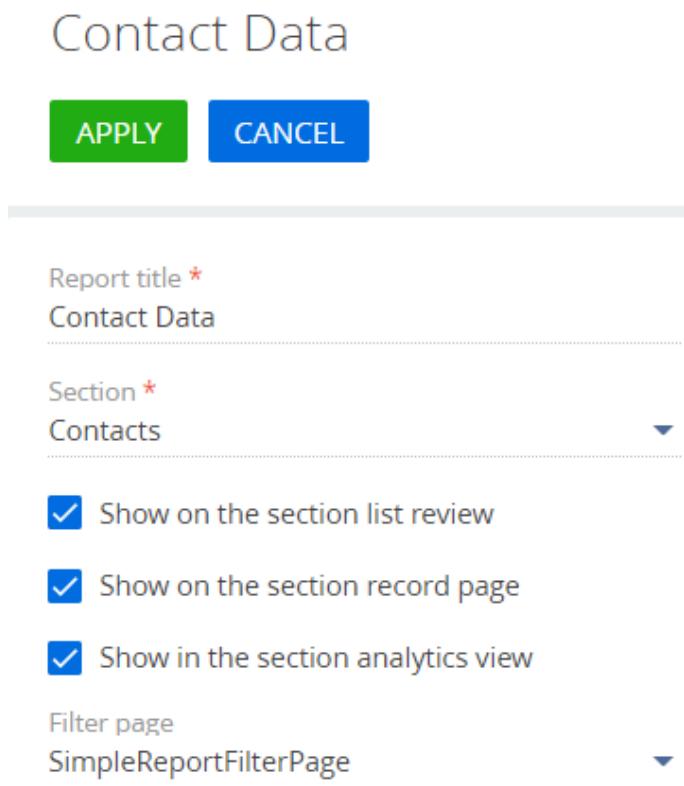


Fig. 2. – Setting up the report display parameters



2. Specify the data sources

In the [Specify data sources for the report] block of the working area (Fig. 1, 4), add the code below:

```
{
```

```
// Name of the data provider class.
"ProviderName": "ContactDataProvider",
"Schemas": {
    "ContactData": {
        "Full name": {"ValueType": 1},
        "Birthday": {"ValueType": 1},
        "Gender": {"ValueType": 1},
        "Account": {"ValueType": 1}
    },
    // Added for localization.
    "LocalizableStrings": {
        "ReportTitle": {"ValueType": 1},
        "FullNameLabel": {"ValueType": 1},
        "BirthdayLabel": {"ValueType": 1},
        "GenderLabel": {"ValueType": 1},
        "AccountLabel": {"ValueType": 1}
    }
}
}
```

Click **[Apply]** to save and apply the changes (Fig. 1, 1).

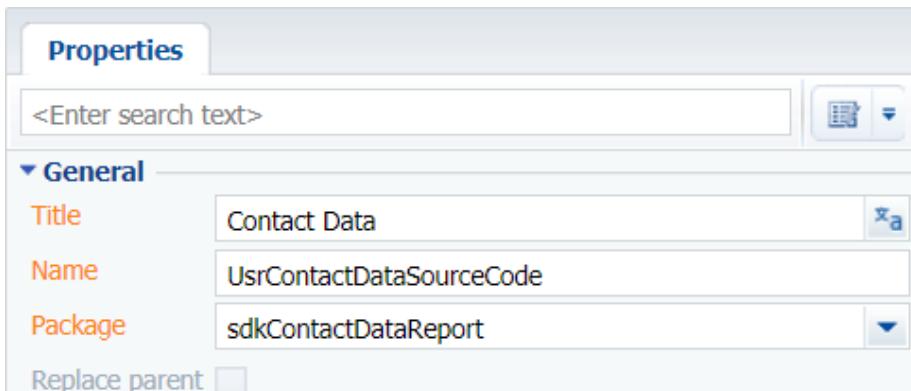
3. Create the report data provider

Go to the **[Advanced settings]** section -> **[Configuration]** -> **[Custom package]** -> the **[Schemas]** tab. Click **[Add]** -> **[Source code]**. Learn more about creating a schema of the **[Source Code]** type in the “**Creating the [Source code] schema**” article.

Specify the following parameters for the created object schema (Fig. 3):

- **[Title]** – "Contact Data";
- **[Name]** – "UsrContactDataSourceCode".

Fig. 3. – Setting up the **[Source Code]** type object schema



The complete source code of the module is available below:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Linq;
    using System.Threading.Tasks;
    using Terrasoft.Common;
    using Terrasoft.Configuration.Reporting.FastReport;
    using Terrasoft.Core;
    using Terrasoft.Core.Entities;
    using Terrasoft.Core.Factories;
    using Terrasoft.Nui.ServiceModel.Extensions;
    using EntitySchema = Terrasoft.Core.Entities.EntitySchema;
    using EntitySchemaColumn = Terrasoft.Core.Entities.EntitySchemaColumn;
```

```
// Name of the data provider class for the report, whose logic needs to be
implemented.
[DefaultBinding(typeof(IFastReportDataSourceDataProvider), Name =
"ContactDataProvider")]
public class ContactDataProvider : IFastReportDataSourceDataProvider
{

    private Guid _entitySchemaUID = new Guid("16BE3651-8FE2-4159-8DD0-A803D4683DD3");
    // Name of the source code schema.
    private readonly string _resourceManagerName = "UsrContactDataSourceCode";
    private readonly string[] _localizableStringNames = new[] {
        "ReportTitle",
        "FullNameLabel",
        "BirthdayLabel",
        "GenderLabel",
        "AccountLabel"
    };

    // Populating the report columns.
    private IEnumerable<IReadOnlyDictionary<string, object>> GetContactData(
        UserConnection userConnection,
        Guid entitySchemaUID,
        IEntitySchemaQueryFilterItem filter) {
        // Getting the object schema.
        var entitySchema =
userConnection.EntitySchemaManager.GetInstanceByUID(entitySchemaUID);
        // Creating the object of the EntitySchemaQuery class.
        EntitySchemaQuery query = new EntitySchemaQuery(entitySchema);
        // Adding columns to query.
        query.AddColumn("Name");
        query.AddColumn("BirthDate");
        var gender = query.AddColumn("Gender.Name");
        var account = query.AddColumn("Account.Name");
        // Adding the created filter.
        query.Filters.Add(filter);
        // Getting the collection of contacts.
        var contacts = query.GetEntityCollection(userConnection);
        var contactsCollection = new Collection<Dictionary<string, object>>();
        // Populating the report columns.
        foreach (var entity in contacts)
        {
            contactsCollection.Add(new Dictionary<string, object> {
                ["Full name"] = entity.GetTypedColumnValue<string>("Name"),
                ["Birthday"] = entity.GetTypedColumnValue<string>("BirthDate"),
                ["Gender"] = entity.GetTypedColumnValue<string>(gender.Name),
                ["Account"] = entity.GetTypedColumnValue<string>(account.Name)
            });
        }
        return contactsCollection;
    }

    // Localization of the report title.
    private IEnumerable<IReadOnlyDictionary<string, object>>
GetLocalizableStrings(UserConnection userConnection) {
        var localizableStrings = _localizableStringNames.ToDictionary(
            x => x,
            x => (object)(new LocalizableString(userConnection.ResourceStorage,
_resourceManagerName, $"LocalizableStrings.{x}.Value")).Value);
        return new[] { localizableStrings };
    }

    // Adding the interface filters.
    private IEntitySchemaQueryFilterItem ExtractFilterFromParameters(UserConnection
userConnection, Guid entitySchemaUID,
        IReadOnlyDictionary<string, object> parameters) {
```

```
        var managerItem =
userConnection.EntitySchemaManager.GetItemById(entitySchemaId);
        return parameters.ExtractEsqFilterFromReportParameters(userConnection,
managerItem.Name) ?? throw new Exception();
    }

    // Getting data.
    public Task<ReportDataDictionary> GetData(UserConnection userConnection,
IReadOnlyDictionary<string, object> parameters) {
        var filter = ExtractFilterFromParameters(userConnection, _entitySchemaId,
parameters);
        var result = new ReportDataDictionary {
            // Populating the report columns.
            ["ContactData"] = GetContactData(userConnection, _entitySchemaId,
filter),
            ["LocalizableStrings"] = GetLocalizableStrings(userConnection)
        };
        return Task.FromResult(result);
    }
}
```

Populate the localizable strings of the report with the following values (table 1):

Table 1. Setting up the localizable strings

Name	English (United States)
ReportTitle	Contact Data
FullNameLabel	Full name
BirthdayLabel	Birthday
GenderLabel	Gender
AccountLabel	Account

Learn more about working with localizable strings in the "[Source code designer](#)" article.

Publish the schema.

4. Download the template and set up its layout in FastReport

Download *ContactData.frx*. Click **[Download file]** in the **[Download file with data sources to design a report in the FastReport Designer]** block (Fig. 1, 4) of the working area.

To open the template in the Report Designer:

1. Run the *Terrasoft.Reporting.FastReport.Designer.exe* file from the [zip archive](#) (Fig. 4) and open the FastReport designer.

Fig. 4. – The FastReport Designer file

Name	Type	Compressed ...
 FastReport.Bars.dll	Application extension	1,731 KB
 FastReport.dll	Application extension	3,101 KB
 FastReport.Editor.dll	Application extension	338 KB
 Terrasoft.Reporting.FastReport.Designer.exe	Application	3 KB
 Terrasoft.Reporting.FastReport.Designer.exe.config	XML Configuration File	1 KB

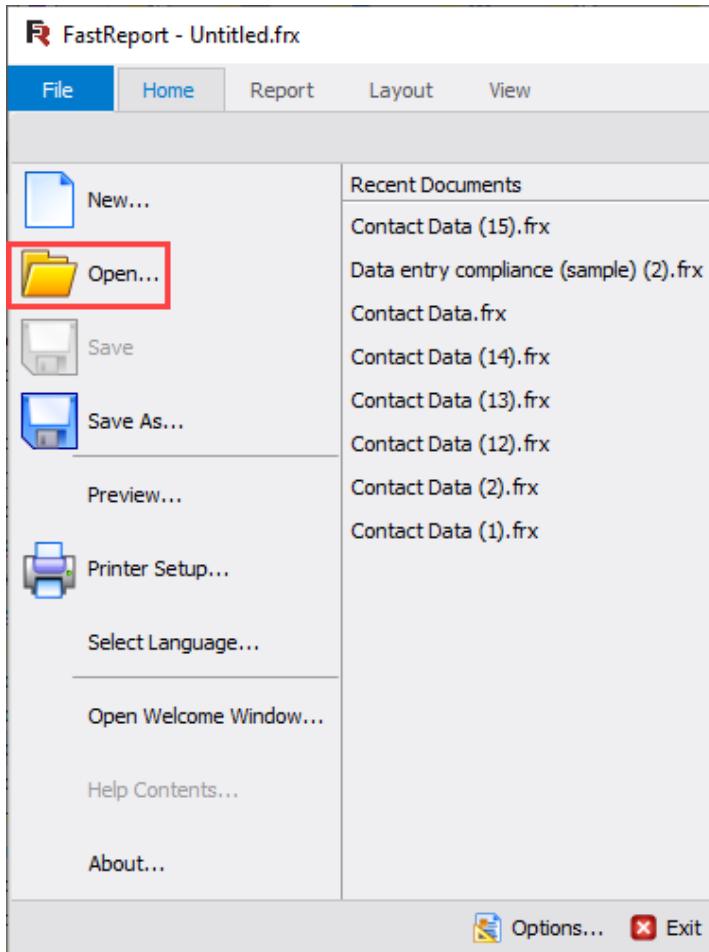
2. Click **[Open...]** in the window that pops up (Fig. 5).

Fig. 5. – The [Open...] button



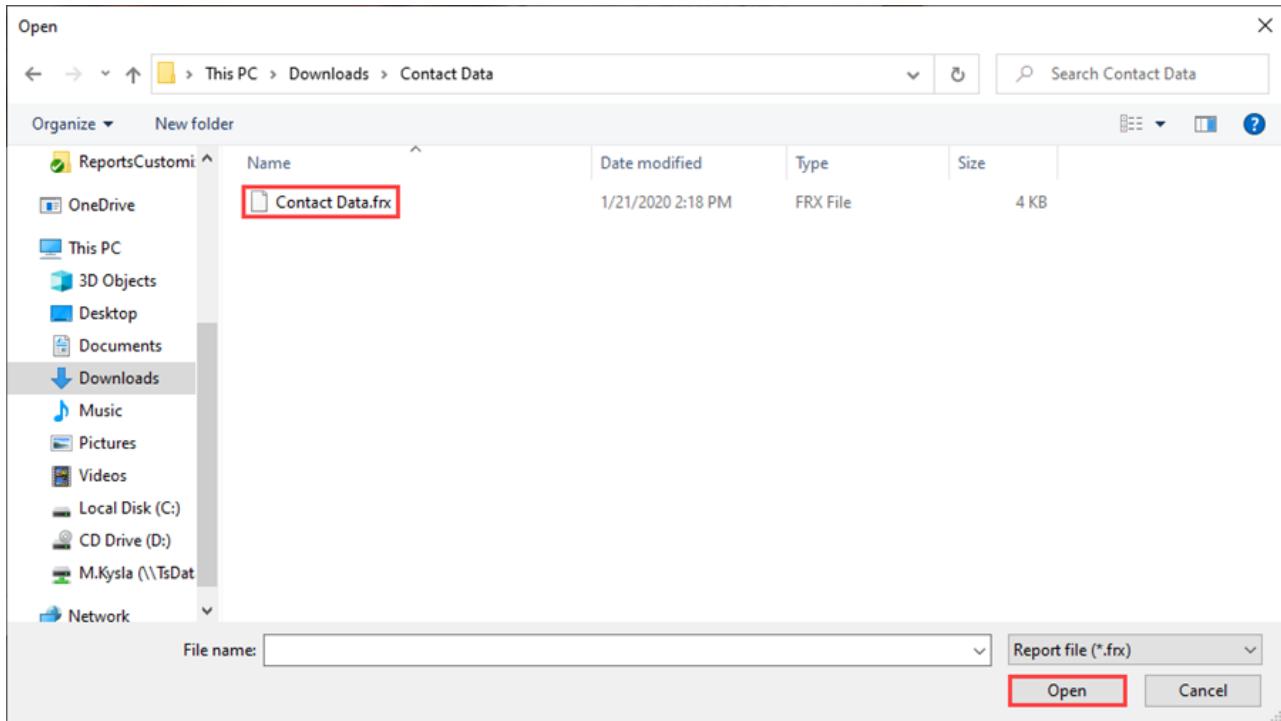
You can also open the report template from the **[File]** menu -> **[Open...]** or by pressing **[Ctrl+O]** key combination.

Fig. 6. – The **[File]** menu with the **[Open...]** option



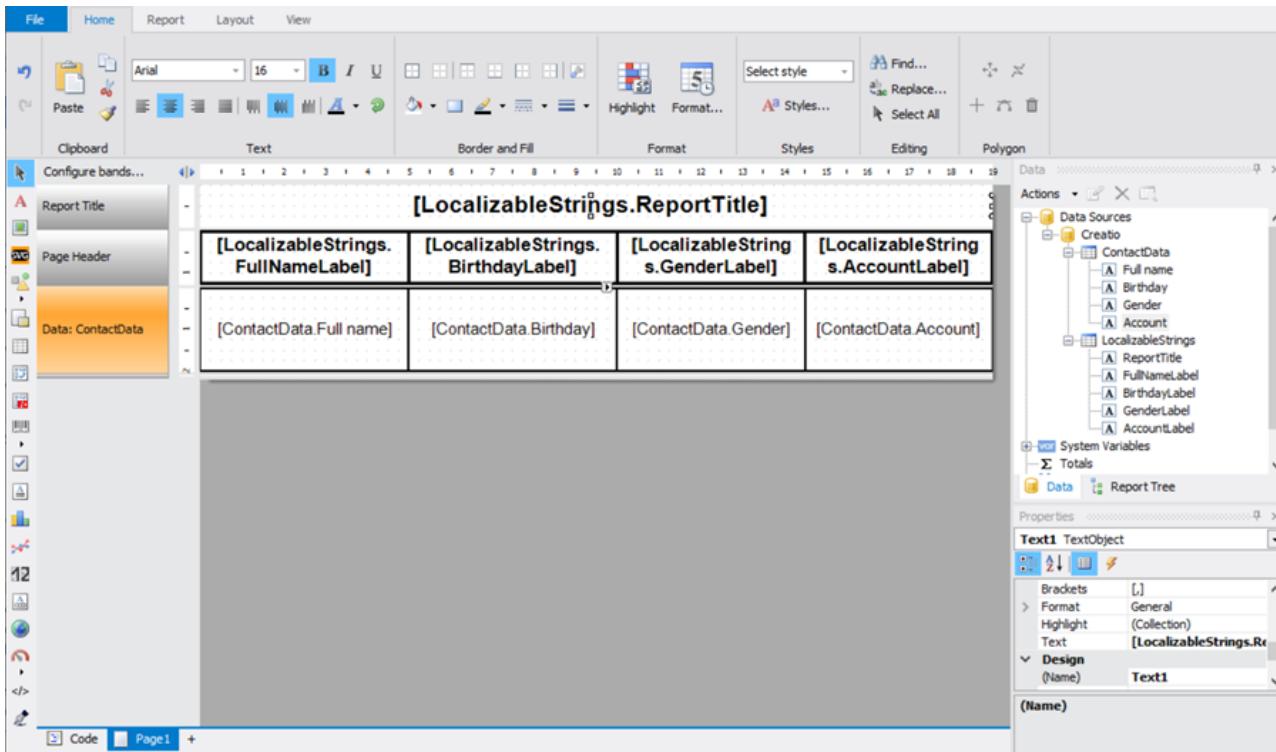
3. Navigate to the folder containing the downloaded report (usually, it is the “Downloads” folder), select the file with the template and click [Open] (Fig. 7).

Fig. 7. – Opening the report template in FastReport



Set up the template layout (Fig. 8).

Fig. 8. – Setting up template layout in the FastReport Designer



5. Upload the configured report template to Creatio

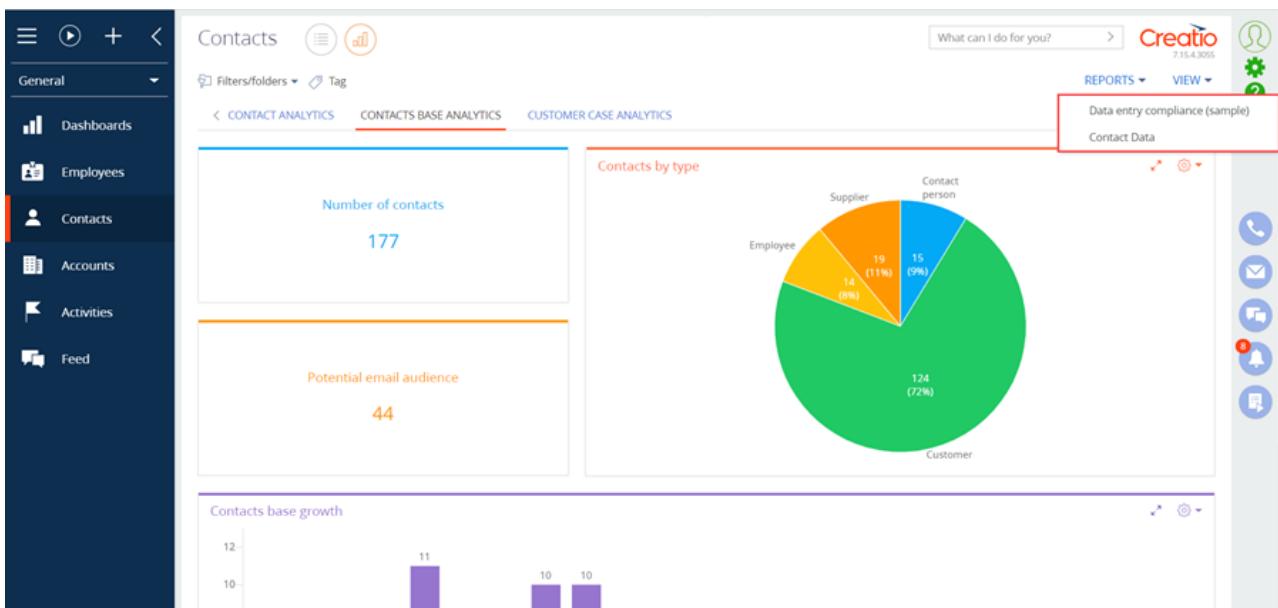
To upload the *ContactData.frx* file, click **[Upload template]** in the **[Import a file with the report template]** block of the section working area (Fig. 1, 4). Confirm that the template has been uploaded successfully.

As a result, the "Contact Data" report will be available on the contact page under the **[Print]** (Fig. 9).

Fig. 9. – The "Contact Data" report displayed on the record page of the [Contacts] section

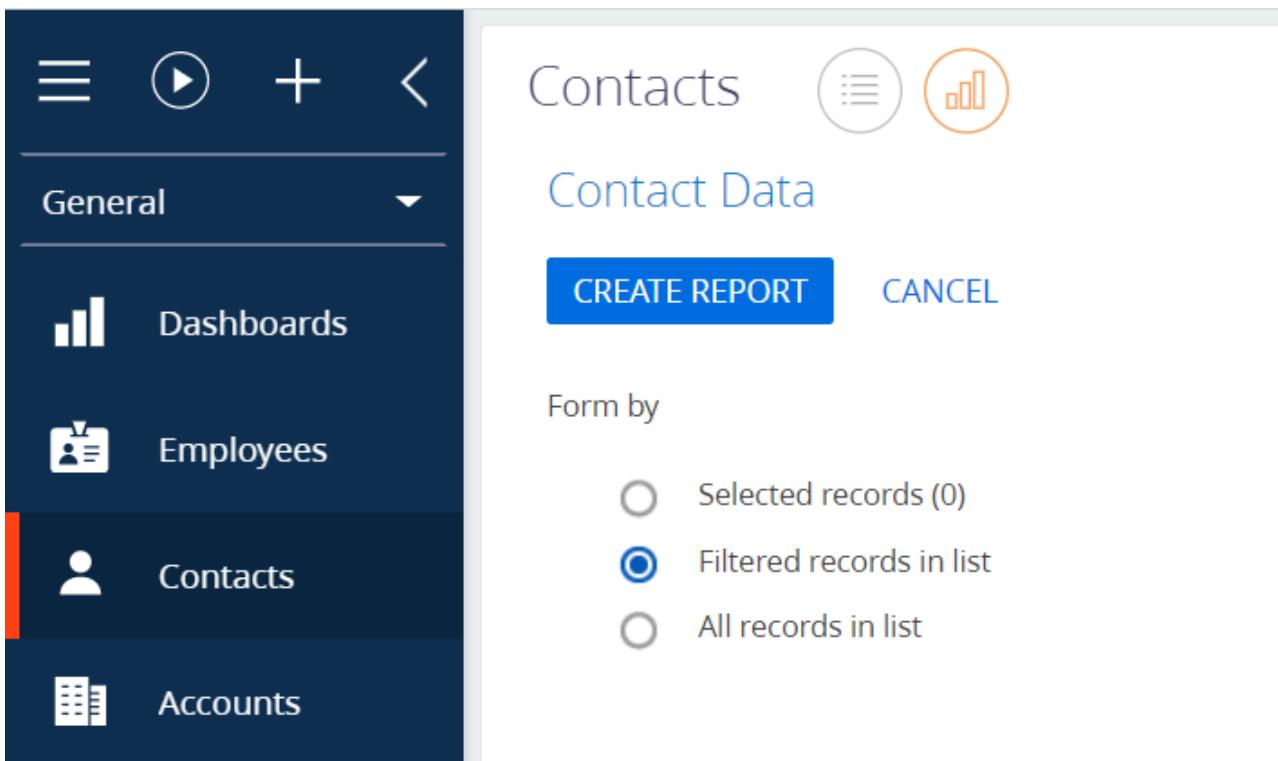
You can also find the report on the **[Contacts]** sectin dashboard view, under the **[Reports]** button (Fig. 10).

Fig. 10. – The "Contact Data" report displayed in the [Contacts] section dashboard view



The report is generated as per the selected [Filtered records in list] option in the filtering page (Fig. 11). To upload the report, click [Create report]. To close the filtering page and cancel generating the report, click [Cancel].

Fig. 11. – The filtering page of the "Contact Data" report



The report looks as follows (Fig. 12):

Fig. 12. – Example of the "Contact Data" report

Contact Data

Full name	Birthday	Gender	Account
Andrew Baker	1/20/1986 12:00:00 AM	Male	Accom

Report by several records in a section

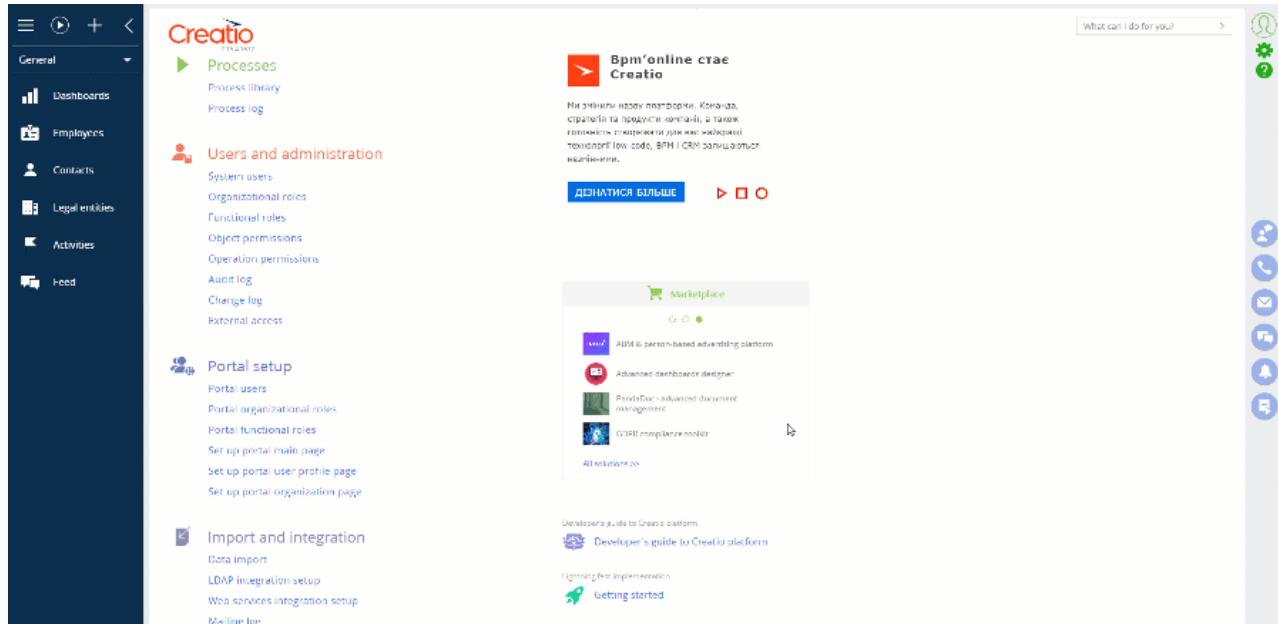
You can generate a report based on data from several records. To this, set up the following elements:

- [Show in section] – select the checkbox (Fig. 2).
- [Filtered records in list] – select the checkbox (Fig. 11).
- [All records in list] – select the checkbox (Fig. 11).

To include information from several section records to your report (Fig. 13):

1. Open the needed section.
2. Apply filters if needed.
3. Click [Actions] → [Select multiple records].
4. Select the needed report in the [Print] button drop-down list.

Fig. 13. – Getting the “Contact Data” report from several records in the [Contacts] section



The report looks as follows (Fig. 14):

Fig. 14. – Example of the "Contact Data" report based on several section records

Contact Data

Full name	Birthday	Gender	Account
Alice Philips	1/8/2020 12:00:00 AM	Male	Our company
Andrew Baker	1/20/1986 12:00:00 AM	Male	Accom

See also:

- **The "Report setup" section**
- **Setting up reports in Creatio**
- **Setting up the report with an image**

Setting up the report with an image

[Основы](#)[Легкий](#)[Средний](#)[Сложный](#)

Case description

You can set up custom reports using Creatio version 7.15.3 and up.

Create an "Account Info" report that would display the following information about the accounts:

- [Name].
- [Logo].

Source code

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

Case implementation algorithm

1. Set up the report display parameters

Set the following values (Fig. 2) in the parameter setup area:

- [Report title] – "Account Info".
- [Section] – "Accounts".
- [Show in section].
- [Show in card].
- [Show in the section analytics view].
- [Filter page] – SimpleReportFilterPage.

SimpleReportFilterPage – a client schema that implements standard simple filters.

Fig. 1. – The report setup page

The screenshot shows the 'New report' setup page in Creatio. The interface is divided into two main sections: 'Specify data sources for the report' on the right and 'Report title' on the left.

Left Section (Report title):

- Buttons: APPLY, CANCEL.
- Input field: Report title * (with placeholder 'New report').
- Section dropdown: Section *
- Checkboxes:
 - Show on the section list review
 - Show on the section record page
 - Checked:** Show in the section analytics view (marked with a blue circle labeled 2)
- Filter page dropdown.
- Note icon (marked with a blue circle labeled 3):
 - To set up a report:
 - Install the FastReport designer (the designer can be downloaded from the "Documentation" section in the Academy).
 - Specify the report data source.
 - Implement the logic for processing report data in Creatio configuration.
 - Download file with the data source and open it in the FastReport Designer.
 - Design the report in the FastReport Designer.
 - Save the report file and upload it to Creatio.

Right Section (Specify data sources for the report):

- Header: Specify data sources for the report.
- Text: Set up the report data to add Creatio data to your report. Indicate the list of objects, their columns, links in json format. Examples of data source descriptions are available in the Academy.
- JSON code:

```
{  "ProviderName": "ProviderName",  "Schemas": {    "TableName1": {      "GuidColumnName": {        "DataValueType": 0      },      "TextColumnName": {        "DataValueType": 1      },      "IntegerColumnName": {        "DataValueType": 4      },      "FloatColumnName": {        "DataValueType": 5      }    }  }}
```
- Buttons: DOWNLOAD FILE (marked with a blue circle labeled 4), UPLOAD FILE.
- Text: Import a file with the report template. After you import the .frx file, the report will appear in the "Print" menu of the section or on the record page as per the report settings.

Fig. 2. – Setting up the report display parameters

Account Info

APPLY**CANCEL**

Report name *

Account Info

Section *

Accounts

 Show in the section list view Show in the section record page Show in the section analytics view

Filter page

SimpleReportFilterPage

2. Specify the data sources

In the **[Specify data sources for the report]** block of the working area (Fig. 1, 4), add the code below:

```
{  
    // The name of the data provider class.  
    "ProviderName": "AccountInfoProvider",  
    "Schemas": {  
        "Data": {  
            "Name": { "DataProviderType": 1 },  
            "Logo": { "DataProviderType": 14 }  
        },  
        // Added for localization.  
        "LocalizableStrings": {  
            "ReportTitle": { "DataProviderType": 1 },  
            "NameLabel": { "DataProviderType": 1 },  
            "LogoLabel": { "DataProviderType": 1 }  
        }  
    }  
}
```

Click **[Apply]** to save and apply the changes (Fig. 1, 1).

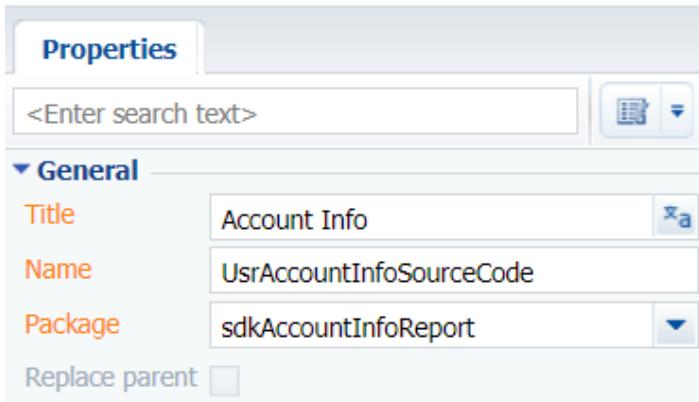
3. Create the report data provider

Go to the **[Advanced settings]** section -> **[Configuration]** -> Custom package -> the **[Schemas]** tab. Click **[Add]** -> **[Source code]**. Learn more about creating a schema of the **[Source Code]** type in the “[Creating the \[Source code\] schema](#)” article.

Specify the following parameters for the created object schema (Fig. 3):

- **[Title]** – "Account Info".
- **[Name]** – "UsrAccountInfoSourceCode".

Fig. 3. – Setting up the [Source Code] type object schema



The complete source code of the module is available below:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Threading.Tasks;
    using Terrasoft.Common;
    using Terrasoft.Configuration.Reporting.FastReport;
    using Terrasoft.Core;
    using Terrasoft.Core.Entities;
    using Terrasoft.Core.Factories;

    // The name of the data provider class for the report, whose logic needs to be
    implemented.
    [DefaultBinding(typeof(IFastReportDataSourceDataProvider), Name =
    "AccountInfoProvider")]
    public class AccountInfoProvider : IFastReportDataSourceDataProvider
    {
        // The name of the schema with the source code.
        private readonly string _resourceManagerName = "UsrAccountInfoSourceCode";
        private readonly string[] _localizableStringNames = new[] {
            "ReportTitle",
            "NameLabel",
            "LogoLabel"
        };

        // Populating columns in the report.
        private IEnumerable<IReadOnlyDictionary<string, object>> GetData(UserConnection
userConnection, IEntitySchemaQueryFilterItem filter) {
            var esq = new EntitySchemaQuery(userConnection.EntitySchemaManager,
    "Account");
            // Adding columns to the request.
            var nameColumn = esq.AddColumn("Name").OrderByDesc();
            var logoColumn = esq.AddColumn("AccountLogo.Data");
            // Adding the created filter.
            esq.Filters.Add(filter);
            return esq.GetEntityCollection(userConnection)
                .Select(x => new Dictionary<string, object> {
                    ["Name"] = x.GetTypedColumnValue<string>(nameColumn.Name),
                    ["Logo"] = x.GetStreamValue(logoColumn.Name)?.ToArray()
                });
        }

        // Localization of the report title.
        private IEnumerable<IReadOnlyDictionary<string, object>>
GetLocalizableStrings(UserConnection userConnection) {
            var localizableStrings = _localizableStringNames.ToDictionary(

```

```
        x => x,
        x => (object)(new LocalizableString(userConnection.ResourceStorage,
_resourceManagerName, $"LocalizableStrings.{x}.Value")).Value);
    return new[] { localizableStrings };
}

// Adding the interface filters.
private IEntitySchemaQueryFilterItem ExtractFilterFromParameters(UserConnection
userConnection, IReadOnlyDictionary<string, object> parameters) {
    return parameters.ExtractEsqFilterFromReportParameters(userConnection,
"Account") ?? throw new Exception();
}

// Receive data.
public Task<ReportDataDictionary> GetData(UserConnection userConnection,
IReadOnlyDictionary<string, object> parameters) {
    var filter = ExtractFilterFromParameters(userConnection, parameters);
    var result = new ReportDataDictionary {
        // Populate columns in the report.
        ["Data"] = GetData(userConnection, filter),
        ["LocalizableStrings"] = GetLocalizableStrings(userConnection)
    };
    return Task.FromResult(result);
}
}
```

Populate the localizable strings of the report with the following values (table 1):

Table 1. Setting up the localizable strings

Name	English (United States)
ReportTitle	Account Info
NameLabel	Name
LogoLabel	Logo

Learn more about working with localizable strings in the ["Source code designer"](#) article.

Publish the schema.

4. Download the template and set up its layout in FastReport

Download the *AccountInfo.frx* file. Click [Download file] in the [Download file with data sources to design a report in the FastReport Designer] block (Fig. 1, 4) of the working area.

To open the template in the Report Designer:

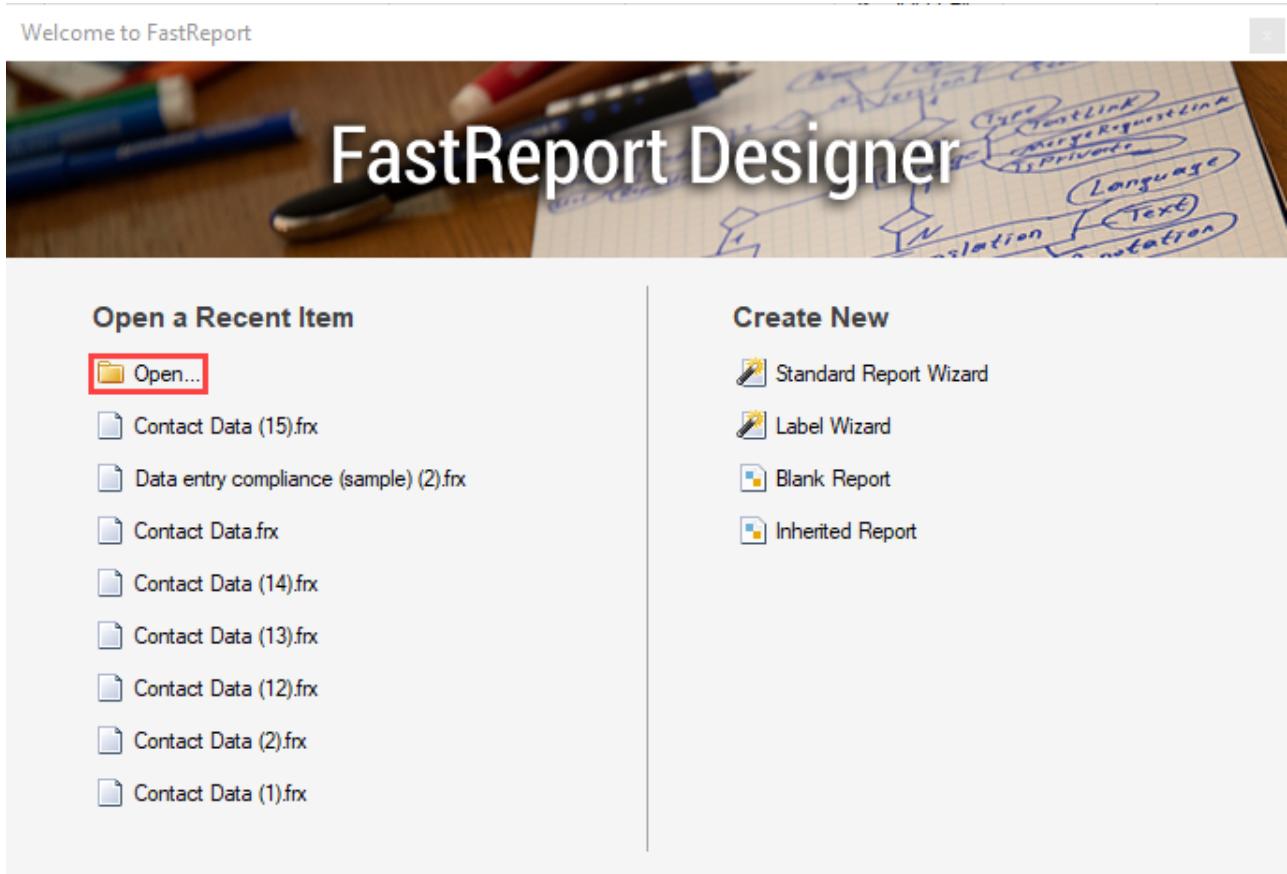
1. Run the *Terrasoft.Reporting.FastReport.Designer.exe* file from the [zip archive](#) (Fig. 4) and open the FastReport designer.

Fig. 4. – The FastReport Designer file

Name	Type	Compressed ...
FastReport.Bars.dll	Application extension	1,731 KB
FastReport.dll	Application extension	3,101 KB
FastReport.Editor.dll	Application extension	338 KB
Terrasoft.Reporting.FastReport.Designer.exe	Application	3 KB
Terrasoft.Reporting.FastReport.Designer.exe.config	XML Configuration File	1 KB

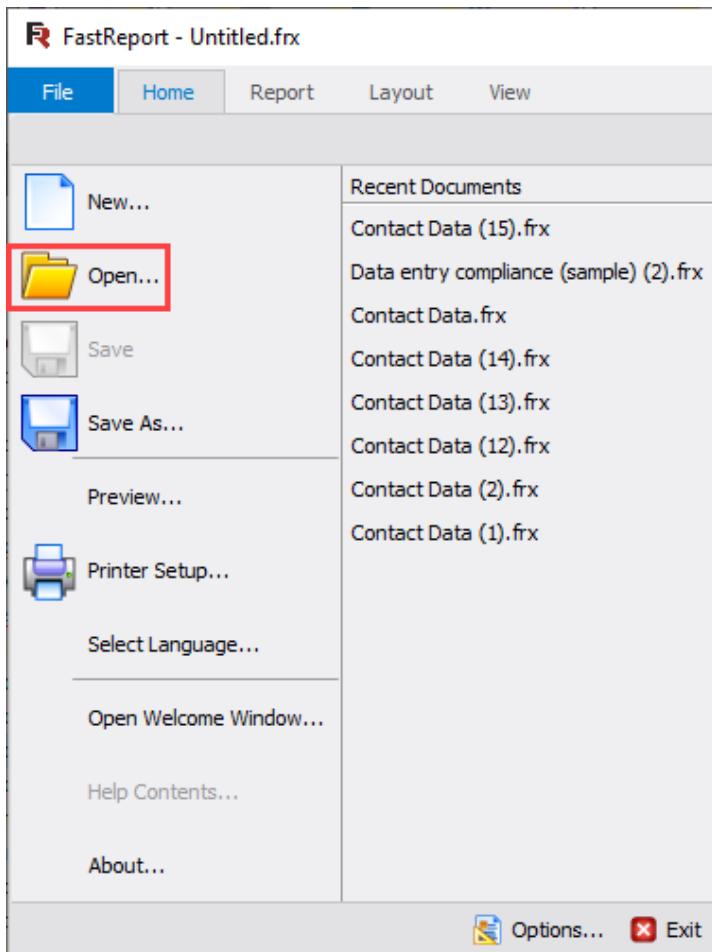
2. Click [Open...] in the window that pops up (Fig. 5).

Fig. 5. – The [Open...] button



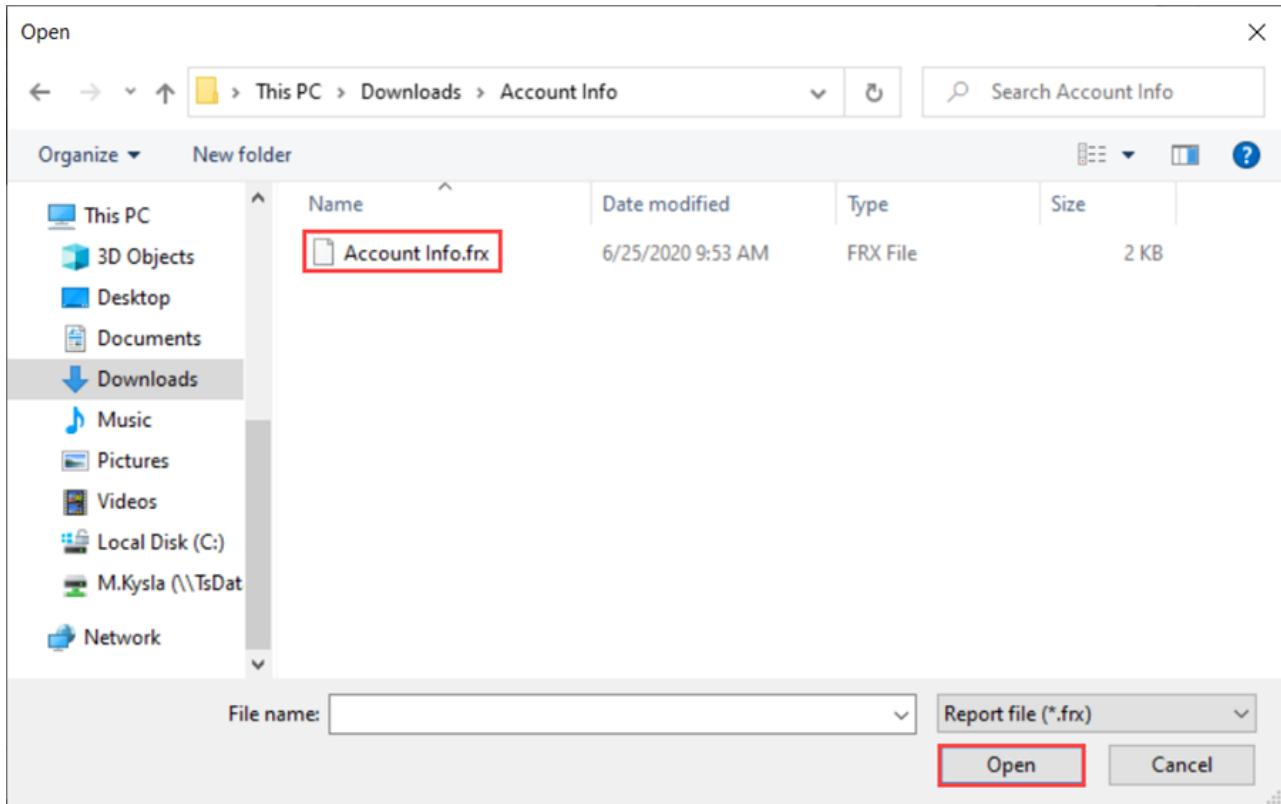
You can also open the report template from the **[File]** menu -> **[Open...]** (Fig. 6) or by pressing **[Ctrl+O]** key combination.

Fig. 6. – The **[File]** menu with the **[Open...]** option



3. Navigate to the folder containing the downloaded report (usually, it is the “Downloads” folder), select the file with the template and click **[Open]** (Fig. 7).

Fig. 7. – Opening the report template in FastReport



Set up the template layout (Fig. 8).

Fig. 8. – Setting up template layout in the FastReport Designer

The screenshot shows the FastReport Designer application. On the left, there's a toolbar with various icons for file operations like Paste, Clipboard, and Report Title/Page Header/Data/Page Footer. The main workspace displays a report layout with a table containing two columns: 'LocalizableStrings.NV' and 'LocalizableStrings.LC'. The 'Data' column contains the expression '[Data.Name]'. To the right of the workspace is a 'Data' pane showing a tree structure of data sources and system variables, and a 'Properties' pane where the 'Logo' properties are being edited. The 'BindableControl' field in the properties pane is highlighted with a red box.

To display an image in the [Logo] column of the report, select *Picture* in the [BindableControl] field (Fig. 8) of the report designer when setting up the template.

5. Upload the configured report template to Creatio

To upload the *AccountInfo.frx* file, click **[Upload template]** in the **[Import a file with the report template]** block of

the section working area (Fig. 1, 4). Confirm that the template has been uploaded successfully.

As a result, the "Account Info" report will be available on the contact page under **[Print]** (Fig. 9).

Fig. 9. – Displaying the "Account Info" report on a record page of the [Accounts] section

The screenshot shows the 'Partner Consulting' account record. The left sidebar menu is visible with 'Sales' selected. The main area displays account details: Name (Partner Consulting), Type (Customer), Owner (Marina Kysla), Web (www.partnerconsulting.co.uk), Primary phone (+44 (20) 3496 3596), Category (B), and Industry. On the right, there's a 'NEXT STEPS (0)' section with a message: 'You don't have any tasks yet. Press F above to add a task'. Below it are tabs for 'ACCOUNT INFO' (highlighted with a red box), 'CONTACTS AND STRUCTURE', and 'CONNECTED TO'. A 'PRINT' button in the top right corner is also highlighted with a red box. The top right corner also shows the Creatio logo and version (7.16.1.2135).

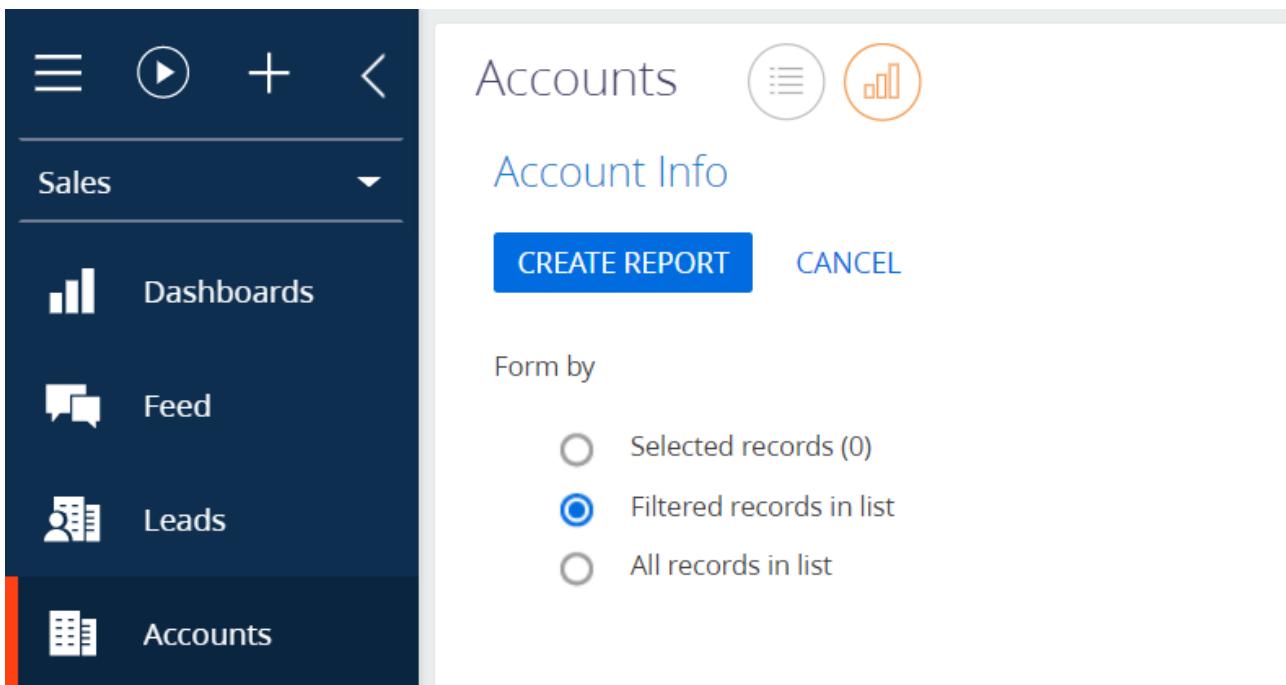
You can also find the report in the **[Accounts]** section dashboard view under **[Reports]** (Fig. 10).

Fig. 10. – Displaying the "Account Info" report in the dashboards of the [Accounts] section

The screenshot shows the 'Accounts' dashboard. The left sidebar menu is visible with 'Sales' selected. The main area features a 'FILTERS/folders' dropdown and a 'REPORTS' button (highlighted with a red box) in the top right corner. Below it is a chart titled 'Customer base growth' showing the number of new customers over time: 10 in August 2019, 1 in September 2019, and 42 in May 2020. To the right of the chart is a table titled 'Customers with no recent activity' listing three entries: Vertigo Sys... (Owner: Mary King), Infocom (Owner: Marina Kysla), and Sunrise Inv... (Owner: Marina Kysla). A summary box on the right indicates 'Number of customers: 53'.

The report is generated as per the selected **[Filtered records in list]** option on the filtering page (Fig. 11). To upload the report, click **[Create report]**. To close the filtering page and cancel generating the report, click **[Cancel]**.

Fig. 11. – The filtering page of the "Account Info" report



The report looks as follows (Fig. 12):

Fig. 12. – Example of the "Account Info" report

Account Info

Name	Logo
Infocom	

Report by several records in a section

To receive the report with several records:

1. Enable the **[Show in section]** (Fig. 2) checkbox and select several records in the section list.
2. You can use the section filters or the filtering page (Fig. 11) with the **[Show in the section analytics view]** checkbox selected (Fig. 2, 2).

To include information from several section records in your report (Fig. 13):

1. Open the needed section.
2. Apply filters if needed.
3. Click **[Actions] → [Select multiple records]**.
4. Select the needed report in the **[Print]** button drop-down list.

Fig. 13. – Receiving the "Account Info" report from several records of the [Accounts] section

The screenshot shows the Creatio platform interface. On the left is a dark sidebar with various icons and labels: Sales, Dashboards, Feed, Leads, Accounts, Contacts, Activities, Opportunities, Orders, Contracts, Invoices, and Documents. The main area has a light background. At the top left is the 'Creatio' logo and the date '7.16.2015'. Below it is a green navigation bar with 'Processes' selected, followed by 'Process library' and 'Process log'. To the right of the navigation bar is a banner for 'Серія ACCELERATE ONLINE 2020' with a red arrow icon. The banner text is in Ukrainian: 'На вас чекають 50+ інтерактивних онлайн-зусилей у різних сферах з бізнес- та ІТ-підтримкою! Присвітлюється, щоб дізнатися про підходи, інструменти та технології, які допомагають компаніям пристосуватися у новій цифровій реальності!' Below the banner is a blue button labeled 'ДІЗНАТИСЯ БІЛЬШЕ'. Further down are links to 'Developer's guide to Creatio platform', 'Lightning fast implementation', 'Getting started', 'Academy', and 'Community'. A vertical sidebar on the right contains icons for user profile, gear, question mark, phone, email, messaging, notifications, and a help icon.

The report looks as follows (Fig. 14):

Fig. 14. – Example of the "Account Info" report based on several section records

Account Info

Name	Logo
Vertigo Systems	 The logo consists of three overlapping blue circles of increasing size from top to bottom, with the text 'VERTIGO SYSTEMS' in a sans-serif font below them.
RealWay	 The logo features a stylized red and black 'R' shape followed by the word 'RealWay' in a bold, red, sans-serif font.

See also

- The "Report setup" section
- Setting up reports in Creatio
- Setting up the report

Phone integration

Contents

- **Introduction**
- **Oktell**
- **Webitel**
- **Asterisk**

Phone integration

Beginner

Easy

Medium

Advanced

General provisions

Creatio can be integrated with a number of [automatic telephone exchanges](#) ([Private Branch Exchange](#), PBX), which enables users to manage calls directly in Creatio UI. Phone integration functions are available in the form of a CTI ([Computer Telephony Integration](#)) panel, as well as the [Calls] section. Standard CTI panel functions:

- Displaying incoming calls with contact/account identification by the subscriber's phone number
- One-click calls initiated from Creatio UI
- Call management (reply, place on hold, end or transfer call)
- Displaying call history for managing connections of calls to various system records and call follow-up.

All calls made or received are stored in the [Calls] section. In this section, you can view when a call was started, when it ended and how long the call was; as well as the list of system records connected to the call.

By default, Creatio cloud has a function for making calls between system users without using any additional software.

Depending on the integrated phone system and specifics of its API ([Application Programming Interface](#)), different architectural mechanisms are used. The API also affects available phone integration functions. For example, the call playback function is not available for all phone systems, the web phone is available when integrating with Webitel, etc. Regardless of the phone integration mechanism being used, the CTI panel interface remains the same for all Creatio users.

Phone integration methods in Creatio

There are two types of integration methods: *first party* and *third party* integrations.

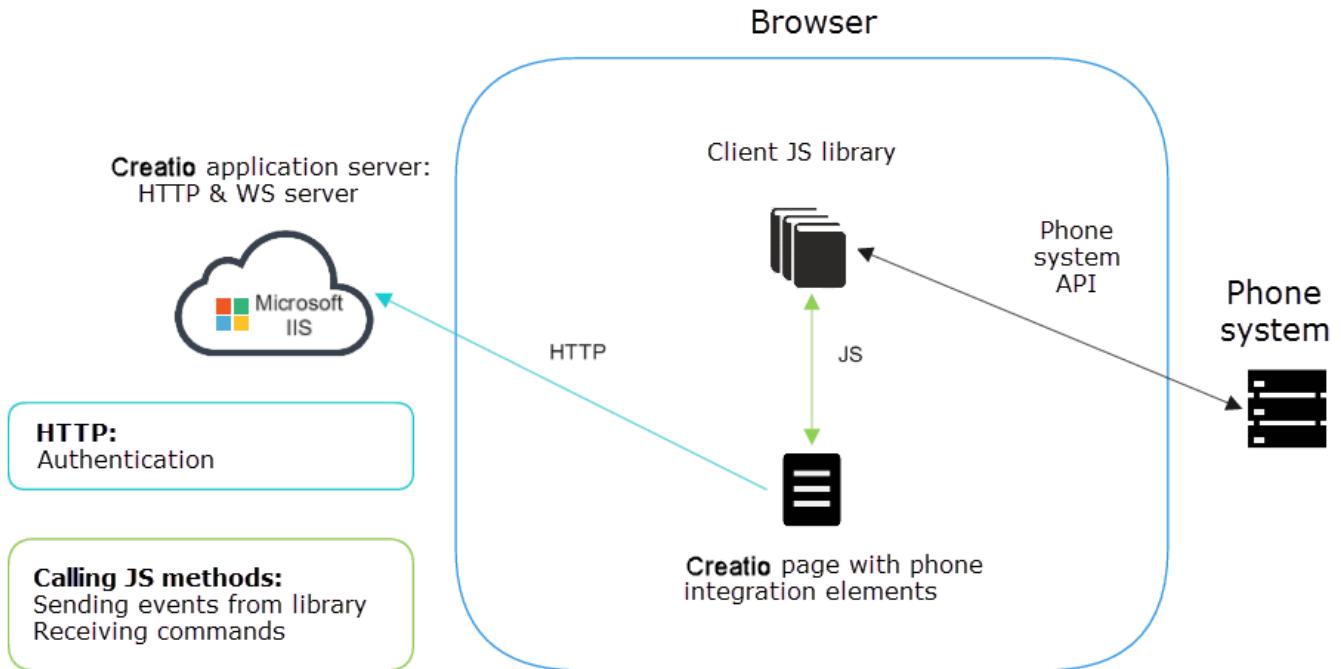
In a *first party* integration each user has a separate integration connection. Phone system events are handled as part of that connection.

For a *third party* integration, a single connection to the phone system server is used for handling phone system events for all users. In a third party integration an intermediate *Messaging Service* link is used for distributing information streams for all users.

JavaScript adapter on the client side

When integrating with JavaScript adapter on the client side (Fig. 1), the work with the phone system is done directly from a web browser. Interactions with the phone system and JavaScript-library, usually supplied by the phone system manufacturer, is done through the phone system API. The library broadcasts events and accepts execution commands using JavaScript. In the context of this integration, the Creatio page interacts with the application server for authentication using the HTTP(S) protocol.

Fig. 1. First party phone system API integration with a javascript adapter on the client side



This integration method can be used with a first party phone system API, such as Webitel, Oktell, Finesse. Webitel and Oktell connectors use [WebSocket](#) as connection protocol, while the Finesse connector uses [long-polling](#) http queries.

The advantage of the *first party* integration method is that it does not require any additional nodes, such as Messaging Service. Using an integration library, the CTI panel connects directly to the phone system server API from a browser on the user's PC (Fig. 1).

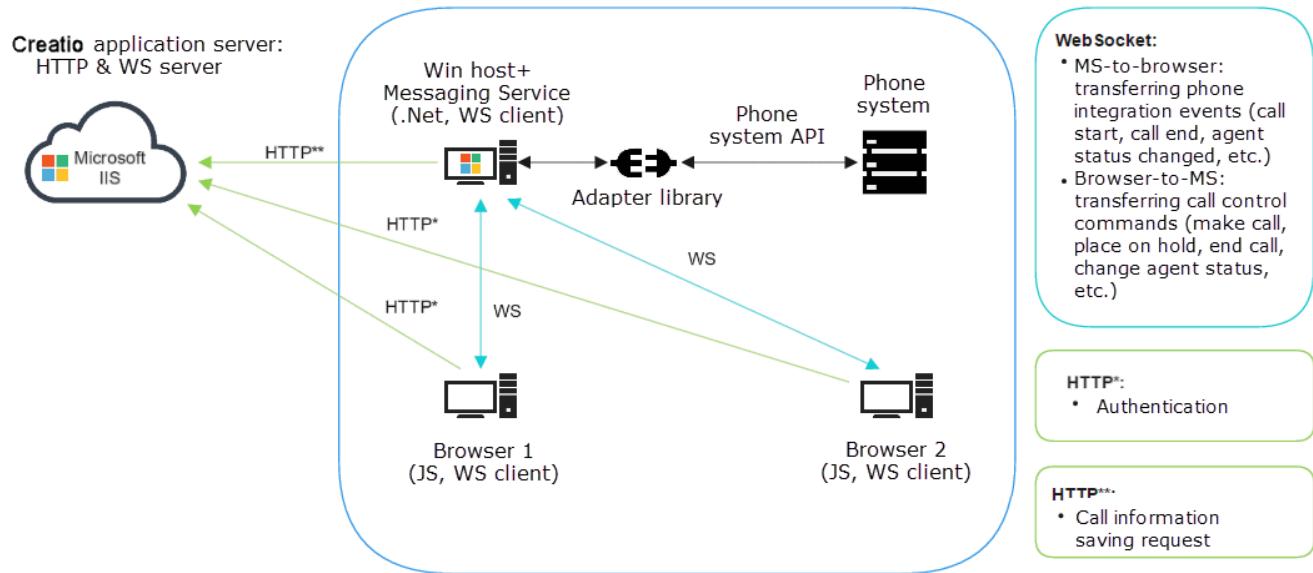
For incoming calls the phone server passes the new call start event and call parameters through WebSocket to the client integration library. When receiving a new call command, the library generates the *RingStarted* event that is passed to the application page.

For incoming calls, client part generates the call start command that is passed through WebSocket to the phone integration server.

Terrasoft Messaging Service on the server side

If integrating with Terrasoft Messaging Service (TMS) on the server side (Fig. 2), all phone integration events pass through TMS, which interacts with the phone system through the manufacturer's library. The library interacts with the phone system through the API. TMS also interacts with the Creatio application server for executing query for saving call information in the database using HTTP(S). Interaction with a client application, such as passing events and receiving commands, is done via WebSocket. In case of integration with JavaScript adapter on the client side, Creatio page interacts with the application server for authentication, using HTTP(S).

Fig. 2. Third party API integration with TMS on server side



This integration method applies to third party phone system API (TAPI, TSAPI, New Infinity protocol, WebSocket Oktell). This integration type requires *Messaging Service* – a Windows proxy service that works with the phone system adapter library. The *Messaging Service* is a universal phone system library hoster, such as Asterisk, Avaya, Callway, Ctios, Infinity, Infra, Tapi. When receiving client messages, the *Messaging Service* automatically connects used Creatio library and initiates connection to phone system. The *Messaging Service* is essentially a functional wrapper for those phone integration connectors that do not support client integration for interacting with phone functions in browsers (event generation and handling, data transfer). A user's PC conducts two types of communication:

- HTTP connection with Creatio application server for authentication with host on which the *Messaging Service* is installed
- WebSocket connection for working directly with phone integration (Fig. 2).

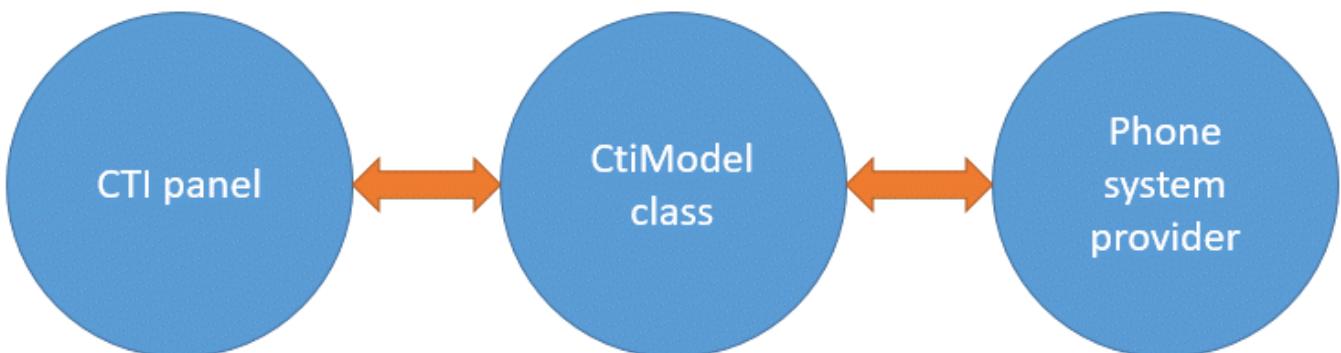
For incoming calls the phone system passes the new call start event and call parameters through the adapter library. When receiving a new call command, the *Messaging Service* generates the *RingStarted* event that is passed to the client.

For an outgoing calls, the client generates a call start command, which is passed via WebSocket to the *Messaging Service*, which generates an outgoing call message for the phone system.

Interaction between the phone connectors and Creatio

All connectors interact with configuration through the *CtiModel* class. It handles the events received from the connector.

Fig. 3. Phone system component interaction with Creatio



The list of supported class events is provided in table 1.

Table 1. Supported events of the CtiModel class

Event	Description
initialized	Triggered on completion of provider initialization.
disconnected	Triggered on provider disconnection.
callStarted	Triggered at the start of a new call.
callFinished	Triggered after call completion.
commutationStarted	Triggered after establishing call connection.
callBusy	Triggered on changing call status to "busy" (TAPI only).
hold	Triggered after placing call on hold.
unhold	Triggered after resuming a call.
error	is triggered on errors.
lineStateChanged	Triggered after changing available operations for a line or a call.
agentStateChanged	Triggered on changing the agent status.
activeCallSnapshot	Triggered on updating the list of active calls.
callSaved	Triggered after creating or updating a call in the database.
rawMessage	Generic provider event. Triggered on any provider event.
currentCallChanged	Triggered on changing the main call. For example, primary call ends during a consultation.
callCentreStateChanged	Triggered if an agent enters or exits Call center mode.
callInfoChanged	Triggered on modifying a call data by database Id.
dtmfEntered	Triggered if Dtmf signals were sent to the phone line.
webRtcStarted	Triggered on a webRtc session start.
webRtcVideoStarted	Triggered on a webRtc video stream session start.
webRtcDestroyed	Triggered on a webRtc session end.

Oktell

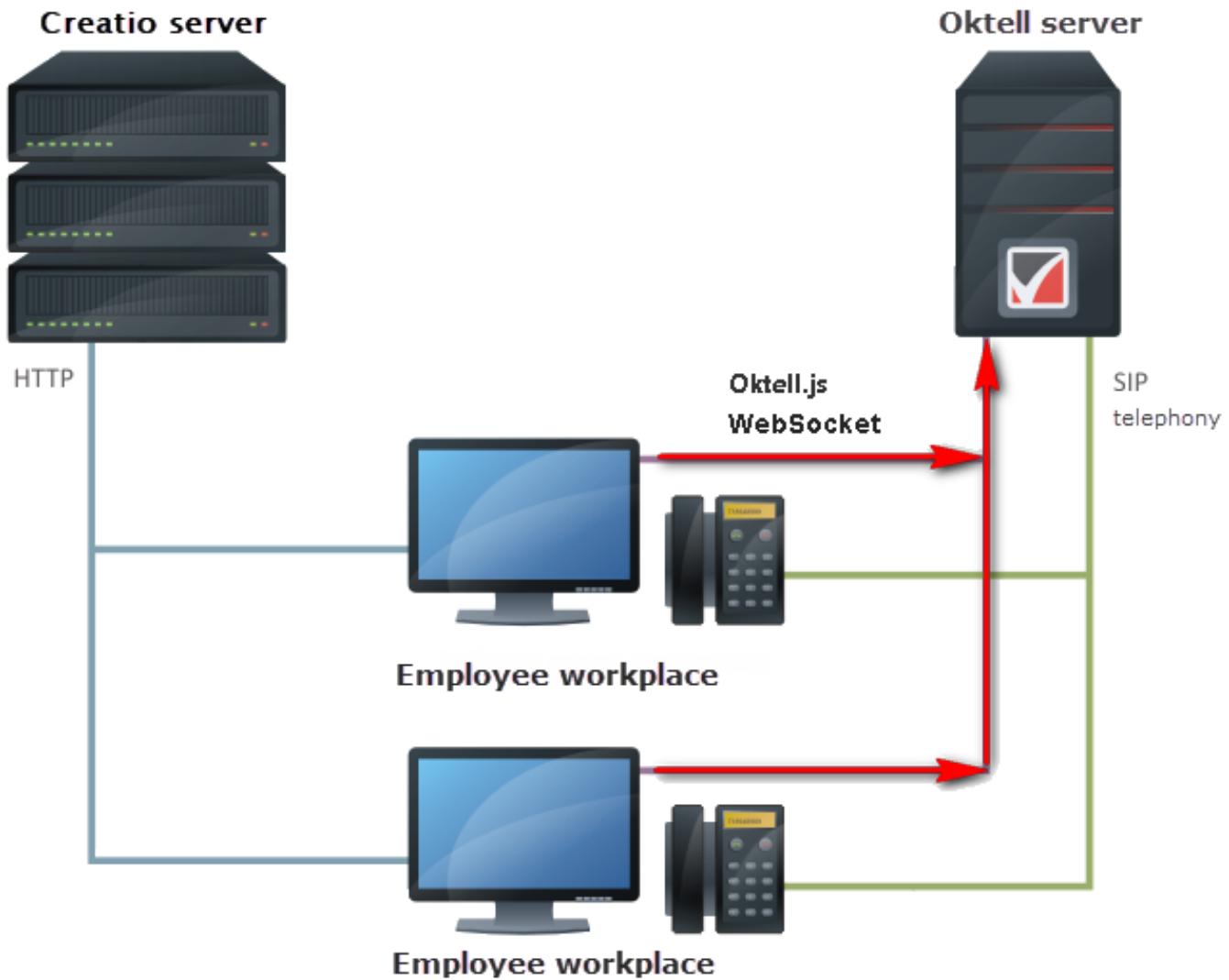
Beginner Easy Medium **Advanced**

General information

Oktell integration with Creatio is implemented on the client level using the `oktell.js` library. The `oktell.js` source code is located in the *OktellModule* configuration schema of the *CTIBase* package.

The Oktell server communicates with phones and with the end clients (browsers). With this integration method Creatio does not require its own WebSocket server. Each client connects via the WebSocket Protocol directly to the Oktell server. The Creatio application server creates pages and provides data from the application database. There is no direct relationship between Creatio and Oktell server. Access is not required, so customers process and combine the data of the two systems independently. The Oktell web client and the `oktell.js` plugin, embedded in other projects, are implemented according to this principle (Fig. 1).

Fig. 1. Oktell integration with Creatio schema



Oktell.js

Oktell.js is a javascript library for embedding the functionality of the call control in a CRM system. Oktell.js uses the Oktell WebSocket Protocol to connect to the Oktell server. The advantage of this Protocol is the establishing of a permanent asynchronous connection to the server, which enables you to receive events from the server Oktell and execute certain commands. Because the Oktell WebSocket protocol is quite complicated to implement, the Oktell.js wraps the WebSocket Protocol methods inside itself thus providing simple management functionality.

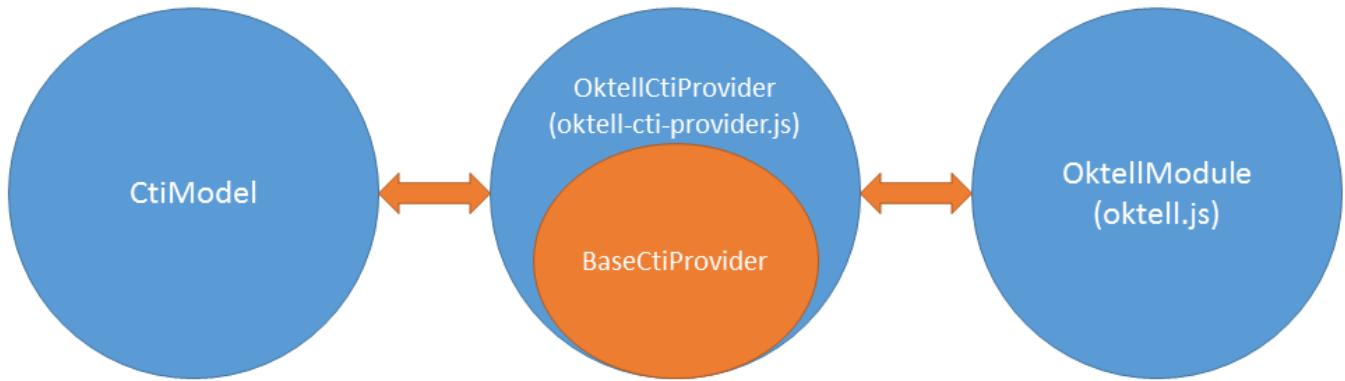
Voice transmission between subscribers

In a conversation between the oktell and Creatio operators, voice is transmitted via the [Session Initiation Protocol \(SIP\)](#). This requires that either the [VoIP phone](#) or the [Softphone](#) operator be installed on your computer (Fig. 1).

Interaction of components

The interaction with the oktell.js library is executed via the *OktellCtiProvider* class, which is a link between *CtiModel* and *OktellModule* that contains the oktell.js code. The [*OktellCtiProvider*](#) class implements the [*BaseCtiProvider*](#) interface class (Fig. 2).

Fig. 2. The components interaction schema in the process of Oktell integration with Creatio



Examples of interaction between CtiModel, OktellCtiProvider and OktellModule are displayed on Fig. 3 and Fig. 4.

Fig. 3. Operator outgoing call to a subscriber: putting a call on hold, putting off hold by a subscriber and finishing the call by the operator.

(scr_oktell_events_01.png in the on-line documentation)

Fig. 4. Incoming call of a subscriber 1 to an operator with a consultation call to subscriber 2 with the subsequent connection of the subscriber 1 and subscriber 2 by the operator.

(scr_oktell_events_02.png in the on-line documentation)

The list of supported oktell.js class library events is listed in table 1.

Table 1. The list of supported oktell.js class library events

Event	Description
connect	Successful connection to server event
connectError	Connection to server error in the 'connect' method event. Error codes are the same as for the callback function of the 'connect' method
disconnect	Server connection closing event. The object describing the reason of the disconnection is passed to the callback function.
statusChange	Agent status change event. Two string parameters are passed to the callback function – the new and previous state
ringStart	Incoming call start event
ringStop	Incoming call stop event
backRingStart	Returning call start event
backRingStop	Returning call stop event
callStart	Outgoing call start event
callStop	Call UUID change event
talkStart	Conversation start event.
talkStop	Conversation stop event.
holdAbonentLeave	Caller hold leave event The abonent object is passed to the callback function with information on the caller.
holdAbonentEnter	Caller hold enter event The abonent object is passed to the callback function with information on the caller.
holdStateChange	Hold status change event. The information on the hold is passed to the hold function.
stateChange	Line status change event.
abonentsChange	Current abonents list change event
flashstatechanged	Hold status change low-level event

userstatechanged

User status change low-level event

Wabitel

Beginner

Easy

Medium

Advanced

General information

[Wabitel](#) integration is implemented in the form of separate Creatio modules. Modules in the integration include:

The WabitelCore package — modules that contain low-level interactions with Wabitel using the Verto module and the CTI panel on the Creatio application page.

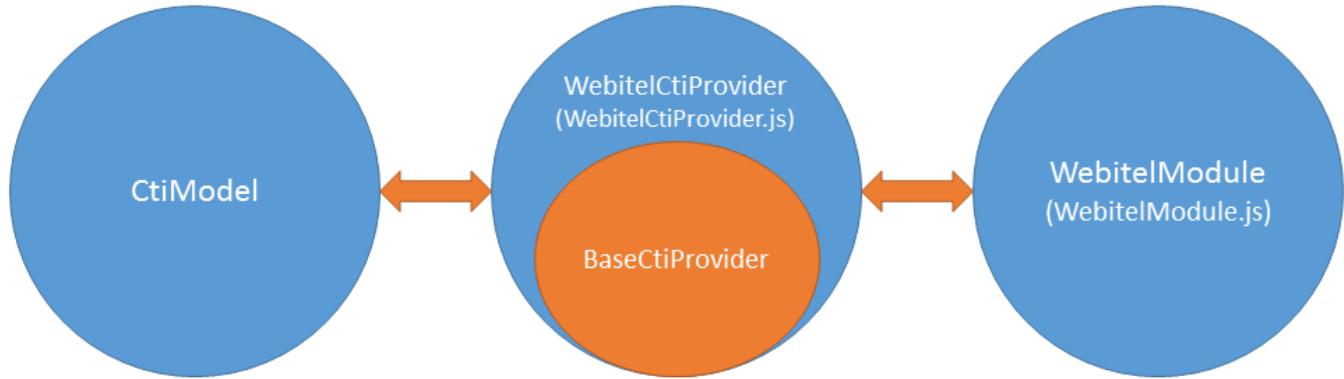
The WabitelCollaborations package implements basic interfaces for working with Wabitel in Creatio. The package contains the *WabitelCtiProvider* module, the *WabitelCtiProvider* class, Wabitel connector, the connection parameters settings page, the lookup to edit Wabitel users directly in Creatio.

Detailed information about Wabitel architecture can be found in the [documentation](#).

Interaction of components

The *WabitelCtiProvider* class (the heir of the *Terrasoft.BaseCtiProvider* class) implements the required interaction between CtiModel and the Wabitel low-level global object (the *WabitelCore.WabitelModule.js* module) (Fig. 1).

Fig. 1. The components interaction schema in the process of Wabitel integration with Creatio



The integration is as follows. If a user has set the system setting of the Wabitel integration library, *CtiProviderInitializer* loads the *WabitelCtiProvider* module. Next, it calls the *init* method in *WabitelCtiProvider*, which carries out the user login in the telephony session (the *LogInMsgServer* of the *MsgUtilService.svc* service). If the login was successful, the *connect* method is invoked, which verifies that you don't have an existing connection (the *this.isConnected* property is set to *false* and *this.wabitel* — to empty). After that, the *connect* method requests the connection settings to Wabitel that are stored in the system settings of the *webitelConnectionString* and *webitelWebrtcConnectionString* (Fig. 2).

Fig. 2. Loading the WabitelCtiProvider provider and connecting the CTI panel

(scr_wabitel_events_01.png in the on-line documentation)

After receiving the system settings, the user settings are received from the [Wabitel users] lookup by using the *GetUserConnection* method of the *WUserConnectionService* customer service. After receiving the user settings, the *WabitelModule* and *WabitelVerto* are loaded if the [Use web phone] checkbox is selected in the user settings. Next, the *onConnected* method is called that creates the *Wabitel* global object, in which properties are populated with the connection settings. The subscription to *Wabitel* object events occurs and the *connect* method is invoked, which performs connection via *WebSocket*, authorization of Wabitel and other low-level connection operations. When the *onConnect* event occurs, the connection is considered successful and the user can work with calls. During the connector operation, *WabitelCtiProvider* reacts to *Wabitel* object events, processes them, and optionally generates connector events described in the *Terrasoft.BaseCtiProvider* class. To manage calls, *WabitelCtiProvider* implements abstract methods of the *Terrasoft.BaseCtiProvider* class by using the *Wabitel* object methods.

Examples of CtiPanel, CtiModel and WebitelCtiProvider interaction

Outgoing call to a subscriber: putting a call on hold, taking a call off hold by a subscriber and finishing a call.

Fig. 3. Sequence of events during a call

('scr_webitel_events_o2.png' in the on-line documentation)

Webitel list of ports

- 871 – the WebSocket port for the Webitel server and receiving events.
- 5060 and 5080 – signal ports for SIP phones and telephony providers.
- 5066 – the port for the Web phone and WebRTC signal port.
- 4004 – the port for receiving call records.

Webitel events

Table 1. WebitelCtiProvider events

Event	Description
onNewCall	New call start event.
onAcceptCall	Accept call event.
onHoldCall	Call hold event.
onUnholdCall	Call Unhold event.
onDtmfCall	Tone dialing event.
onBridgeCall	Connection to channel event.
onUuidCall	Call UUID change event
onHangupCall	Call stop event.
onNewWebRTCCall	New WebRTC session event.

Asterisk

Beginner

Easy

Medium

Advanced

General information

Use AMI ([Asterisk Manager Interface](#)) to interact with the [Asterisk](#) server. The API enables client programs to connect to Asterisk server by using TCP/IP protocol. The [Application Programming Interface](#) enables you to process events in the digital multiplex system (DMS), and send commands to control calls.

Currently the integration of Creatio with Asterisk is supported up to version 11.

A client uses a simple text protocol for communication between the Asterisk and the connected Manager API: "parameter: value". The end of a string is determined by the sequence of Carriage Return and Line Feed ([CRLF](#)).

In the future, for a set of strings like "parameter: value", followed by a blank line containing only a CRLF, for simplicity the term "package" will be used.

How to set up the configuration file of the Messaging Service to integrate Asterisk to Creatio

For integration to work with Creatio, you need to install Terrasoft Messaging Service (TMS) on a dedicated computer that will be used as the integration server. You must set the following parameters for Asterisk in the *Terrasoft.Messaging.Service.exe.config* configuration file:

```
<asterisk filePath="" url="Name_or_address_of_Asterisk_server"
port="Asterisk_server_port" userName="Asterisk login" secret="Asterisk password"
```

```
originateContext="Originate context" parkingLotContext="Parking lot context"
autoPauseOnCommutationStart="true" queueExtensionFormat="Local/{0}@from-queue/n"
asyncOriginate="true" sendRingStartedOnRingingState="true" traceQueuesState="false"
packetInfoConfig="Additional package parameters to be processed in configuration" />
```

Ports for Asterisk integration with Creatio

- TMS accepts WebSocket connection to the 2013 port via TCP.
- TMS connects to the Asterisk server by default via the 5038 port.

The Terrasoft Messaging Service for Asterisk integration with Creatio

The integration part of the Messaging Service is implemented in the main Creatio solution kernel in the *Terrasoft.Messaging.Asterisk* library.

Library main classes:

- *AsteriskAdapter* — an Asterisk class that transforms events to the top-level call model events used in Creatio integration.
- *AsteriskManager* — a class that creates and deletes user connections to the Asterisk server.
- *AsteriskConnection* — a class that represents the user connection for integration with Asterisk.
- *AsteriskClient* — a class used to send commands to the Asterisk server.

Example of CtiModel, Terrasoft Messaging Service and Asterisk Manager API interaction

Operator outgoing call to a subscriber: putting a call on hold, putting off hold by a subscriber and finishing the call by the operator.

Fig. 1 shows the occurrence of events for this example While table 1 shows an example of processing of events — how these events are interpreted by the TMS, which values from these events are used in processing an incoming call.

Fig. 3. Sequence of events during a call

('scr_asterisk_events.png' in the on-line documentation)

Table 1. Asterisk events

Asterisk log	TMS	Action	Client
{ Event: newchannel Channel: <channel_name> UniqueID: <unique_id> }	A channel is created and added to a collection <i>new AsteriskChannel({ Name: <channel_name>, UniqueId: <unique_id> });</i>		
{ Event: Hold UniqueID: <unique_id> Status: "On" } { Event: Hold UniqueID: <unique_id> Status: "Off" }	Search for the channel by <unique_id> and generate an event by using the fireEvent method. PutHoldAction	PutHoldAction	Processing the PutHoldAction and displaying the call on hold.
	Search for the channel by <unique_id> and generate an event by using the fireEvent method.	EndHoldAction	Processing the EndHoldAction and displaying the call on hold.

{	Search for the channel by <unique_id> and generate an event by using the fireEvent method.	RingFinished	Processing event and displaying the call end.
Event: Hangup			
UniqueID: <unique_id>			
}			
{	Search for the channel by <unique_id>, fill in the data and generate an event by using the <i>fireEvent</i> method.	RingStarted	Processing the RingStarted event and displaying it on the outgoing call panel.
Event: Dial			
SubEvent: Begin			
UniqueID: <unique_id>			
}			
{	Search for the channel by <unique_id> and generate an event by using the fireEvent method.	CommutationStarted	Processing the CommunicationStarted event and displaying the communication.
Event: Bridge			
UniqueID: <unique_id>			
}			
			Clicking the "Answer" button initiates a new event in Asterisk.

Asterisk events

A detail list of events and information about their parameters is described in the [Asterisk documentation](#).

Creatio marketing

Contents

- **Campaign elements**

Campaign elements

Beginner

Easy

Medium

Advanced

Introduction

Marketing campaign diagrams are created in a visual campaign designer in the [\[Campaigns\] section](#). The campaign diagram consists of campaign elements and transitions (flows).

Once the campaign is launched, the flow-schema of the campaign is created. The campaign elements are converted to a campaign execution chain and the start time is calculated for each element. The flow-schema can be significantly different from the visual campaign diagram in the designer.

Campaign elements can be synchronous and asynchronous.

Synchronous elements are executed according to the order specified in the flow-schema. The transition to the subsequent elements is performed once the synchronous element is executed. The execution flow is blocked and waits for the operation to complete.

Asynchronous elements wait for the finished execution of certain external systems, resources, asynchronous services, or user reactions (e.g., clicking a link in an email).

Their position in the flow-schema is determined by their element type. The [Add from folder] and [Exit according to folder conditions] elements are executed first. These elements are used to add or remove participants from the campaign audience. Campaign participants are moving from one element to the other through the flows. If the flow has certain configured conditions, the system filters the participants based on these conditions and determines the

execution time of the subsequent element.

The mechanism for planning the next campaign launch

The following is the algorithm for planning the next campaign launch:

1. The time of the next launch of an element is determined by the configured delay.

- The “In a day” option is selected. The date and time of the next execution of this element is calculated with the help of the following formula:

Date and time of execution = current date and time + N minutes / hour,

where N is the value of the [Number of days] field, populated by the user.

- The “Few days” option is selected. The next execution of this element is performed with the help of the following formula:

Date = [current date+ N days],

where N is the value of the [Number of days] field, populated by the user.

Execution time = time specified by the user.

- The “No, execute after the previous one” option is selected. The next execution of this element is performed at the time of the next launch of the campaign.

2. According to the variant described in paragraph 1, the launch time for each element of the campaign scheme is calculated.

3. Upon comparing all values, the closest launch time selected and set as the campaign launch time.

4. Forming a list of elements, which will be executed upon next launch. The list contains all elements, the launch time of which is the same as the campaign launch time.

Main campaign element classes

JavaScript classes

The base element schema class is *ProcessFlowElementSchema*. The *CampaignBaseCommunicationSchema* is the parent class for all elements in the [Communications] group. The *CampaignBaseAudienceSchema* is the parent class for the [Audience] group of elements.

When creating an element in a new group of elements, it is recommended to implement the base schema of the element first, and then inherit each element from it.

Each schema corresponds to the schema of the element properties edit page. The base edit page schema is *BaseCampaignSchemaElementPage*. Each new element page extends the base page.

The *CampaignSchemaManager* class manages the schemas of elements available in the system. It inherits the main functionality of the *BaseSchemaManager* class.

C# classes

Simple element classes

CampaignSchemaElement – base class. All other elements are inherited from this class.

SequenceFlowElement – base class for the [Sequence flow] element.

ConditionSequenceFlowElement – base class for the [Condition flow] element.

EmailConditionalTransitionElement – transition element class by response.

AddCampaignParticipantElement – add audience (participants) element class.

ExitFromCampaignElement – the class of the audience exit element.

MarketingEmailElement – the class of the Email element.

Executable element classes

CampaignProcessFlowElement – base class. All other executable elements are inherited from this class.

AddCampaignAudienceElement – audience element class.

ExcludeCampaignAudienceElement – the class of the audience exit element.

BulkEmailCampaignElement – the class of the Email element.

Sync Engine synchronization mechanism

Contents

- **Creatio synchronization with external storages**
- **Synchronizing metadata in Creatio**
- **Synchronizing tasks with MS Exchange**
- **Synchronizing email with MS Exchange**
- **Synchronizing contacts with MS Exchange**
- **Synchronizing appointments with MS Exchange**

Creatio synchronization with external storages

Beginner

Easy

Medium

Advanced

General information

The mechanism in Creatio for synchronization with external data storages is the Sync Engine. This mechanism enables you to create, modify, and delete *Entity* in the system based on synchronization with external systems and export data to external systems.

Synchronization is performed by using the *SyncAgent* class implemented in the *Terrasoft.Sync* namespace of the application kernel.

Classes used in the synchronization mechanism

- Synchronization agent (*SyncAgent*) – a class with one public *Synchronize* method that triggers synchronization between storages.
- Synchronization context (*SyncContext*) – a class representing the aggregation of providers and metadata for *SyncAgent*.
- Storage – storage of synchronized data.
 - Local storage (*LocalProvider*) – enables you to work with *LocalItem* in Creatio.
 - External storage (*RemoteProvider*) – an external service or application from which data is synchronized with Creatio.
- Synchronization item (*SyncItem*) – a set of objects from external and local storage which are synchronized.
 - External storage synchronization item (*RemoteItem*) – represents a set of data from external storage that syncs automatically. It can consist of one or more entities (records) from the external storage.
 - Synchronization item (*SyncEntity*) – a wrapper of a specific *Entity*. *SyncEntity* is required to work with *Entity* as the synchronizing object (add, delete, change).
 - Synchronization item (*LocalItem*) – one or more objects from Creatio that are synchronized with the external storage as a unit. The synchronization item from the external storage, converted in the *LocalItem* entity contains a set of instances of the *SyncEntity* class.
- *SysSyncMetadata* metadata table – contains service information of the synchronized elements and is essentially *RemoteItem-LocalItem* interchanges table. Metadata sync description can be found in the "**Synchronizing metadata in Creatio**" article.

General synchronization algorythm

Before starting synchronization, you must create an instance of *SyncAgent* and the *SyncContext* sync context, then update records in the metadata table with data from Creatio. For this, you need to call the *CollectChangesInSyncedEntities* class method that implements the *IReplicaMetadata* interface.

The algorithm for updating metadata records is the following:

1. If any previously synchronized entity in Creatio has been modified since the last synchronization, then the corresponding record in the metadata changes its modification date, the *LocalState* property is set to "Modified", and the source of the modification is set to the *LocalStore* ID.
2. If a synchronized entity in Creatio has been deleted since the last synchronization — the corresponding record in the metadata *LocalState* is set to "Deleted".
3. If there is no corresponding record in the metadata for the entity in Creatio — it is ignored.

The process of synchronizing storages then starts the following:

1. All changes from the external storage are requested alternately.
2. You need to obtain the metadata for each item in the external storage.
 - a. If the metadata can not be obtained — this is a new item which should be converted to a Creatio element. To fill in a synchronization object, a *FillLocalItem* method is called from the specific *RemoteItem* instance. It is also recorded in the metadata (ID of the external storage, the element ID in the external storage, date of creation and modification is set as current, the source of creation and modification — external storage).
 - b. If the metadata is received, so this item has already been synchronized with Creatio. You need to go to the version conflict resolution. By default, the last change in the application or external storage (*RemoteProvider*) has the priority.
 - c. The metadata for the current pair of synchronization items is actualized.

After looking through all the modified items from the external storage, the elements that were changed in Creatio, but was not changed in the external storage remain in the metadata (in the interval between the last and the current synchronization).

1. You need to get elements changed in Creatio in the interval between the last synchronization and the current synchronization.
2. Save the changes in the external storage.
3. You must update the modification date of the items in the metadata (Creatio is the change source).

After that, you need to add new, not synchronized records to the external storage, and add metadata for new items.

The synchronization context

SyncContext class

A class representing the aggregation of providers and metadata for *SyncAgent*. The properties of the *SyncContext* class are presented in Table 1.

Table 1. *SyncContext* class properties

Field	Type	Purpose
<i>Logger</i>	<i>ISyncLogger</i>	The object that enables messages to be saved into the integration log.
<i>LocalProvider</i>	<i>LocalProvider</i>	Enables you to work with <i>LocalItem</i> .
<i>RemoteProvider</i>	<i>RemoteProvider</i>	External service or application, data from which is synchronized with Creatio.
<i>ReplicaMetadata</i>	<i>IReplicaMetadata</i>	Works with metadata.
<i>LastSyncVersion</i>	<i>DateTime</i>	The date and time of the last synchronization in UTC.
<i>CurrentSyncStartVersion</i>	<i>DateTime</i>	The current date and time synchronization in UTC. Set after the metadata update.

Requirements for synchronization with external storage

External storage

External storage (*RemoteProvider*) — encapsulates data from the external storage.

RemoteProvider — a basic class that enables you to work with an external storage. In fact, it is the only way to work with external systems. Properties of the *RemoteProvider* class are presented in table 2 and the methods — in table 3.

Table 2. *RemoteProvider* class properties

Field	Type	Description
<i>StoreId</i>	<i>Guid</i>	ID of external storage that will be synchronized.
<i>Version</i>	<i>DateTime</i>	Date and time of the last synchronization in UTC.
<i>SyncItemSchemaCollection</i>	<i>List</i>	External storage element schema
<i>RemoteChangesCount</i>	<i>Int</i>	Number of items processed from external storage.
<i>LocalChangesCount</i>	<i>Int</i>	Number of items processed from local storage.

Table 3. *RemoteProvider* class methods

Method	Returning value type	Description
<i>KnownTypes()</i>	<i>IEnumerable</i>	Returns a collection of all types that implement the <i>IRemoteItem</i> interface. <i>SyncAgent</i> builds the <i>SyncItemSchema</i> instances that describe the entities that participate in synchronization.
<i>ApplyChanges(SyncContext context, IRemoteItem synItem)</i>	<i>Void</i>	Applies changes to external storage element.
<i>CommitChanges(SyncContext context)</i>	<i>Void</i>	Called after processing changes in external and local storage. Intended for the implementation of necessary additional steps for the specific integration implementation.
<i>EnumerateChanges(SyncContext context)</i>	<i>IEnumerable</i>	Returns an enumeration of new and modified elements of external storage.
<i>LoadSyncItem(SyncItemSchema schema, string id)</i>	<i>IRemoteItem</i>	Fills in the <i>IRemoteItem</i> instance with data from external storage.
<i>CreateNewSyncItem(SyncItemSchema schema)</i>	<i>IRemoteItem</i>	Creates a new instance of <i>IRemoteItem</i> .
<i>CollectNewItem(SyncContext context)</i>	<i>IEnumerable</i>	Returns an enumeration of new Creatio entities that will be synchronized with external storage.
<i>ResolveConflict(IRemoteItem syncItem, ItemMetadata itemMetaData, Guid localStoreId)</i>	<i>SyncConflictResolution</i>	Resolves conflicts between changed elements of local and external storages. By default, (<i>RemoteProvider</i>) priority is given to changes in Creatio.

<i>NeedMetaDataActualization()</i>	<i>Boolean</i>	Returns the sign that checks whether there is a need to update the metadata before starting synchronization.
<i>GetLocallyModifiedItemsMetadata(SyncContext context, EntitySchemaQuery modifiedItemsEsq)</i>	<i>IEnumerable</i>	Returns synchronization elements changed in the local store since the last synchronization.

IRemoteItem interface

The class that implements the *IRemoteItem* interface is an indivisible unit of synchronization and represents one element of the synchronization of the external data storage. This class is a container for data coming from an external system, and it knows how to convert the data to the *Entity* entity, and vice versa. The interface contains two methods: *FillLocalItem* and *FillRemoteItem* for converting external synchronization elements (*RemoteItem*) to *LocalItem*, and vice versa. Interface methods are presented in Table 4.

Table 4. IRemoteItem interface methods

Method	Returning value type	Description
<i>FillLocalItem(SyncContext context, ref LocalItem localItem)</i>	<i>Void</i>	Fills in properties of an element of the <i>LocalItem</i> local storage with values of external storage element. Used to apply changes in local storage.
<i>FillRemoteItem(SyncContext context, ref LocalItem localItem)</i>	<i>Void</i>	Fills in properties of an element of external storage with element values from the <i>LocalItem</i> local storage. Used to apply changes in external storage.

Map attribute

The *Map* attribute decorates the *iRemoteItem* interface implementations. *SchemaName* is the main parameter. This is the name of the *EntitySchema* that is included in the current synchronization element.

```
[Map("Activity", 0)
[Map("ActivityParticipant", 1)]
public class GoogleTask: IRemoteItem {
    . . .
```

This class declaration task from Google Calendar will sync with the activity and its participants from Creatio.

The second parameter, *Order*, specifies in which order *Entity* will be stored in the local storage. *Activity* is indicated first, because *ActivityParticipant* stores a link to the created activity.

In most cases, *SyncAgent* can automatically generate a request for a sample of the new elements of synchronization with Creatio. To do this, you must specify the basic entity and method of communication with the details:

```
[Map("Activity", 0, IsPrimarySchema = true)
[Map("ActivityParticipant", 1, PrimarySchemaName = "Activity", ForeingKeyColumnName =
"Activity")]
public class GoogleTask: IRemoteItem {
    . . .
```

In this case, a request for new activities will be sent to the database along with a request for each selected activity to receive their participants. The attribute properties list is presented in Table 5.

Table 5. Map attribute properties

Parameter	Type	Description
<i>SchemaName</i>	<i>String</i>	Object schema name.

<i>Order</i>	<i>Int</i>	The entity processing order for the synchronization element.
<i>IsPrimarySchema</i>	<i>Boolean</i>	A flag that indicates that this schema is a key element of this synchronization element. It can be installed in one schema only.
<i>PrimarySchemaName</i>	<i>String</i>	The schema name of the main object. It can not be set in tandem with <i>IsPrimarySchema</i> .
<i>ForeignKeyColumnName</i>	<i>String</i>	The column name for the connection with the details of the main object. It can not be set in tandem with <i>IsPrimarySchema</i> .
<i>Direction</i>	<i>SyncDirection</i>	<p>It specifies the synchronization direction for the objects of this type. By default - <i>DownloadAndUpload</i>.</p> <p>If it contains the <i>Download</i> flag - the changes will not apply to Creatio.</p> <p>If it does not contain the <i>Upload</i> flag - the new entities will not be selected from Creatio.</p>
<i>FetchColumnNames</i>	<i>String[]</i>	The names of the columns that will be loaded from the local storage.

Local storage

Local Storage (*LocalProvider*) - encapsulates the work with data in internal storage (Creatio).

LocalProvider - basic class that implements work with the local storage. Enables you to work with *LocalItem*. Methods of this class are immutable and are listed in Table 6.

Table 6. *LocalProvider* class methods

Method	Returning value type	Description
<i>AddItemSchemaColumns(EntitySchemaQuery esqForFetching, EntityConfig entityConfig)</i>	<i>Void</i>	Adds <i>EntitySchemaQuery</i> column specified in <i>EntityConfig</i> .
<i>ApplyChanges(SyncContext context, LocalItem entities)</i>	<i>Void</i>	Applies changes to each <i>SyncEntity</i> in <i>LocalItem</i> .
<i>FetchItem(ItemMetadata itemMetaData, SyncItemSchema itemSchema, bool loadAllColumns = false)</i>	<i>LocalItem</i>	Loads a collection of entities associated with a particular synchronization element.

SyncEntity class

The class encapsulates SyncEntity Entity instance and all the necessary actions to perform the synchronization of the instance properties. Class Properties are summarized in Table 7.

Table 7. *SyncEntity* class properties

Parameter	Type	Description
<i>EntitySchemaName</i>	<i>String</i>	The name of the schema for which the wrapper is created.
<i>Entity</i>	<i>Entity</i>	<i>Entity</i> for which the wrapper is created.
<i>State</i>	<i>SyncState</i>	The last action performed on the entity (0 - not changed, 1 - new, 2 - changed 3 - deleted).

SystemSchema class

Synchronization element entity settings schema. Class properties are shown in Table 8 and methods in Table 9.

Table 8. *SyncItemSchema* class properties

Parameter	Type	Description
<i>SyncValueName</i>	<i>String</i>	Entity type name
<i>SyncValueType</i>	<i>Type</i>	Entity type
<i>PrimaryEntityConfig</i>	<i>EntityConfig</i>	Synchronization element entity settings.
<i>Configs</i>	<i>List</i>	Synchronization element entity settings list.
<i>DetailConfigs</i>	<i>List</i>	Synchronization element detail entity settings list.
<i>Direction</i>	<i>SyncDirection</i>	It specifies the synchronization direction for the objects of this type. By default - <i>DownloadAndUpload</i> .
		If it does not contain the <i>Download</i> flag, changes will not be applied in Creatio.
		If it does not contain the <i>Upload</i> flag, new entities will not be selected from Creatio.

Table 9. *SyncItemSchema* class methods

Method	Returning value type	Description
<i>CreateSyncItemSchema(Type syncValueType)</i>	<i>SyncItemSchema</i>	It creates a configuration element synchronization entity with all the settings of the related entities.
<i>Validate(UserConnection userConnection)</i>	<i>Void</i>	The method checks that <i>EntityConfig</i> is well-formed.
<i>FetchItem(ItemMetadata itemMetaData, SyncItemSchema itemSchema, bool loadAllColumns = false)</i>	<i>LocalItem</i>	If authentication fails, an exception is applied. If the <i>EntityConfig</i> schema name is specified twice, <i>DuplicateDataException</i> is generated. If the name of the defunct schema is specified, <i>InvalidSyncItemSchemaException</i> is generated).
		Loads a collection of entities associated with a particular synchronization element.

EntityConfig Class

Synchronization element entity settings. Class Properties are summarized in Table 10.

Table 10. *EntityConfig* class properties

Parameter	Type	Description
<i>SchemaName</i>	<i>String</i>	Object schema name.
<i>Order</i>	<i>Int</i>	The order of processing entities for a synchronization element. The lower the value, the sooner the entity will be processed in the processing of the synchronization element.
<i>Direction</i>	<i>SyncDirection</i>	It specifies the synchronization direction for the objects of

<i>FetchColumnNames</i>	<i>String[]</i>	this type. By default - <i>DownloadAndUpload</i> . If it does not contain the <i>Download</i> flag, changes will not be applied in Creatio. If it does not contain the <i>Upload</i> flag, new entities will not be selected from Creatio.
		The names of the columns that will be loaded from the local storage. If the value is not specified, it will load all object columns

DetailEntityConfig class

Synchronization element detail entities settings. Class properties are displayed on Table 11.

Table 11. *DetailEntityConfig* class properties

Parameter	Type	Description
<i>PrimarySchemaName</i>	<i>String</i>	Creatio main synchronized entity schema name.
<i>ForeignKeyColumnName</i>	<i>String</i>	The column name for the connection with the details of the main object

LocalItem class

One or more objects from Creatio that are synchronized with external storage as a unit. It contains a set of *SyncEntity* class instances. Class properties are shown in Table 12 and methods in Table 13.

Table 12. *LocalItem* class properties

Parameter	Type	Description
<i>Entities</i>	<i>Dictionary</i> >	The <i>SyncEntity</i> collection that is set in accordance with a <i>SyncItem</i> . It contains a collection of "key-value" pairs, where the key is the schema name, and the value is the <i>SyncEntity</i> collection of the scheme.
<i>Version</i>	<i>DateTime</i>	The highest value of the date and time of the modification of all <i>Entities</i> in LocalItem.
<i>Schema</i>	<i>SyncItemSchema</i>	Synchronization element entity settings schema.

Table 13. *LocalItem* class methods

Method	Returning value type	Description
<i>AddOrReplace(string schemaName, SyncEntity syncEntity)</i>	<i>Void</i>	Adds new <i>SyncEntity</i> to the collection. If <i>SyncEntity</i> with EntityId already exists, it replaces it.
<i>Add(UserConnection userConnection, string schemaName)</i>	<i>SyncEntity</i>	Creates and adds a new <i>SyncEntity</i> collection.

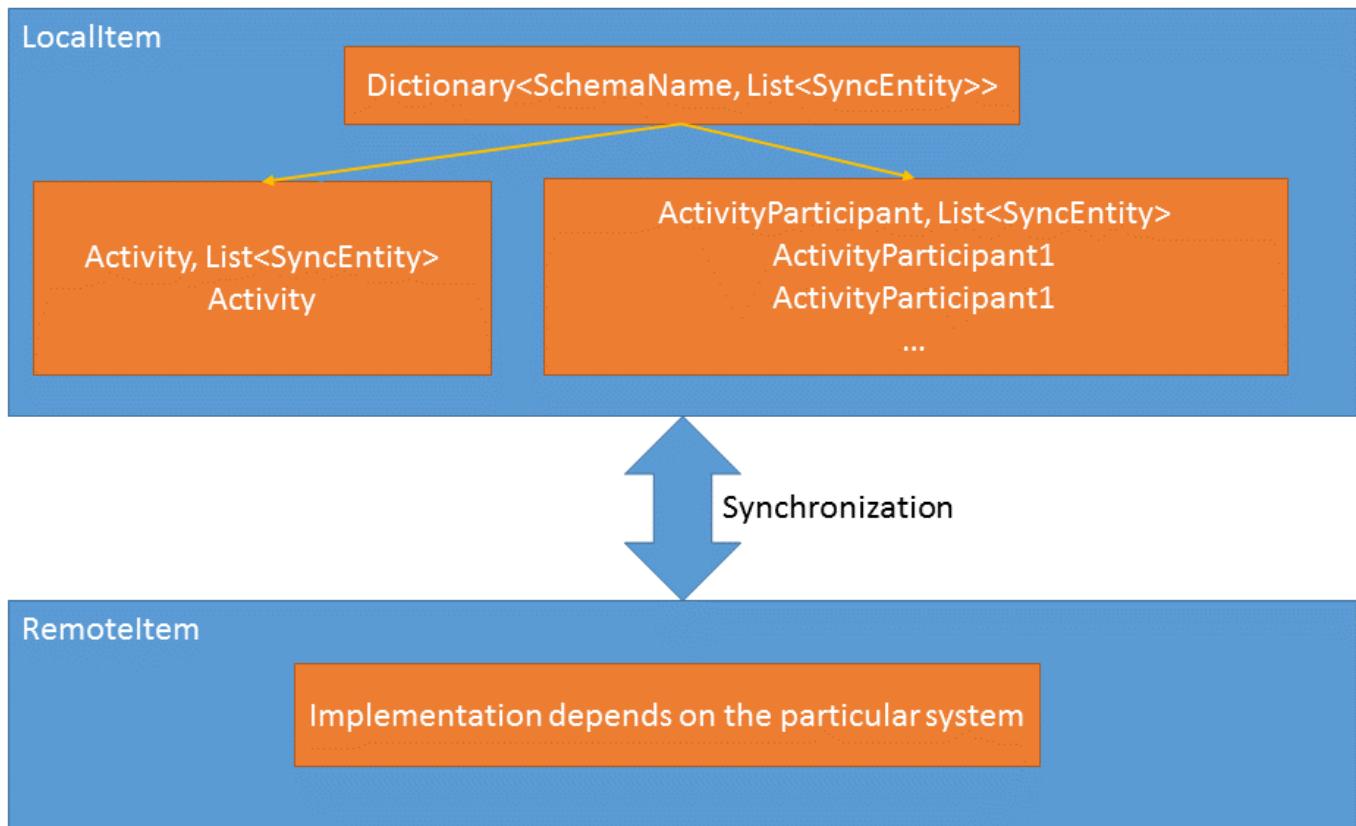
Synchronization example

An activity and participants are synchronized into one Google-calendar task. An activity (*Activity*) and participants (*SyncEntity*) are one element of the synchronization - *LocalItem*.

RemoteItem - Google task received outside Creatio. LocalItem - a set of objects (*SyncEntity*), to which the Google task is converted.

The synchronization schema is displayed on Fig. 1.

Fig. 1. Synchronization schema



Synchronizing metadata in Creatio

Beginner Easy Medium **Advanced**

General information

The auxiliary *SysSyncMetaData* metadata table is used for synchronization, which is the junction between the outer *RemoteItem* table (synchronization element in external storage) and *LocalItem* (synchronization element in Creatio). Each table row is represented in the system as an instance of *SysSyncMetaData*. The *SysSyncMetaData* class properties are shown in Table 1.

Table 1. *SysSyncMetaData* class properties.

Parameter	Type	Description
<i>RemoteId</i>	<i>String</i>	Element ID in external storage
<i>SyncSchemaName</i>	<i>String</i>	Synchronized element schema name.
<i>LocalId</i>	<i>Guid</i>	Element ID in local storage
<i>IsLocalDeleted</i>	<i>Boolean</i>	It indicates whether an item has been removed from the local storage since the last synchronization. The parameter is updated before the synchronization and application of changes in the local storage. On the basis of its value, when selecting modified elements from local storage, the <i>SyncEntity</i> state is set. Obsolete, left for compatibility. <i>LocalState</i> is currently used.
<i>IsRemoteDeleted</i>	<i>Boolean</i>	It indicates whether an item has been removed from the external storage since the last synchronization. Obsolete, left for compatibility. <i>RemoteState</i> is currently used.

<i>Version</i>	<i>Date</i>	Date of the last element modification.
<i>ModifiedInStoreId</i>	<i>Guid</i>	ID of storage in which the last modification was performed.
<i>CreatedInStoreId</i>	<i>Guid</i>	ID of storage in which the synchronization element was created.
<i>RemoteStoreId</i>	<i>Guid</i>	ID of external storage with which the element was synchronized.
<i>ExtraParameters</i>	<i>String</i>	Additional element parameters.
<i>LocalState</i>	<i>Int</i>	Element state in local storage (0 - not modified, 1 - new, 2 - modified, 3 - deleted).
<i>RemoteState</i>	<i>Int</i>	Element state in external storage (0 - not modified, 1 - new, 2 - modified, 3 - deleted).

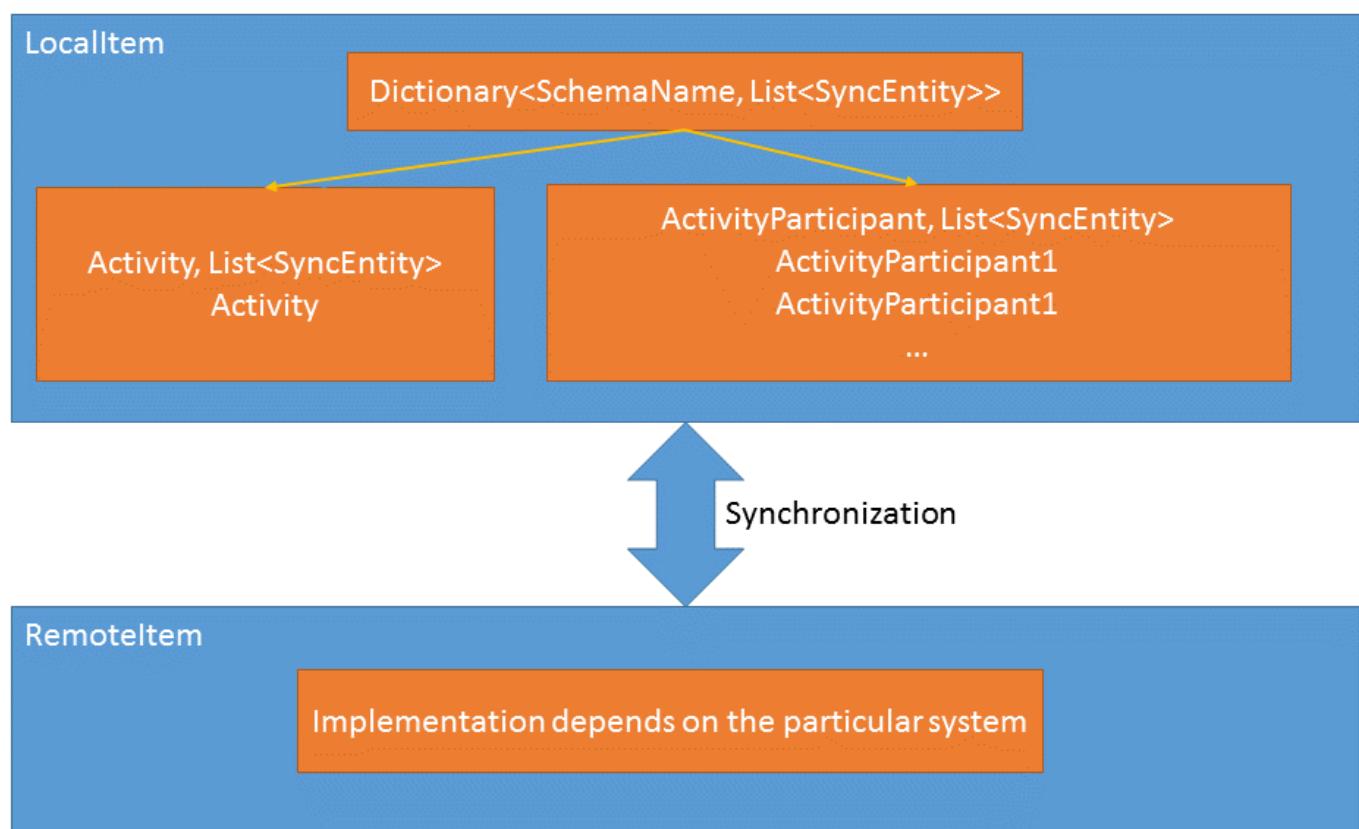
Only information about synchronized elements is stored in the metadata.

There can be multiple metadata table records for a single synchronization element - one for each application object included in a synchronization element.

Activity and participants — a single synchronization element. However, the metadata contains one record for each activity and one record for each participant.

Currently, only one object from external storage is transformed into several Creatio objects, as shown in Fig. 1.

Fig. 1. Schema of transformation of an external storage object into a local storage object.



The metadata system for a single synchronization element is represented as the *ItemMetadata* object class (*SysSyncMeta* element collection). Metadata management is carried out through the class that implements the *IReplicaMetadata* interface. An instance of the class that implements the *IReplicaMetadata* interface is created via the *MetaDataStore* factory class for a particular storage.

MetaDataStore class

Creates the specific class instance that implements the *IReplicaMetadata* interface for a storage. The class methods

are shown in Table 2.

Table 2. *MetaDataStore* class methods

Method	Returned value type	Description
<i>GetReplicaMetadata(Guid localStoreId, Guid remoteStoreId)</i>	<i>IreplicaMetadata</i>	Creates the class instance that implements the <i>IReplicaMetadata</i> interface for the specific storage.

ItemMetadata class

This class is an indivisible object of metadata synchronization. It contains a set of metadata for each synchronization element (*SysSyncMetadata* element collection). The class properties are shown in Table 3.

Table 3. *SysSyncMetadata* class properties.

Parameter	Type	Description
<i>RemoteId</i>	<i>String</i>	Element ID in external storage
<i>RemoteItemName</i>	<i>String</i>	Element name in external storage

IReplicaMetadata interface

This class implements the *IReplicaMetadata* interface, encapsulates the synchronization metadata and works with *ItemMetadata* objects. The interface properties are shown in table 4 and methods in table 5.

Table 4. The *IReplicaMetadata* interface properties

Parameter	Type	Description
<i>RemoteStoreId</i>	<i>Guid</i>	External storage ID.
<i>LocalStoreId</i>	<i>Guid</i>	Local storage ID.

Table 5. The *IReplicaMetadata* interface methods

Method	Returned value type	Description
<i>FindItemStore (string remoteItemId)</i>	<i>ItemMetadata</i>	Finds and returns the <i>ItemMetadata</i> synchronization metadata object by an ID in the <i>remoteItemId</i> external storage.
<i>UpdateItemMetadata (ItemMetadata oldItemMetaDatas, IRemoteItem remoteItem, LocalItem localItem, bool changesToBpm)</i>	<i>Void</i>	Updates metadata after synchronization.
<i>EnumerateItemsWithChangesInBpm (SyncContext context)</i>	<i>IEnumerable<ItemMetadata></i>	Returns a collection of <i>ItemMetadata</i> objects that have been modified since the last synchronization and not processed during the current synchronization session.
<i>CollectChangesInSyncedEntities (UserConnection userConnection, string schemaName, DateTime lastSyncVersion)</i>	<i>Void</i>	Updates metadata for synchronization elements modified in Creatio. If an element has been modified since the last SysMetadata synchronization, the <i>Version</i> column will be filled in with the date of element modification. The <i>ActualizeSysSyncMeta</i> procedure is used to update metadata.

<code>CollectNewDetailsForSyncedEntities Void (UserConnection userConnection, DetailEntityConfig detailEntityConfig, string remoteItemName)</code>	Creates new records in the <i>SysSyncMetaDataTable</i> for the synchronization element details.
<code>TryResolveRemoteId (Guid localId, Boolean out string remoteId)</code>	Returns the element ID in the external <i>remoteId</i> storage from metadata by a unique <i>localId</i> synchronization element in Creatio. If an element is marked as deleted, the <i>remoteId</i> will not be returned, and the method will return <i>false</i> .
<code>TryResolveExtraParameters (Guid localId, Boolean out string extraParameters)</code>	Returns additional <i>extraParameters</i> parameters for the synchronization element by <i>localId</i> . If <i>extraParameters</i> are not found, the method returns <i>false</i> .

Synchronizing tasks with MS Exchange

Beginner Easy Medium **Advanced**

General information

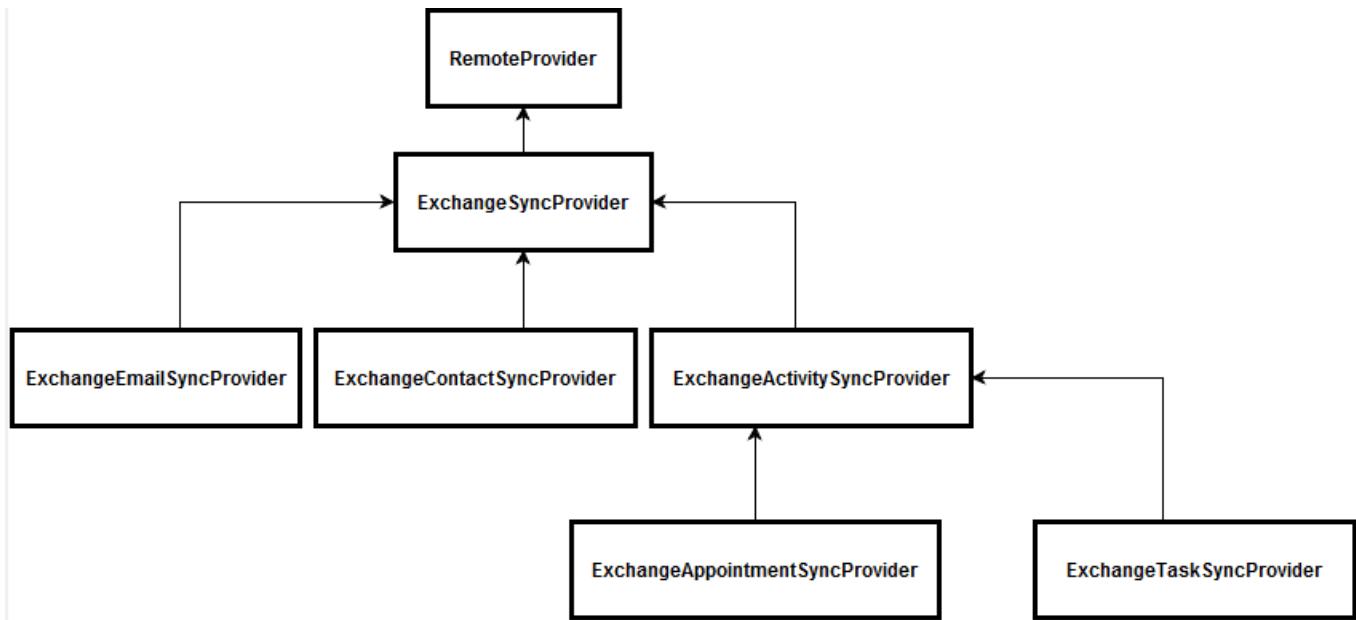
Integration with various entities of MS Exchange via EWS protocol (Exchange Web Services) is supported by the Sync Engine synchronization mechanism. This article describes synchronization of tasks between Creatio and MS Exchange. The task synchronization algorithm is no different from that described in the "[Creatio synchronization with external storages](#)" article. The process runs in three stages:

1. Retrieving changes from MS Exchange and applying them;
2. Retrieving changes from Creatio and applying them
3. Creating new tasks from Creatio in MS Exchange.

Integration classes

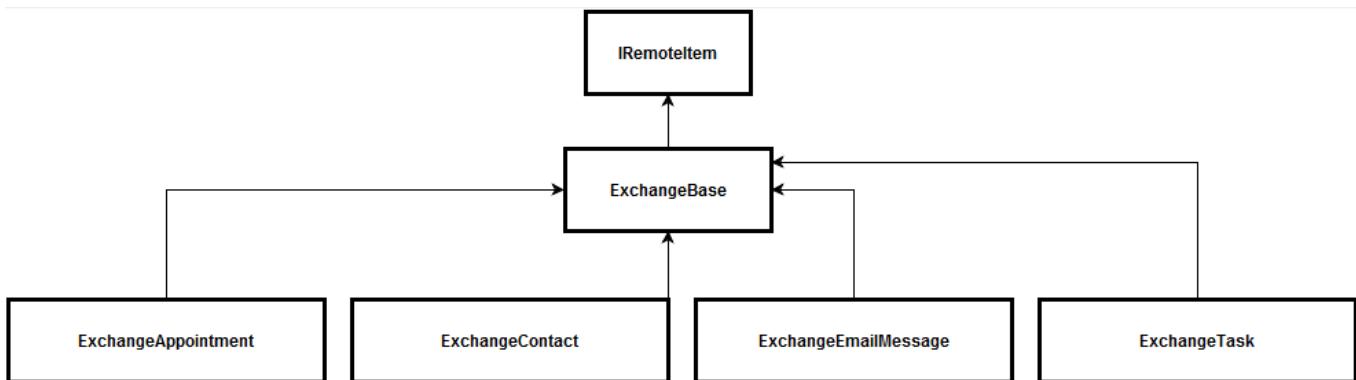
As described in the "[Creatio synchronization with external storages](#)" article, in order to implement an integration using this mechanism, you need a class that implements the logic of the external storage (an heir of the *RemoteProvider* class). The hierarchy of provider classes is shown in figure 1. You also need a class that implements the *IRemoteItem* interface, which represents a single instance of a synchronization item (in this case – the MS Exchange task). The *RemoteItem* hierarchy is shown in figure 2.

Fig. 1. RemoreProvider hierarchy schema



The `ExchangeTaskSyncProvider` class is the service provider for the MS Exchange external storage. This class implements the logic of selecting data and saving changes in Creatio and MS Exchange. The `ExchangeTask` class implements the `IRemoteItem` interface. The logic of filling in data in the corresponding systems is implemented in it.

Fig. 2. RemoteItem hierarchy schema



Synchronized data

The correspondence of Creatio objects to the `ExchangeTask` class fields is shown on table 1.

Table 1. The correspondence of Creatio objects to the `ExchangeTask` class fields

Creatio object	Object field	<code>ExchangeTask</code>
<code>Activity</code>	<code>Title</code>	<code>Subject</code>
	<code>StartDate</code>	<code>StartDate</code>
	<code>DueDate</code>	<code>CompleteDate</code> or <code>DueDate</code> depending on whether a task is finished or not.
	<code>Priority</code>	<code>Importance</code>
	<code>Status</code>	<code>Status</code>
	<code>RemindToOwner</code>	<code>IsRemindSet</code>
	<code>RemindToOwnerDate</code>	<code>ReminderDueBy</code>
	<code>Notes</code>	<code>Body.Text</code>

Logic of selecting data for synchronization

To select changes to the list of tasks selected for MS Exchange folder synchronization, use the following terms: select tasks for MS Exchange, which were modified after the last task synchronization or an MS Exchange task, which was not marked as synced. The MS Exchange task which were modified have corresponding activities in Creatio. The updated changes are applied in the corresponding system.

When you select modified Creatio activities, select the following:

- activities that are recorded in the metadata synchronization as MS Exchange tasks
- activities that have the current user as an author
- activities with a date of the last modification that does not correspond to the date of the last synchronization.

When selecting new Creatio activities, configure a set of common and custom filters. The main filter conditions are:

1. Activity type is not "email".
2. Activity does not have the [Display in calendar] checkbox selected.

A user can specify activity groups that will be exported from Creatio.

Extra

Filling in the [Start Date] and [Due Date] fields

The *ExchangeTask* object has several features for working with start date and due date:

- These fields are stored without time values. If you change a task in MS Exchange after synchronization, Creatio will apply the date from the MS Exchange task, and the time from the Creatio activity.
- The due date in *ExchangeTask* has two fields: *due date* and *complete date*.
- The start date and due date are optional in MS Exchange. If either of them is not filled in, the current date is set. Due to this, conflicts may arise, as both the start date and due date are required in Creatio, and the start date should be earlier than the due date.

Synchronizing email with MS Exchange

Beginner Easy Medium **Advanced**

General information

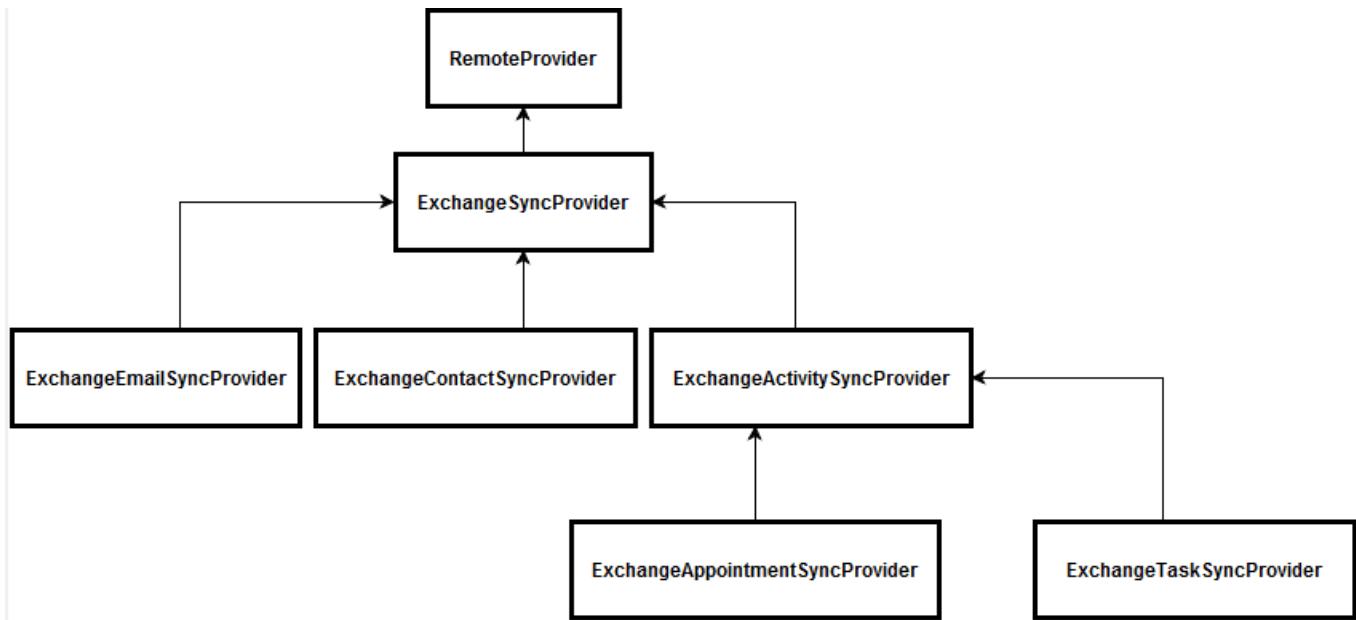
Synchronization with various services of MS Exchange via EWS protocol (Exchange Web Services) is supported by the Sync Engine synchronization mechanism. This article describes the synchronization of email in Creatio with MS Exchange. Email in Creatio is synchronized only from MS Exchange. Since emails can no longer be modified after they have been sent, only the emails that have not been synchronized previously are synchronized. The main difference between the email synchronization mechanism and integration is the email search engine in Creatio. Since the same email can be synchronized on behalf of any of the recipients or even via IMAP protocol, metadata synchronization can not be used for searching for previously synchronized emails. Use *subject*, *send date* and *message* to search for emails. All markups and spaces are removed from the message. To speed up the search, use the md5 hash that is stored in the *MailHash* column of the *Activity* object.

The second difference of this synchronization is that attachments are synchronized by a separate process, after all emails are processed. This is done in order to make the attachment download time not affect the email save time.

Integration classes

As described in the "**Creatio synchronization with external storages**" article, to implement integration using this mechanism, a class is required that implements the logic of working with external storage (*RemoteProvider* heir) and a class that implements the *IRemoteItem* interface, which is an instance of the synchronization element (in our case — email MS Exchange).

Fig. 1. RemoreProvider hierarchy schema



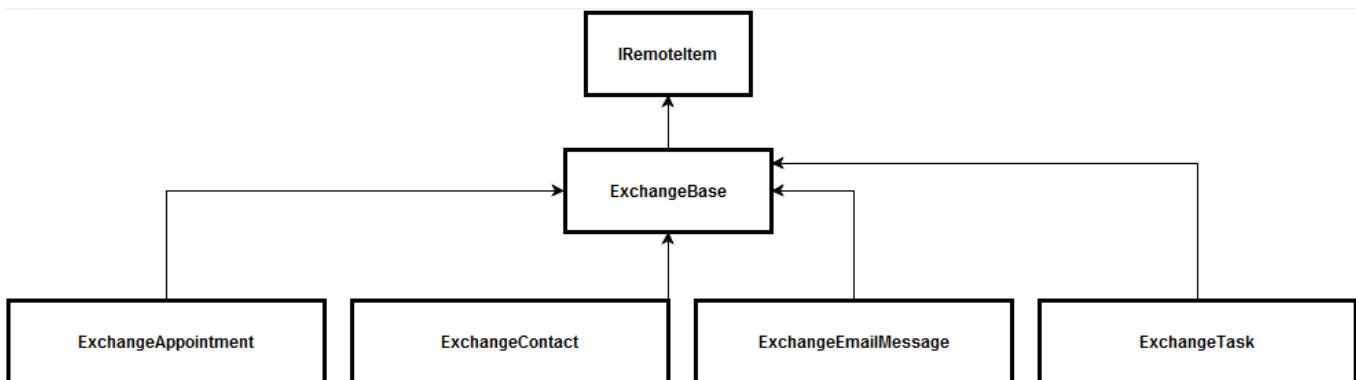
The *ExchangeEmailSyncProvider* class is the service provider for the exchange external storage. This class implements the logic of selecting data from MS Exchange.

The *ExchangeEmailMessage* class implements the *IRemoteItem* interface, in which the logic of filling in data in Creatio objects is implemented.

The *ExchangeUtility* class contains methods for the EWS API library and the methods used to download email attachments.

The *ExchangeEmailMessageUtility* class contains methods for converting the lookup values of the email fields.

Fig. 2. *RemoteItem* hierarchy schema



Synchronized data

The correspondence of Creatio objects to the *EmailMessage* class fields is shown on table 1.

Table 1. The correspondence of Creatio objects to the *EmailMessage* class fields

Creatio object	Object field	EmailMessage corresponding field
Activity	Title	Subject
	Body	Body.Text
	Sender	Sender
	Recipient	ToRecipients
	CopyRecipient	CcRecipients
	BlindCopyRecipient	BccRecipients

	<i>SendDate</i>	<i>DateTimeSpent</i>
	<i>Priority</i>	<i>Importance</i>
	<i>DueDate, StartDate</i>	<i>DateTimeReceived</i>
<i>ActivityFile</i>	<i>Name</i>	<i>Name</i>
	<i>Data</i>	<i>Content</i>
	<i>Size</i>	<i>Content.Length</i>

Logic of filling in email participants

For an email to be displayed correctly only for users who have synchronized it, the following mechanism of filling in the [Activity participants] detail has been implemented. Conventionally, this logic can be divided into two parts:

1. Adding participants to a new email.
2. Updating the list of participants when an email changes (including re-synchronization).

Adding participants to a new email

The main value that affects who becomes the participants of an email is the [Communication] contact detail. If a contact has an email address in the [Communication] detail, and this email is listed in one of the email address fields ([From], [To], [CC], [BCC]), the contact can be added to the participants. Additionally, a check is made whether there is a system user for this contact who is not a portal user. A user is added to the participants only after they have synchronized their email.

Updating the list of participants

For a user to become a participant after the synchronization of an existing email, the list of participants is updated - all participants who are not Creatio users are removed from the detail, and the algorithm of filling in detail for a new email runs. Thus, the users who have previously synchronized the email remain as participants, a new user is added, and the contact list is updated.

Logic of selecting data for synchronization

When selecting emails for synchronization with MS Exchange, use the following filter set: select emails that have been modified since the last synchronization and are not drafts. There is a limit for synchronization folders: "Deleted" and "Conflicting elements" folders do not participate in synchronization. When selecting emails the synchronization metadata is not taken into account. Always "save changes in Creatio". When processing each email, the system first checks for emails in Creatio. If an email already exists in Creatio, the participants are updated. If not, a new email is created. At the end of the synchronization, the system adds a task to synchronize attachments.

Synchronizing contacts with MS Exchange

Beginner Easy Medium **Advanced**

General information

Integration with various entities of MS Exchange via EWS protocol (Exchange Web Service) is supported by the Sync Engine synchronization mechanism. This article describes synchronization of contacts between Creatio and MS Exchange. The task synchronization algorithm is no different from that described in the article about Sync Engine synchronization. The process runs in three stages:

1. Retrieving changes from MS Exchange and applying them
2. Retrieving changes from Creatio and applying them
3. Creating new contacts from Creatio in MS Exchange.

Integration classes

As described in the "**Creatio synchronization with external storages**" article, to implement integration using this mechanism, a class is required that implements the logic of working with external storage (*RemoteProvider*

heir) and a class that implements the *IRemoteItem* interface, which is an instance of the synchronization element (in our case — MS Exchange contact).

Fig. 1. RemoreProvider hierarchy schema

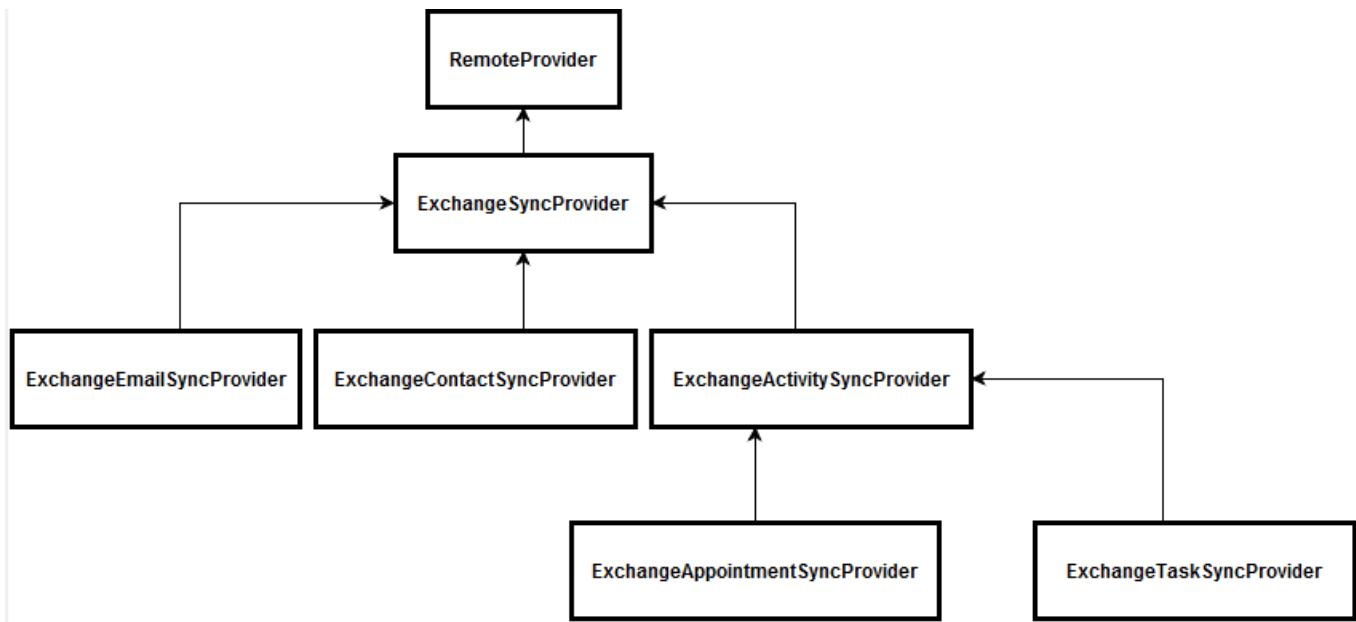
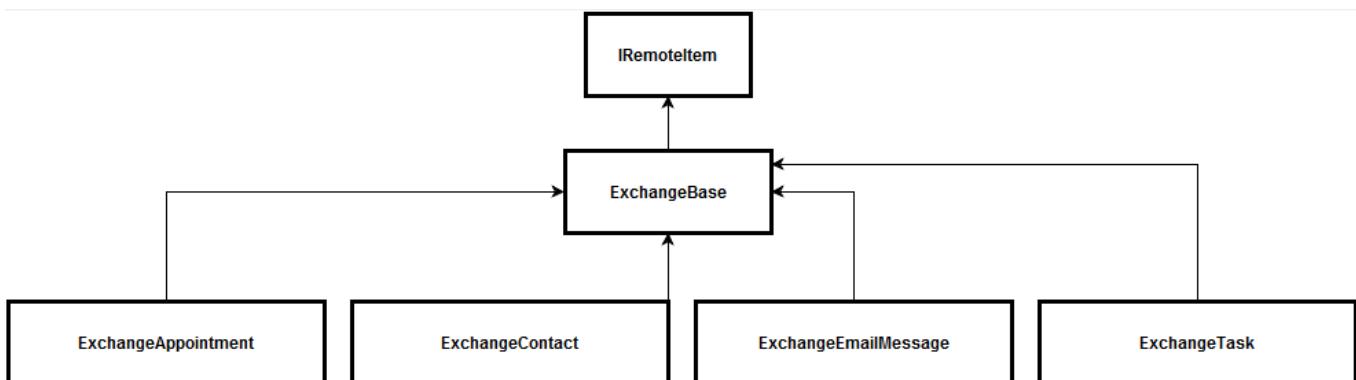


Fig. 2. RemoteItem hierarchy schema



The following classes are used for contact synchronization:

- The *ExchangeContactSyncProvider* class is the service provider for the MS Exchange external storage. This class implements the logic of selecting data and saving changes in Creatio and MS Exchange.
- The *ExchangeContact* class implements the *IRemoteItem* interface. The logic of filling in data in the corresponding systems is implemented in it.
- The *ExchangeAddressDetailsSynchronizer* class contains methods for converting contact addresses.
- The *ExchangeEmailAddressDetailsSynchronizer* class contains methods for converting contact email addresses.
- The *ExchangePhoneNumbersDetailsSynchronizer* class contains methods for converting contacts phones.

The logic of filling in details is located in separate classes, as there are significant differences in the formats of data storage in Creatio and MS Exchange. Additional conversion is required.

Synchronized data

The correspondence of Creatio objects to the Contact MS Exchange class fields is shown on table 1.

Table 1. The correspondence of Creatio objects to the Contact MS Exchange class fields

Creatio object	Object field	The Contact MS Exchange class field
----------------	--------------	-------------------------------------

<i>Contact</i>	<i>Name</i>	<i>DisplayName</i>
	<i>Surname</i>	<i>Surname</i>
	<i>GivenName</i>	<i>GivenName</i>
	<i>MiddleName</i>	<i>MiddleName</i>
	<i>Account</i>	<i>CompanyName</i>
	<i>JobTitle</i>	<i>JobTitle</i>
	<i>Department</i>	<i>Department</i>
	<i>BirthDate</i>	<i>Birthday</i>
	<i>SalutationType</i>	<i>TitleTag</i>
	<i>Gender</i>	<i>GenderTag</i>
<i>ContactCommunication</i>	<i>Number</i>	The <i>PhoneNumbers</i> collection values
	<i>CommunicationType</i>	The <i>PhoneNumbers</i> collection value key
<i>ContactAddresses</i>	<i>City</i>	The <i>PhysicalAddresses</i> collection element <i>City</i> field
	<i>Country</i>	The <i>PhysicalAddresses</i> collection element <i>CountryOfOrigin</i> field
	<i>Region</i>	The <i>PhysicalAddresses</i> collection element <i>State</i> field
	<i>Address</i>	The <i>PhysicalAddresses</i> collection element <i>Street</i> field
	<i>Zip</i>	The <i>PhysicalAddresses</i> collection element <i>PostalCode</i> field
	<i>AddressType</i>	The <i>PhysicalAddresses</i> collection value key

The correspondence of communication types is shown in Table 2.

Table 2. The correspondence of communication types of Creatio to MS Exchange

Creatio communication type	MS Exchange communication type
<i>Email</i>	<i>EmailAddress1, EmailAddress2, EmailAddress3</i>
<i>WorkPhone</i>	<i>BusinessPhone, BusinessPhone2</i>
<i>HomePhone</i>	<i>HomePhone</i>
<i>MobilePhone</i>	<i>MobilePhone</i>

The correspondence of addresses is shown in Table 3.

Table 3. The correspondence of addresses of Creatio to MS Exchange

Creatio address type	MS Exchange address type
<i>HomeAddress</i>	<i>Home</i>
<i>BusinessAddress</i>	<i>Business</i>

Logic of selecting data for synchronization

To select changes to the list of contacts selected for MS Exchange folder synchronization, use the following terms: select contacts for MS Exchange, which were modified after the last contact synchronization or an MS Exchange contact, which was not marked as synced. The contacts which were modified have corresponding contacts in Creatio. The updated changes are applied in the corresponding system.

When you select Creatio contacts for synchronization, select the following:

- contacts that have the current user as an author
- contacts with a date of last modification that does not correspond to the date of the last synchronization.
- contacts that were not used on the first step of synchronization

User settings also affect the rules for selecting new contacts in Creatio. Three settings are available:

1. Synchronize employees contacts. When you select this setting, the "Contact type" filter will be added to the request, and only the "Employee" type contacts will be synchronized.
2. Synchronize customers contacts. When you select this setting, the "Contact type" filter will be added to the request, and only the "Customer" type contacts will be synchronized.
3. Sync contacts from certain groups. When you select this setting, the selected contact group filters will be added to the request.

Additional features

Using the advanced contact keys in external storage

A situation may occur when there is a large number of MS Exchange contacts and of them will receive the same ID. As a result, synchronization may not correctly identify the appropriate contact in Creatio. To work around this situation, there are advanced external keys, which can be enabled by the [Use composite IDs for MS Exchange synchronized contacts] setting. Setting code - *UseComplexExchangeContactId*. After enabling it, you may need to resynchronize.

Synchronizing appointments with MS Exchange

Beginner Easy Medium **Advanced**

General information

Integration with various entities of Exchange via EWS protocol ([Exchange Web Services](#)) is supported by the Sync Engine synchronization mechanism. This article describes synchronization of appointments between Creatio and MS Exchange.

Creatio appointment synchronization is performed only for new activities or when the *Title*, *Location*, *StartDate*, *DueDate*, *Priority*, *Notes* fields are modified. The hash stored in the additional metadata parameters (in the *ExtraParameters* field) is formed according to these fields. If an appointment has been changed in Creatio, and the hash for *ExtraParameters* does not match the new hash, this appointment should be synchronized.

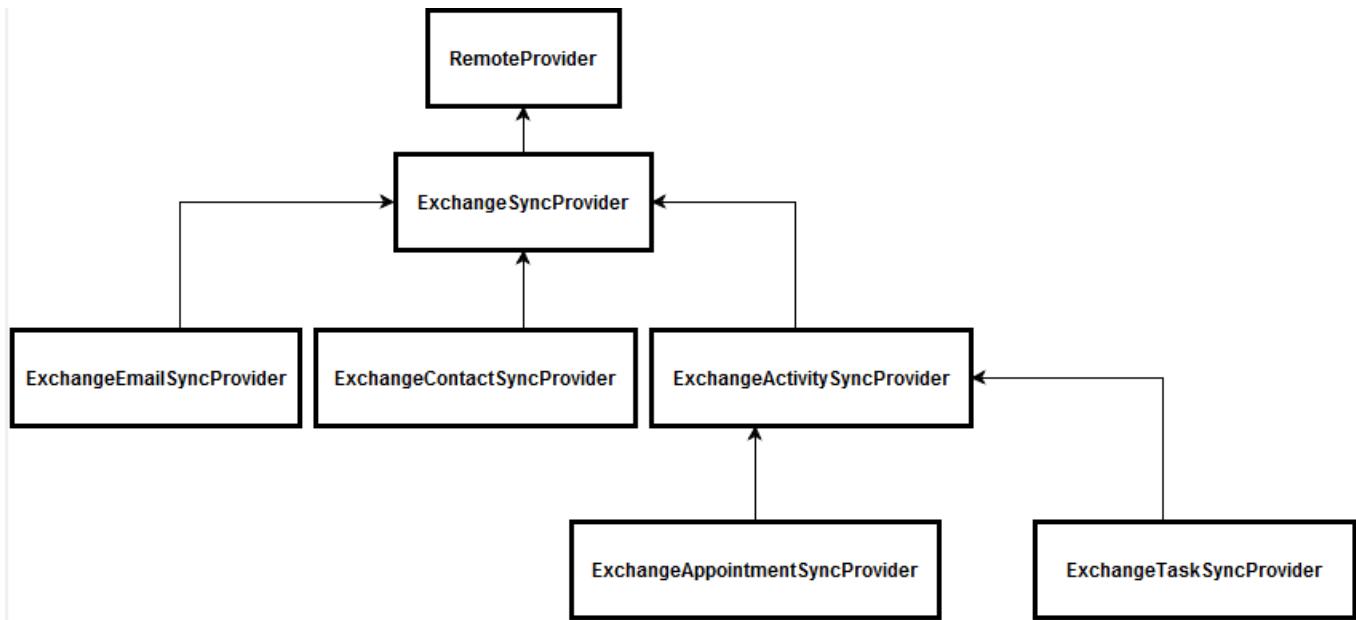
The appointment synchronization algorithm is no different from that described in the "**Creatio synchronization with external storages**" article. The process runs in three stages:

1. Retrieving changes from MS Exchange and applying them
2. Retrieving changes from Creatio and applying them
3. Creating new appointments from Creatio in MS Exchange.

Integration classes

As described in the "**Creatio synchronization with external storages**" article, to implement integration using this mechanism, a class is required that implements the logic of working with external storage (*RemoteProvider* heir) and a class that implements the *IRemoteItem* interface, which is an instance of the synchronization element (in our case — MS Exchange appointment).

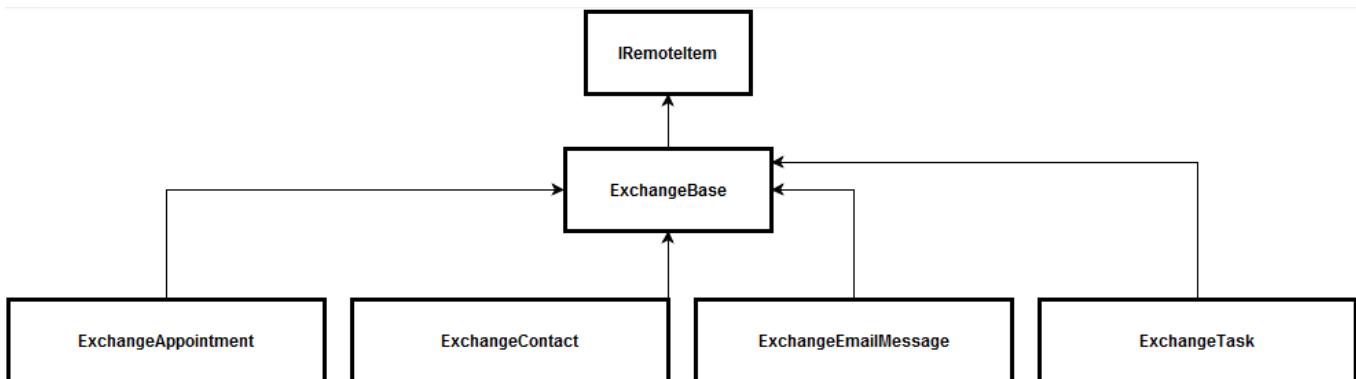
Fig. 1. RemoreProvider hierarchy schema



The *ExchangeAppointmentSyncProvider* is the provider used to work with the Exchange external storage. It contains the logic of selecting data and saving changes in Creatio and Exchange.

The *ExchangeAppointment* class implements the *IRemoteItem* interface, in which the logic of filling in data in Creatio objects is implemented.

Fig. 2. *RemoteItem* hierarchy schema



Synchronized data

The correspondence of Creatio objects to the *ExchangeAppointment* class fields is shown on table 1.

Table 1. The correspondence of Creatio objects to the *ExchangeAppointment* class fields

Creatio object	Object field	MS Exchange Appointment corresponding field
Activity	Title	<i>Subject</i>
	Location	<i>Location</i>
	StartDate	<i>StartDate</i>
	DueDate	<i>CompleteDate</i> or <i>DueDate</i> depending on whether an appointment is finished or not.
	Priority	<i>Importance</i>
	Status	Filled in as follows: If the status is not specified and the due date is later

		than the current date — Creatio sets the "New Appointment" status.
		If the due date is earlier than the current date, the status is set as a closed appointment with the "Information received" status.
	<i>RemindToOwner</i>	<i>IsReminderSet</i>
	<i>RemindToOwnerDate</i>	<i>ReminderDueBy</i>
	<i>Notes</i>	<i>Body.Text</i>
<i>ActivityParticipant</i>	<i>InviteResponse</i>	If the checkbox in MS Exchange is selected that identifies that an appointment was received and the user is its owner, and the "Appointment confirmed" checkbox is selected. Otherwise, if the checkbox is selected that identifies that the appointment was canceled.

Logic of selecting data for synchronization

To select changes to the list of appointments selected for MS Exchange folder synchronization, use the following terms: select appointments for MS Exchange, which were modified after the last contact synchronization or an MS Exchange appointment, which was not marked as synced. The appointments which were modified have corresponding contacts in Creatio. The updated changes are applied in the corresponding system.

When you select modified Creatio activities, select the following:

- activities which are recorded in the synchronization metadata as MS Exchange appointments via *RemoteId* (determined by a unique appointment ID in the *iCalId* calendar);
- activities with a date of the last modification that does not correspond to the date of the last synchronization.
- one appointment in Creatio corresponds to several appointments in MS Exchange for each participant.

When selecting new Creatio activities, configure a set of common and custom filters. The main filter conditions are:

1. Activity type is not "Email".
2. Activity has the [Display in calendar] checkbox selected.

A user can specify activity groups that will be exported from Creatio.

Logic of selecting appointment participants

When you synchronize an activity from MS Exchange to Creatio, only contacts that have email addresses from the list of appointment participants in MS Exchange are added to the [Participants] detail.

When you synchronize activities from Creatio to MS Exchange, the appointment participants for MS Exchange are filled in with primary contact email addresses.

Scheduler setup

Contents

- **Recommendations on scheduler setup**
- **Quartz policies for the processing of overdue tasks**

Recommendations on scheduler setup

Beginner

Easy

Medium

Advanced

Selecting the Quartz policies for the processing of overdue tasks

All Quartz policies for the processing of overdue tasks can be divided into three groups: *Ignore misfire policy*, *Run*

immediately and continue and *Discard and wait for next*. Recommendations about using each specific policy are given below.

Ignore misfire policy

This policy is represented by the *MisfireInstruction.IgnoreMisfirePolicy = -1* constant. It is recommended to use it when it is necessary to ensure that all trigger firings will be executed even with overdue tasks. For example, the task A with 2 minutes execution periodicity. Due to lack of Quartz threads or the scheduler shutdown, the next fire time of task A (NEXT_FIRE_TIME) lags 10 minutes from the current time. If execution of all 5 overdue tasks is required, use the *IgnoreMisfirePolicy* utility.

This policy is recommended to use for triggers that operate with a unique data at each fire and it is important to perform all trigger fires. For example, the task B that is executed once per 1 hour and generates the report for the time period from PREV_FIRE_TIME till PREV_FIRE_TIME + 1 hour. The scheduler was turned off for 8 hours. In this case, all 8 fires of the task B should be performed after launch of the scheduler and all reports should be generated.

Applying this policy to triggers that do not operate with unique data may cause to unnecessary clogging of the scheduler queue and application performance decrease. For example, the Exchange email synchronization is configured with the 1 minute interval for each Creatio user. An update was performed for 1.5 hours. After the update, the Quartz will synchronize user mailboxes 90 times before proceeding with tasks scheduled for the current time. Although it is enough to perform the delayed synchronization of mailboxes once, and then proceed the task according to the schedule.

Run immediately and continue

This group includes:

- *SimpleTrigger.FireNow;*
- *SimpleTrigger.RescheduleNowWithExistingRepeatCount;*
- *SimpleTrigger.RescheduleNowWithRemainingRepeatCount;*
- *CronTrigger.FireOnceNow;*
- *CalendarIntervalTrigger.FireOnceNow.*

More information about these policies can be found in the “**Quartz policies for the processing of overdue tasks**” article.

This policies should be used if the overdue task should be executed one time and as a priority and then execute scheduled tasks. For example, the email synchronization (`<user>@<server>_LoadExchangeEmailsProcess_<userId>`, `SyncImap_<user>@<server>_<userId>`) or the RemindingCountersJob and SyncWithLDAPProcess tasks.

For example, email synchronization is configured with the 5 minutes interval for all users and is fired by the `<user>@<server>_LoadExchangeEmailsProcess_<userId>`Trigger triggers. The update was performed since 1:30 AM till 2:43 AM. After update, the next fire time for the `<user>@<server>_LoadExchangeEmailsProcess_<userId>`Trigger triggers will be changed to 2:43 AM. All overdue tasks will be fired once at 2:43 AM and further will be fired according to the schedule (2:48 AM, 2:53 AM, 2:58 Am, etc.).

Discard and wait for next

This group includes:

- *SimpleTrigger.RescheduleNextWithRemainingCount;*
- *SimpleTrigger.RescheduleNextWithExistingCount;*
- *CronTrigger.DoNothing;*
- *CalendarIntervalTrigger.DoNothing.*

More information about these policies can be found in the “**Quartz policies for the processing of overdue tasks**” article.

These policies should be applied to the tasks that should be fired strictly at a specific time. For example the statistics collection launched daily at 3:00 AM when there is no active users on the website (the *CronTrigger* is used). This task is resource intensive and time consuming, and it can not be run during working hours, because this can slow down the work of users. In this case, use the *CronTrigger.DoNothing* policy. As a result, if the task was not fired, the

next fire will be at 3:00 AM of the next day.

Quartz configuration

thread count

If the scheduler delays the tasks or if some tasks have not been executed, increase the number of Quartz threads. For this, set necessary number of threads in the Web.config of the application loader:

```
<add key="quartz.threadPool.threadCount" value="5" />
```

The Web.config file of the application loader located in the root folder of the installed Creatio application.

misfireThreshold

If the increase in the number of Quartz threads is undesirable (for example, due to limited resources), the change in the misfireThreshold setting in the Web.config file of the application loader can optimize the task execution.

For example, an application with a number of tasks much bigger than a number of threads. The most of tasks are executed with a small interval (1 minute). The value of the *misfireThreshold* setting is 1 minute and the number of threads is 3:

```
<add key="quartz.jobStore.misfireThreshold" value="60000" />
<add key="quartz.threadPool.threadCount" value="3" />
```

For the most of the tasks used the policies from the *Run immediately and continue* group. This means following. For the most tasks that have the MISFIRE_INSTR not equal “-1” and the NEXT_FIRE_TIME less than the current time for 1 minute (60000 ms) the Quartz will periodically set the time of the next fire for the current time. This means that the initial order of the scheduled tasks will be lost because all tasks will have the current time as the fire time. The probability that Quartz will often process the same tasks and ignore the other tasks will increase.

The scheduler queue after 15 minutes of working is displayed on the Fig. 1. Tasks that have the PREV_FIRE_TIME = NULL never have been executed. There is a big number of these tasks.

Fig. 1. Scheduler queue with the misfireThreshold, = 1 minute

TRIGGER_NAME	TRIGGER_GROUP	NEXT_FIRE_TIME	PREV_FIRE_TIME	Last repeat interval, min	TRIGGER_STATE	TRIGGER_TYPE	START_TIME	MISFIRE_INSTR	...
MinutelyJob_15Trigger	TestGroup	2017-11-21 19:17:20.860	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.860	3	...
MinutelyJob_16Trigger	TestGroup	2017-11-21 19:17:20.863	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.863	3	...
MinutelyJob_17Trigger	TestGroup	2017-11-21 19:17:20.867	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.867	3	...
MinutelyJob_18Trigger	TestGroup	2017-11-21 19:17:20.867	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.867	3	...
MinutelyJob_19Trigger	TestGroup	2017-11-21 19:17:20.870	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.870	3	...
Syncimap_postulga.alexy@gmail.com_73b869f-34f3...	IMAP	2017-11-21 19:17:20.870	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.870	3	...
MinutelyJob_5Trigger	TestGroup	2017-11-21 19:17:20.873	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.873	3	...
Terrasoft.Configuration.CampaignFailoverHandlerTrigger	CampaignJobGroup	2017-11-21 19:17:20.873	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.873	3	...
Terrasoft.Configuration.TriggerEmailFailoverHandlerTr...	Mailing	2017-11-21 19:17:20.877	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.877	3	...
Terrasoft.Configuration.BulkEmailFailoverHandlerTrigger	Mailing	2017-11-21 19:17:20.880	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.880	3	...
MinutelyJob_6Trigger	TestGroup	2017-11-21 19:17:20.880	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.880	3	...
MinutelyJob_7Trigger	TestGroup	2017-11-21 19:17:20.883	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:17:20.883	3	...
MinutelyJob_0Trigger	TestGroup	2017-11-21 19:17:20.883	2017-11-21 19:04:37.643	12.72065943000	WAITING	SIMPLE	2017-11-21 19:17:20.883	3	...
MinutelyJob_1Trigger	TestGroup	2017-11-21 19:17:20.887	2017-11-21 19:04:37.663	12.72035926333	WAITING	SIMPLE	2017-11-21 19:17:20.887	3	...
MinutelyJob_2Trigger	TestGroup	2017-11-21 19:17:20.887	2017-11-21 19:04:37.667	12.72034262833	WAITING	SIMPLE	2017-11-21 19:17:20.887	3	...
CESWebHooksSyncTrigger	Mailing	2017-11-21 19:17:20.890	2017-11-21 19:08:20.733	9.00257776333	WAITING	CAL_INT	2017-11-21 18:39:38.140	1	...
MandrillScheduledMailingTrigger	Mailing	2017-11-21 19:17:20.890	2017-11-21 19:08:20.740	9.00252771833	WAITING	CAL_INT	2017-11-21 18:39:38.197	1	...
MT_2874667341633338083	DEFAULT	2017-11-21 19:18:20.857	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:08:42.410	0	...
Terrasoft.Configuration.ML.MLModelTrainerJob_Terra...	DEFAULT	2017-11-21 19:18:20.860	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:08:42.433	0	...
Terrasoft.Configuration.EmailMining.EmailMiningJob_T...	DEFAULT	2017-11-21 19:18:20.860	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:08:42.450	0	...
Terrasoft.Configuration.NotificationsJobTrigger	NotificationsCountersGroup	2017-11-21 19:18:20.863	2017-11-21 19:08:20.727	10.00026696166	WAITING	SIMPLE	2017-11-21 19:18:20.863	3	...
A.Postulga@terrasoft.ru_LoadExchangeEmailsProces...	Exchange	2017-11-21 19:18:20.863	2017-11-21 19:08:20.743	10.000205025333	WAITING	SIMPLE	2017-11-21 19:18:20.863	3	...
MinutelyJob_8Trigger	TestGroup	2017-11-21 19:18:20.867	2017-11-21 19:08:20.743	10.000205029000	WAITING	SIMPLE	2017-11-21 19:18:20.867	3	...
MinutelyJob_9Trigger	TestGroup	2017-11-21 19:18:20.870	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:18:20.870	3	...
MinutelyJob_10Trigger	TestGroup	2017-11-21 19:18:20.870	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:18:20.870	3	...
MinutelyJob_11Trigger	TestGroup	2017-11-21 19:18:20.873	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:18:20.873	3	...
MinutelyJob_12Trigger	TestGroup	2017-11-21 19:18:20.873	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:18:20.873	3	...
c995bb9e-070e-411f-8d19-181e6393b6e8	DEFAULT	2017-11-21 19:18:25.517	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:08:25.517	0	...
RemindingCountersJob_73b869f-34f3-4f20-ab4d-748...	RemindingCountersGroup	2017-11-21 19:18:49.613	2017-11-21 19:17:48.613	1.00000000000	BLOCKED	CAL_INT	2017-11-21 19:08:46.813	0	...
MinutelyJob_3Trigger	TestGroup	2017-11-21 19:19:20.723	2017-11-21 19:18:20.723	1.00000000000	BLOCKED	SIMPLE	2017-11-21 19:08:20.723	3	...
MinutelyJob_4Trigger	TestGroup	2017-11-21 19:19:20.723	2017-11-21 19:18:20.723	1.00000000000	BLOCKED	SIMPLE	2017-11-21 19:08:20.723	3	...
Terrasoft.Configuration.DelayedNotifyingTrigger	DelayedNotificationGroup	2017-11-21 19:23:20.697	2017-11-21 19:18:20.697	5.00000000000	WAITING	SIMPLE	2017-11-21 19:08:20.697	3	...
SyncWithLDAPProcessTrigger	LDAP	2017-11-21 19:43:35.327	NULL	NULL	WAITING	CAL_INT	2017-11-21 18:43:35.327	0	...
NotificationCleanerTrigger	NotificationCleanerGroup	2017-11-22 02:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 18:39:37.000	0	...
ActualizeActiveContactsTrigger	Mailing	2017-11-22 02:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 18:39:37.997	1	...
GenerateAnniversaryRemindingsTrigger	GenerateAnniversaryRem...	2017-11-22 03:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 19:08:25.000	0	...
Terrasoft.Configuration.DelayedNotificationCleaningTr...	DelayedNotificationGroup	2017-11-22 18:39:37.843	NULL	NULL	WAITING	SIMPLE	2017-11-21 18:39:37.843	3	...

Increase the *misfireThreshold* value to 10 minutes (and clear the PREV_FIRE_TIME in QRTZ_TRIGGERs):

```
<add key="quartz.jobStore.misfireThreshold" value="600000" />
```

The scheduler queue after 15 minutes of working is displayed on the Fig. 2.

Fig. 2. Scheduler queue with the misfireThreshold, = 10 minutes

TRIGGER_NAME	TRIGGER_GROUP	NEXT_FIRE_TIME	PREV_FIRE_TIME	Last repeat interval, min	TRIGGER_STATE	TRIGGER_TYPE	START_TIME	MISFIRE_INSTR
MinutelyJob_13Trigger	TestGroup	2017-11-21 19:23:20.960	2017-11-21 19:22:20.960	1.00000000000	WAITING	SIMPLE	2017-11-21 19:20:20.960	3
Terasoft.Configuration.TriggerEmailFailoverHandlerTri...	Mailing	2017-11-21 19:25:20.980	2017-11-21 19:20:20.980	5.00000000000	WAITING	SIMPLE	2017-11-21 19:20:20.980	3
Terasoft.Configuration.BulkEmailFailoverHandlerTrigger	Mailing	2017-11-21 19:25:20.980	2017-11-21 19:20:20.980	5.00000000000	WAITING	SIMPLE	2017-11-21 19:20:20.980	3
RemindingCountersJob_73b869f-34f3-4f20-ab4d-748...	RemindingCountersGroup	2017-11-21 19:25:50.787	2017-11-21 19:24:50.787	1.00000000000	BLOCKED	CAL_INT	2017-11-21 19:22:50.787	0
MinutelyJob_3Trigger	TestGroup	2017-11-21 19:26:20.723	2017-11-21 19:25:20.723	1.00000000000	BLOCKED	SIMPLE	2017-11-21 19:08:20.723	3
MinutelyJob_4Trigger	TestGroup	2017-11-21 19:26:20.723	2017-11-21 19:25:20.723	1.00000000000	BLOCKED	SIMPLE	2017-11-21 19:08:20.723	3
Terasoft.Configuration.DelayedNotifyingTrigger	DelayedNotificationGroup	2017-11-21 19:28:20.697	2017-11-21 19:23:20.697	5.00000000000	WAITING	SIMPLE	2017-11-21 19:08:20.697	3
SyncImap_postulga.alexey@gmail.com_73b869f-34f3...	IMAP	2017-11-21 19:32:39.110	2017-11-21 19:20:20.973	12.30228216333	WAITING	SIMPLE	2017-11-21 19:32:39.110	3
MinutelyJob_5Trigger	TestGroup	2017-11-21 19:32:39.117	2017-11-21 19:20:20.977	12.30234882000	WAITING	SIMPLE	2017-11-21 19:32:39.117	3
MinutelyJob_6Trigger	TestGroup	2017-11-21 19:32:39.117	2017-11-21 19:20:20.983	12.30224896000	WAITING	SIMPLE	2017-11-21 19:32:39.117	3
MinutelyJob_7Trigger	TestGroup	2017-11-21 19:32:39.120	2017-11-21 19:20:20.983	12.30226555000	WAITING	SIMPLE	2017-11-21 19:32:39.120	3
MinutelyJob_0Trigger	TestGroup	2017-11-21 19:32:39.120	2017-11-21 19:20:20.987	12.30224884166	WAITING	SIMPLE	2017-11-21 19:32:39.120	3
MinutelyJob_1Trigger	TestGroup	2017-11-21 19:32:39.123	2017-11-21 19:20:20.987	12.30224826833	WAITING	SIMPLE	2017-11-21 19:32:39.123	3
MinutelyJob_2Trigger	TestGroup	2017-11-21 19:32:39.123	2017-11-21 19:20:20.990	12.30224890666	WAITING	SIMPLE	2017-11-21 19:32:39.123	3
Terasoft.Configuration.NotificationsJobTrigger	NotificationsCountersGroup	2017-11-21 19:32:39.127	2017-11-21 19:20:21.057	12.30116543500	WAITING	SIMPLE	2017-11-21 19:32:39.127	3
A.Postulga@terasoft.ru_LoadExchangeEmailsProces...	Exchange	2017-11-21 19:32:39.130	2017-11-21 19:20:21.060	12.30116547833	WAITING	SIMPLE	2017-11-21 19:32:39.130	3
MinutelyJob_8Trigger	TestGroup	2017-11-21 19:32:39.130	2017-11-21 19:20:21.060	12.30116555333	WAITING	SIMPLE	2017-11-21 19:32:39.130	3
MinutelyJob_9Trigger	TestGroup	2017-11-21 19:32:39.133	2017-11-21 19:20:21.063	12.30116544666	WAITING	SIMPLE	2017-11-21 19:32:39.133	3
MinutelyJob_10Trigger	TestGroup	2017-11-21 19:32:39.133	2017-11-21 19:20:21.067	12.30116546166	WAITING	SIMPLE	2017-11-21 19:32:39.133	3
MinutelyJob_11Trigger	TestGroup	2017-11-21 19:32:39.137	2017-11-21 19:20:21.067	12.30116550166	WAITING	SIMPLE	2017-11-21 19:32:39.137	3
MinutelyJob_12Trigger	TestGroup	2017-11-21 19:32:39.140	2017-11-21 19:20:21.070	12.301168126666	WAITING	SIMPLE	2017-11-21 19:32:39.140	3
CESWebHooksSyncTrigger	Mailing	2017-11-21 19:32:39.143	2017-11-21 19:20:38.140	12.01673093833	WAITING	CAL_INT	2017-11-21 18:39:38.140	1
MandrilScheduledMailingTrigger	Mailing	2017-11-21 19:32:39.143	2017-11-21 19:20:38.197	12.01579751166	WAITING	CAL_INT	2017-11-21 18:39:38.197	1
MinutelyJob_14Trigger	TestGroup	2017-11-21 19:32:39.147	2017-11-21 19:20:963	11.30308236666	WAITING	SIMPLE	2017-11-21 19:32:39.147	3
MinutelyJob_15Trigger	TestGroup	2017-11-21 19:32:39.150	2017-11-21 19:21:20.963	11.30308227500	WAITING	SIMPLE	2017-11-21 19:32:39.150	3
MinutelyJob_16Trigger	TestGroup	2017-11-21 19:32:39.150	2017-11-21 19:21:20.967	11.30308224833	WAITING	SIMPLE	2017-11-21 19:32:39.150	3
MinutelyJob_17Trigger	TestGroup	2017-11-21 19:32:39.153	2017-11-21 19:20:20.967	11.30308234666	WAITING	SIMPLE	2017-11-21 19:32:39.153	3
MinutelyJob_18Trigger	TestGroup	2017-11-21 19:32:39.210	2017-11-21 19:21:20.970	11.30399938833	WAITING	SIMPLE	2017-11-21 19:32:39.210	3
MinutelyJob_19Trigger	TestGroup	2017-11-21 19:32:39.213	2017-11-21 19:21:20.970	11.30401564500	WAITING	SIMPLE	2017-11-21 19:32:39.213	3
MT_520821574165140644	DEFAULT	2017-11-21 19:32:39.217	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:22:30.010	0
Terasoft.Configuration.ML.MLModelTrainerJob, Terra...	DEFAULT	2017-11-21 19:32:39.227	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:22:30.037	0
Terasoft.Configuration.EmailMining.EmailMiningJob, T...	DEFAULT	2017-11-21 19:32:39.227	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:22:30.047	0
Terasoft.Configuration.CampaignFailoverHandlerTrigger	CampaignJobGroup	2017-11-21 19:35:20.977	2017-11-21 19:20:20.977	15.00000000000	WAITING	SIMPLE	2017-11-21 19:20:20.977	3
0778a81c-e91-402a-826a-77ab0e3afb25	DEFAULT	2017-11-21 19:37:25.223	NULL	NULL	WAITING	SIMPLE	2017-11-21 19:22:25.223	0
SyncWithLDAPProcessTrigger	LDAP	2017-11-21 19:43:35.327	NULL	NULL	WAITING	CAL_INT	2017-11-21 18:43:35.327	0
NotificationCleanerTrigger	NotificationCleanerGroup	2017-11-22 02:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 18:39:37.000	0
ActualizeActiveContactsTrigger	Mailing	2017-11-22 02:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 18:39:37.997	1
GenerateAnniversaryRemindersTrigger	GenerateAnniversaryRem...	2017-11-22 03:00:00.000	NULL	NULL	WAITING	CRON	2017-11-21 19:22:25.000	0
Terasoft.Configuration.DelayedNotificationCleaningTri...	DelayedNotificationGroup	2017-11-22 18:39:37.843	NULL	NULL	WAITING	SIMPLE	2017-11-21 18:39:37.843	3

The number of non-executed tasks is decreased.

Increasing of the value of the *misfireThreshold* will lead to more equal tasks execution. Almost all jobs from the queue will be executed. Due to lack of threads, the scheduler does not have time to fire each of the tasks in a minute. This is displayed in the [Last repeat interval] column that has the value NEXT_FIRE_TIME - PREV_FIRE_TIME, min. However, the scheduler fires each of the tasks.

batchTriggerAcquisitionMaxCount

Increase the *batchTriggerAcquisitionMaxCount* to optimize the scheduler performance if you do not use the clustered Quartz configuration (one scheduler node is used).

Quartz policies for the processing of overdue tasks

Beginner Easy Medium **Advanced**

Introduction

The Quartz has policies common to all types of triggers, and policies that are specific to a specific type of trigger. All policies used for the *SimpleTrigger*, *CronTrigger* or *CalendarIntervalTrigger* triggers are listed in the Table 1.

Table 1. Trigger policies

Quartz policy	The MISFIRE_INSTR value	The Terrasoft.Core.Scheduler.AppSchedulerMisfireInstruction value	Trigger type
<i>IgnoreMisfirePolicy</i>	-1	IgnoreMisfirePolicy	for all types
<i>IgnoreMisfirePolicy</i> behavior description			

Triggers with the *IgnoreMisfirePolicy* always will be fired in time. For such triggers, the Quartz will not update the next fire time (NEXT_FIRE_TIME).

The Quartz will fire all overdue tasks as priority, and then return to the initial trigger schedule. For example, the task with the Simple Trigger was planned for 10 iterations. Initial conditions:

- START_TIME = 9:00
- REPEAT_COUNT = 9 (first fire + 9 iterations)
- REPEAT_INTERVAL = 0:15.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will try to fire 2 overdue tasks as priority (in 9:00 and 9:15). After that the Quartz fires the 8 remaining tasks according to the schedule (at 9:30, 9:45, etc.).

SmartPolicy

0 SmartPolicy

for all types

SmartPolicy behavior description

By default is used by Quartz for all types of triggers. According to the type and configuration of the trigger, the Quartz will select corresponding policy. The selection algorithm for the Quartz version 2.3.2 is shown in the pseudo-code below.

```
if (TRIGGER_TYPE == 'SIMPLE') // Simple trigger.
    if (REPEAT_COUNT == 0) // Without repeats.
        MISFIRE_INSTR = 1 // SimpleTrigger.FireNow
    else if (REPEAT_COUNT == -1) // Continuous repeating.
        MISFIRE_INSTR = 4 // SimpleTrigger.RescheduleNextWithRemainingCount
    else // The number of repetitions is indicated.
        MISFIRE_INSTR = 2 // SimpleTrigger.RescheduleNowWithExistingRepeatCount
else if (TRIGGER_TYPE == 'CAL_INT') // CalendarInterval trigger.
    MISFIRE_INSTR = 1 // CalendarIntervalTrigger.FireOnceNow
else if (TRIGGER_TYPE == 'CRON') // Cron trigger.
    MISFIRE_INSTR = 1 // CronTrigger.FireOnceNow
```

SimpleTrigger.FireNow

1 FireNow

SimpleTrigger

SimpleTrigger.FireNow behavior description

Applied for the *SimpleTrigger* triggers that have the REPEAT_COUNT=0 (triggers fired for one time). If the REPEAT_COUNT value is not "0", then the *SimpleTrigger.RescheduleNowWithRemainingRepeatCount* policy will be applied.

For example, the task planned with the *SimpleTrigger*. Initial conditions:

- START_TIME = 9:00;
- REPEAT_COUNT = 0.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will try to fire the task as priority. As a result, the task will be fired at 9:20 or later.

SimpleTrigger.RescheduleNowWithExistingRepeatCount 2 RescheduleNowWithExistingRepeatCount

SimpleTrigger

SimpleTrigger.RescheduleNowWithExistingRepeatCount behavior description

Scheduler will try to fire the first overdue task as a priority, All other triggers will be fired with the REPEAT_INTERVAL interval.

For example, the task with the Simple Trigger was planned for 10 iterations. Initial conditions:

- START_TIME = 9:00
- REPEAT_COUNT = 9
- REPEAT_INTERVAL = 0:15.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will try to fire first overdue task scheduled for 9:00 (from tasks scheduled on 9:00 and 9:15) at 9:20. The remained 9 tasks will be fired at 9:35, 9:50, etc.

For the *AppScheduler.ScheduleMinutelyJob* methods the behavior of the *RescheduleNowWithExistingRepeatCount* is similar to the *RescheduleNowWithRemainingRepeatCount*, and the behavior of the *RescheduleNextWithRemainingCount* is similar to the *RescheduleNextWithExistingCount*, because the triggers with the REPEAT_COUNT = -1 are used.

SimpleTrigger.RescheduleNowWithRemainingRepeatCount 3 RescheduleNowWithRemainingRepeatCount

SimpleTrigger

SimpleTrigger.RescheduleNowWithRemainingRepeatCount behavior description

Scheduler will try to fire the first overdue task as a priority, Other overdue tasks are ignored. The scheduler fires remained tasks that are not overdue with the REPEAT_INTERVAL interval.

For example, the task with the Simple Trigger was planned for 10 iterations. Initial conditions:

- START_TIME = 9:00
- REPEAT_COUNT = 9
- REPEAT_INTERVAL = 0:15.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will try to fire first overdue task (from tasks scheduled on 9:00 and 9:15) at 9:20. The second overdue task will be ignored and other 8 tasks will be fired at 9:35, 9:50, etc.

SimpleTrigger.RescheduleNextWithRemainingCount 4 RescheduleNextWithRemainingCount

SimpleTrigger

SimpleTrigger.RescheduleNextWithRemainingCount behavior description

The scheduler ignores overdue tasks and waits for the next planned fire of the task. At the next fire time, the remaining non-overdue tasks will be executed with the REPEAT_INTERVAL interval.

For example, the task with the Simple Trigger was planned for 10 iterations. Initial conditions:

- START_TIME = 9:00
- REPEAT_COUNT = 9
- REPEAT_INTERVAL = 0:15.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will fire the rest of 8 non-overdue tasks at 9:30, 9:45, etc.

SimpleTrigger.RescheduleNextWithExistingCount 5 RescheduleNextWithExistingCount

SimpleTrigger

SimpleTrigger.RescheduleNextWithExistingCount behavior description

The scheduler will wait for the next launch time and will fire all remained tasks with the REPEAT_INTERVAL interval.

For example, the task with the Simple Trigger was planned for 10 iterations. Initial conditions:

- START_TIME = 9:00
- REPEAT_COUNT = 9
- REPEAT_INTERVAL = 0:15.

If the scheduler was disabled from 8:50 till 9:20, then after launch the Quartz will fire all 10 tasks at 9:30, 9:45, etc.

CronTrigger.FireOnceNow

1 -

CronTrigger

CronTrigger.FireOnceNow behavior description

Scheduler will try to fire the first overdue task as a priority. Other overdue tasks are ignored. Remained non-overdue tasks are fired by the scheduler according to the schedule.

For example, the task planned with the CronTrigger: CRON_EXPRESSION = '0 0 9-17 ? * MON-FRI' (from Monday to Friday 9:00 AM – 17:00 PM). If the scheduler was disabled from 8:50 till 10:20, then after launch the Quartz will try to fire first overdue task from two (at 9:00 and 10:00). After that, tasks will be fired at 11:00, 12:00, etc.

CronTrigger.DoNothing

2 -

CronTrigger

CronTrigger.DoNothing behavior description

Scheduler ignores all overdue tasks. Remained non-overdue tasks will be fired according to the schedule.

For example, the task planned with the CronTrigger: CRON_EXPRESSION = '0 0 9-17 ? * MON-FRI' (from Monday to Friday 9:00 AM – 17:00 PM). If the scheduler was disabled from 8:50 till 10:20, then after launch the Quartz will start to fire tasks since 11:00 (at 11:00, 12:00, etc).

CalendarIntervalTrigger.FireOnceNow

1 -

CalendarIntervalTrigger

CalendarIntervalTrigger behavior description

The behavior is similar to the *CronTrigger.FireOnceNow*.

CalendarIntervalTrigger.DoNothing 2
CalendarIntervalTrigger.DoNothing behavior description
The behavior is similar to the *CronTrigger.DoNothing*.

CalendarIntervalTrigger

Self-service Portal

Contents

- **Introduction**
- **PortalMessagePublisherExtensions mixin. Portal messages in SectionActionsDashboard**
- **Restricting access to web services for portal users**

Self-service Portal

Beginner

Easy

Medium

Advanced

Introduction

The Self-service Portal (SSP) is an integral part of the [Service Creatio, enterprise edition](#) and [Creatio customer center](#) products, as well as all bundles that these products are part of.

The SSP is a workplace where portal users are automatically redirected after login.

Portal users have access to the [Cases] and [Knowledge base] sections and to the [Self-service portal main page], which contains general summary information and works as a single workplace for a portal user.

The [Cases] section is used for registration of cases by the customers, viewing the status of their cases, entering additional case information and for obtaining information about case resolution process. By default, a portal user has access to those cases where this user is specified as a contact. The user can enter additional information, publish messages and interact with the service staff on the case page. The case history is displayed on the case page at the [Processing] tab.

The portal's [Knowledge base] section is used for searching for reference information or a solution. This section can be filled only by the helpdesk staff from the main interface of the system.

Portal interface

From the development point of view, the portal is a preconfigured separate workplace. By default, this workplace is not available for the ordinary portal users. A system user with the “portal user” type automatically enters this workplace (the portal main page) after authorization.

Portal sections and pages are the same as **sections** and **system edit pages** from the main system interface and they work with the same *Entities*. The case edit page on the portal is simpler compared to the regular case page and does not contain the majority of fields. These edit pages are different objects in configuration (*CasePage* and *PortalCasePage*).

The system of portal access permissions slightly differs. To grant access to the specific entities (EntitySchema) for the portal users, you need to specify these entities in the [List of objects available for portal users] lookup. Self-service portal licenses limit the number of records that can be added to this lookup. By default, the number is limited to 70 records.

Working with the page wizard on the portal

The portal user cannot access the functions of the page-, detail- and section wizards. These functions can be accessed from the main system interface with administrator permissions in the following way:

1. Enter the [Workplace setup] section in the system designer.
2. Select the [Portal] workplace and click the [Open] button.
3. Select the required section and click the [Section wizard] button.

The standard section wizard will open.

Configuring the portal and portal users

To start using the portal:

1. Ensure that the `/configuration/terrasoft/auth` option in the `web.config` file of the application loader (the “external” `web.config`) has the following in it:

```
<terrasoft>
    <auth providerNames="InternalUserPassword, SSPUserPassword" ...>
    ...
</terrasoft>
```

This setting is responsible for the authorization in the portal users in the system.

2. Create a contact for the user.
3. Create a user with the “Portal user” type. Fill out the required fields.
4. Provide all necessary licenses for the user.

A portal user is required to have a valid time zone specified in the profile. The time zone is not specified for new users by default. Portal users must edit their profiles and select the proper time zone. The system will display all dates and times in the portal user’s local time.

PortalMessagePublisherExtensions mixin. Portal messages in SectionActionsDashboard

Beginner

Easy

Medium

Advanced

Introduction

A mixin is a class designed to extend the functions of other classes. Mixins are separately created classes with additional functionality. Learn more about mixins in the “[Mixins. The “mixins” property](#)” article.

The `PortalMessagePublisherExtensions` mixin is used for the extension of the `SectionActionsDashboard` schema (and its derived schemas). It allows you to extend the configuration of the `SectionActionsDashboard` tabs with the `PortalMessageTab` portal message tab and add the corresponding `Portal` message portal. The mixin is implemented in the `PortalMessagePublisher` package and is available in the `ServiceEnterprise` product (or in the bundles that include this product).

Methods

Name	Description
<code>extendTabsConfig(config) : Object</code>	Extends the configuration of the <code>SectionActionsDashboard</code> tabs with the <code>PortalMessageTab</code> portal messages tab. Returns the augmented object (<code>Object</code>) of the <code>SectionActionDashboard</code> tab configuration. The <code>config</code> parameter (<code>Object</code>) – <code>SectionActionsDashboard</code> tab configuration object.
<code>extendSectionPublishers(publishers) : Array</code>	Adds a portal channel (<code>Portal</code>) to the message publisher collection. Returns the augmented collection of message publishers (<code>Array</code>). The <code>publishers</code> (<code>Array</code>) parameter is the collection of message publishers.

Use case

```
define("CaseSectionActionsDashboard", ["PortalMessagePublisherExtensions"],
function() {
    return {
```

```
mixins: {
    /**
     * @class PortalMessagePublisherExtensions extends tabs and publishers
  configs.
    */
  PortalMessagePublisherExtensions:
  "Terrasoft.PortalMessagePublisherExtensions"
},
methods: {
    /**
     * @inheritDoc Terrasoft.SectionActionsDashboard#getExtendedConfig
     * @override
     */
  getExtendedConfig: function() {
    // Getting the tab configuration object from the parent method.
    var config = this.callParent(arguments);
    // Calling the mixin method, adding a portal tab configuration.

  this.mixins.PortalMessagePublisherExtensions.extendTabsConfig.call(this, config)
    // Returns the extended configuration object.
    return config;
  },

  /**
   * @inheritDoc Terrasoft.SectionActionsDashboard#getSectionPublishers
   * @override
   */
  getSectionPublishers: function() {
    // Getting a collection of message publishers from the parent method.
    var publishers = this.callParent(arguments);
    // Calling the mixin method, adding a portal channel.

  this.mixins.PortalMessagePublisherExtensions.extendSectionPublishers.call(this,
  publishers);
    // Returns the extended collection of message publishers.
    return publishers;
  }
},
diff: /**SCHEMA_DIFF*/ [
  {
    "operation": "insert",
    "name": "PortalMessageTab",
    "parentName": "Tabs",
    "propertyName": "tabs",
    "values": {
      "items": []
    }
  },
  {
    "operation": "insert",
    "name": "PortalMessageTabContainer",
    "parentName": "PortalMessageTab",
    "propertyName": "items",
    "values": {
      "itemType": this.Terrasoft.ViewItemType.CONTAINER,
      "classes": {
        "wrapClassName": ["portal-message-content"]
      },
      "items": []
    }
  },
  {
}
```

```

        "operation": "insert",
        "name": "PortalMessageModule",
        "parentName": "PortalMessageTab",
        "propertyName": "items",
        "values": {
            "classes": {
                "wrapClassName": ["portal-message-module", "message-module"]
            },
            "itemType": this.Terrasoft.ViewItemType.MODULE,
            "moduleName": "PortalMessagePublisherModule",
            "afterrender": {
                "bindTo": "onMessageModuleRendered"
            },
            "afterrerender": {
                "bindTo": "onMessageModuleRendered"
            }
        }
    }
} /*SCHEMA_DIFF*/
);
});

```

Restricting access to web services for portal users

[Beginner](#)

[Easy](#)

[Medium](#)

[Advanced](#)

Introduction

The portal is a *Creatio* platform component whose main purpose is to implement self-service processes for your customers and partners. You can also use the portal to organize the work of internal users if you need to restrict access permissions to functionality of the primary application.

Using portal enables many external users (who are not employees of your organization) to access some of the *Creatio* data. For this, you need to manage which users (portal or company employees) have access to the **application web services ('Configuration service development' in the on-line documentation)**.

This article is valid for application version 7.13.2 and up.

More information about creating a custom configuration web service is available in the "**Configuration service development (on-line documentation)**" article.

Route prefixes for web services configuration

Route prefixes in *Creatio* enable managing access to the application web services. You can set the needed route prefix using specific *ServiceRoute* attributes of the service class (table 1).

Table 1. Route prefixes for configuration web services

Access level	Attribute	Route prefix	Example of code
For self-service portal users only	<i>SspServiceRoute</i>	/ssp	<pre>[ServiceContract] [SspServiceRoute] public class SspOnlyService : BaseService {}</pre> <p>Example of call ~/ssp/rest/SspOnlyService</p>

`~/ssp/soap/SspOnlyService`

For internal users only	<i>DefaultServiceRoute</i> or no <i>ServiceRoute</i> attribute is specified	none	<pre>[ServiceContract] public class InternalService : BaseService {}</pre> <p>or</p> <pre>[ServiceContract] [DefaultServiceRoute] public class InternalService : BaseService {}</pre>
For both: the internal and portal users	Both the <i>DefaultServiceRoute</i> and <i>SspServiceRoute</i>	/ssp or none	<pre>[ServiceContract] [DefaultServiceRoute] [SspServiceRoute] public class CommonService : BaseService {}</pre> <p>Example of call</p> <pre>~/rest/InternalService ~/soap/InternalService</pre>
The <i>ServiceRoute</i> attribute with the specified prefix (e.g., "custom")	<i>[ServiceRoute("custom")]</i>	Arbitrary route prefix of the service	<pre>[ServiceContract] [ServiceRoute("custom")] public class CustomPrefixService : BaseService {}</pre> <p>Example of call</p> <pre>~/rest/CustomPrefixService ~/soap/CustomPrefixService ~/ssp/rest/CommonService ~/ssp/soap/CommonService</pre>

You can use several attributes at the same time: *ServiceRoute*, *SspServiceRoute* and *DefaultServiceRoute*. As a result, the routes for services with all prefix variants will be created.

Restricting access to the internal API for portal users

If a portal user (*SspUserConnection*) addresses a service with a route that does not contain the "/ssp" prefix, the service will return error 403 (*Forbidden*).

Restricting access to the portal API for internal users

If an internal user (*UserConnection*) addresses the service with the route that contains the “/ssp” prefix, the service will return a page with code 403 (*Forbidden*).

Changing access to the base service

The application has a set of base services and only internal users have access to them.

To change access to the base service:

1. In the custom package, create a service with the access settings for portal users.
2. In the service, add the base service methods that must be available for portal users.
3. Change the custom or extend the base client schemas by changing the call of base service to the call of the created service (see step 1).
4. Compile the configuration.

Example of the service source code that extends access to the *ActivityUtilService* base service:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.IO;
    using System.Runtime.Serialization;
    using System.ServiceModel;
    using System.ServiceModel.Activation;
    using System.ServiceModel.Web;
    using Terrasoft.Configuration.FileUpload;
    using Terrasoft.Core.Factories;
    using Terrasoft.Web.Common;
    using Terrasoft.Web.Common.ServiceRouting;

    [ServiceContract]
    // The service is available for both: the internal and portal users
    [DefaultServiceRoute]
    [SspServiceRoute]
    [AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
    public class PartnerActivityUtilService: BaseService {
        // Base service, access needs to be extended
        private static readonly ActivityUtilService _baseService = new
ActivityUtilService();

        [OperationContract]
        [WebInvoke(Method = "POST", BodyStyle = WebMessageBodyStyle.Wrapped,
RequestFormat = WebMessageFormat.Json, ResponseFormat = WebMessageFormat.Json)]
        public Guid CreateActivityFileEntity(JsonActivityFile jsonActivityFile) {
            return _baseService.CreateActivityFileEntity(jsonActivityFile);
        }
        [OperationContract]
        [WebInvoke(Method = "POST", BodyStyle = WebMessageBodyStyle.Wrapped,
RequestFormat = WebMessageFormat.Json, ResponseFormat = WebMessageFormat.Json)]
        public Guid CreateFileEntity(JsonEntityFile jsonEntityFile) {
            return _baseService.CreateFileEntity(jsonEntityFile);
        }
    }
}
```

The ServiceStack core services

The *ServiceStack* core services (*DataService*, *ManagerService*, etc.) are available by default to the internal users only.

To make any of these methods available on the portal, tag such method with the *SspServiceAccess* attribute – in this case, the method will have an additional route with the *~/DataService/ssp/...* prefix.

If you need to have a different logic for the portal, create a new service by specifying the `SspServiceAccess` attribute for it and pass the name of the original method to its constructor. Example:

```
[SspServiceAccess(nameof(SelectQuery))]  
public class SspSelectQuery : SelectQuery  
{  
}
```

This service creates a contract whose method will be registered at the following path:

`~/DataService/ssp/SelectQuery`

Access to the `ServiceStack` methods with the “ssp” prefix is denied to the internal users. Access to the `ServiceStack` methods without the “ssp” prefix is denied to the portal users.

Machine learning service

Content

- **Introduction**
- **Creating data queries for the machine learning model**
- **Connecting a custom web-service to the machine learning functionality**

Machine learning service

Beginner

Easy

Medium

Advanced

Introduction

The machine learning (lookup value prediction) service uses statistical analysis methods for machine learning based on historical data. For example, a history of customer communications with customer support is considered historical data in Creatio. The message text, the date and the account category are used. The result is the [Responsible Group] field.

Creatio interaction with the prediction service

There are two stages of model processing in Creatio: training and prediction.

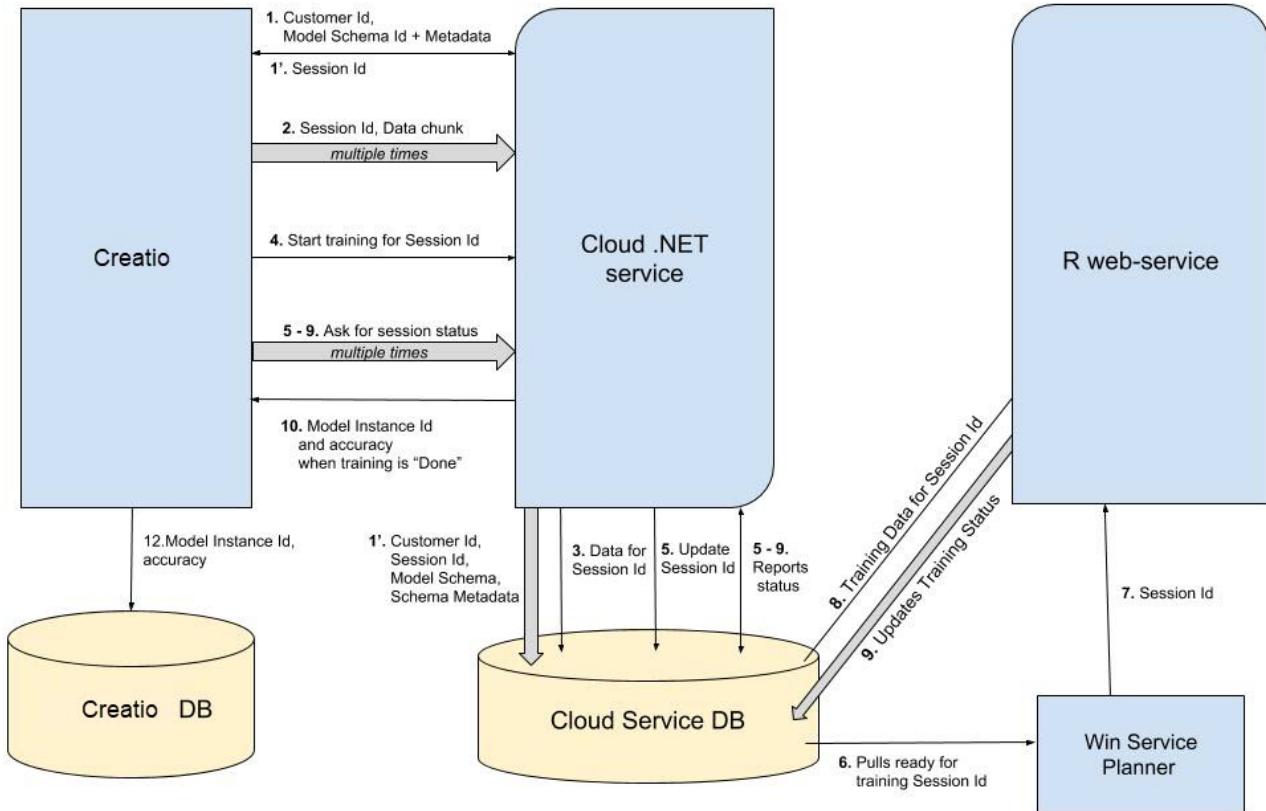
Prediction model is the algorithm which builds predictions and enables the system to automatically make decisions based on historical data.

Training

The service is “trained” at this stage (Fig. 1). Main training steps:

- Establishing a session for data transfer and training.
- Sequentially selecting a portion of data for the model and uploading it to the service.
- Requesting to include a model a training queue.
- *Training engine* processes the queue for model training, trains the model and saves its parameters to the local database.
- Creatio occasionally queries the service to get the model status.
- Once the model status is set to *Done*, the model is ready for prediction.

Fig. 1. Creatio interaction with the prediction service on the training stage

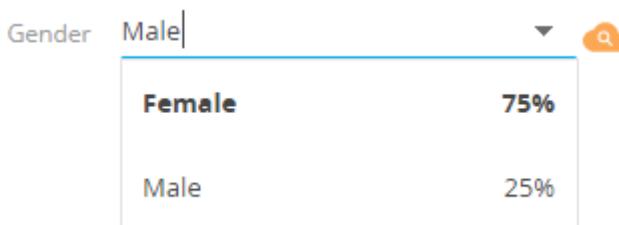


Prediction

The prediction task is performed through a call to the cloud service, indicating the Id of the model instance and the data for the prediction. The result of the service operation is a set of values with prediction probabilities, which is stored in Creatio in the *MLPrediction* table.

If there is a prediction in the *MLPrediction* table for a particular entity record, the predicted values for the field are automatically displayed on the edit page (Fig. 2).

Fig. 2. Displaying prediction data



Creatio settings and data types for working with the prediction service

Creatio setup

The following data is provided for working with the prediction service in Creatio.

1. The *CloudServicesAPIKey* system setting authenticates the Creatio instance in cloud services.
2. The record in the [ML problem types] (*MLProblemTypes*) lookup with the populated [ServiceUrl] field is the

address of the implemented prediction service.

3. The model records in the [ML model] (*MLModel*) lookup that contain information about the selected data for the model, the training period, the current training status, etc. For each model, the *MLProblemType* field must contain a reference to the correct record of the [ML problem types] lookup.
4. The *MLModelTrainingPeriodMinutes* system setting determines the frequency of model synchronization launch.

The *MLModel* lookup

The primary fields of *MLModel* lookup are given in Table 1.

Table 1. – Main *MLModel* lookup fields

Field	Data type	Purpose
Name	String	Model name
ModelInstanceId	Unique identifier	The identifier of the current model instance.
TrainedOn	Date/time	The date/time of instance training.
TriedToTrainOn	Date/time	The date/time of last training attempt.
TrainFrequency	Integer	Model retraining frequency (days).
MetaData	String	Metadata with selection column types. Uses the following JSON format:

```
{
    inputs: [
        {
            name: "Name of the field 1 in the
data sample",
            type: "Text",
            isRequired: true
        },
        {
            name: "Name of the field 2 in the
data sample",
            type: "Lookup"
        },
        //...
    ],
    output: {
        name: "Resulting field",
        type: "Lookup",
        displayName: "Name of the column to
display"
    }
}
```

In this code:

- *inputs* – a set of incoming columns for the model.
- *output* – a column, the value of which the model should predict.

Column descriptions support the following attributes:

- *name* – field name from the *TrainingsetQuery* expression.
- *type* – data type for the training engine. Supported values:
 - “Text” – text column.
 - “Lookup” – lookup column.

- “Boolean” – logical data type.
- “Numeric” – numeric type.
- “DateTime” – date and time.
- *isRequired* – mandatory field value (*true / false*). Default value – *false*.

TrainingsetQuery String

C#-expression of the training data selection. This expression should return the *Terrasoft.Core.DB.Select* class instance. For example:

```
(Select) new Select(userConnection)
    .Column("Id")
    .Column("Symptoms")
    .Column("CreatedOn")
    .From("Case", "c")
    .OrderByDesc("c", "CreatedOn")
```

Select the “Unique identifier” column type in the selection expression. This column should have the *Id* name.

If the selection expression contains a column for sorting, then this column must be present in the resulting selection.

You can find examples of queries in the **“Creating data queries for the machine learning model”** article.

RootSchemaUid Unique identifier

A link to an object schema for which the prediction will be executed.

Status String

The status of model processing (data transfer, training, ready for forecasting).

InstanceMetric Number

A quality metric for the current model instance.

MetricThreshold Number

Lowest threshold of model quality.

PredictionEnabled Logical

A flag that includes the prediction for this model.

TrainSessionId Unique identifier

Current training session.

MLProblemType Unique identifier

Machine learning problem (defines the algorithm and service url for model training).

A set of classes for training

MLModelTrainerJob: IJobExecutor, IMLModelTrainerJob – model synchronization task

Orchestrates model processing on the side of Creatio by launching data transfer sessions, starting trainings, and also checking the status of the models processed by the service. Instances are launched by default by the task scheduler through the standard Execute method of the IJobExecutor interface.

Public methods:

IMLModelTrainerJob.RunTrainer() is a virtual method that encapsulates the synchronization logic. The base implementation of this method performs the following actions:

1. Selecting models for training – the records are selected from MLModel based on the following filter:

- The *MetaData* and *TrainingsetQuery* fields are populated.
- The *Status* field is not in the *NotStarted*, *Done* or *Error* state (or not populated at all).
- *TrainFrequency* is more than 0.
- The *TrainFrequency* days have passed since the last training date (*TriedToTrainOn*).

For each record of this selection, the data is sent to the service with the help of the predictive model trainer (see below).

2. Selecting previously trained models and updating their status (if necessary).

The data transfer session for the selection starts for each suitable model. The data is sent in packages of 1000 records during the session. For each model, the selection size is limited to 75,000 records.

MLModelTrainer: IMLModelTrainer – the trainer of the prediction model.

Responsible for the overall processing of a single model during the training stage. Communication with the service is provided through a proxy to a predictive service (see below).

Public methods:

IMLModelTrainer.StartTrainSession() – sets the training session for the model.

IMLModelTrainer.Upload Data() – transfers the data according to the model selection in packages of 1000 records. The selection is limited to 75,000 records.

IMLModelTrainer.BeginTraining() – indicates the completion of data transfer and informs the service about the need to put the model in the training queue.

IMLModelTrainer.UpdateModelState – requests the service for the current state of the model and updates the Status (if necessary).

If the training was successful (*Status* contains the *Done* value), the service returns the metadata for the trained instance, particularly the accuracy of the resulting instance. If the precision is greater than or equal to the lower threshold (*MetricThreshold*), the ID of the new instance is written in the *ModelInstanceId* field.

MLServiceProxy: IMLServiceProxy – proxy to the prediction service

A wrapper class for http requests to a prediction service.

Public methods:

IMLServiceProxy.UploadData() – sends a data package for the training session.

MLServiceProxy.BeginTraining() – calls the service for setting up training in the queue

IMLServiceProxy.GetTrainingSessionInfo() – requests the current state from the service for the training session.

IMLServiceProxy.Classify(Guid modelInstanceId, Dictionary<string, object> data) – calls the prediction service of the field value for a single set of field values for the previously trained model instance. In the *Dictionary data* parameter, the field name is passed as the key, which must match the name specified in the *MetaData* field of the model lookup. If the result is successful, the method returns a list of values with the *ClassificationResult* type.

Basic properties of the *ClassificationResult* type:

- *Value* – field value.
- *Probability* – the probability of a given value in the [0:1] range. The sum of the probabilities for one list of results is close to 1 (values of about 0 can be omitted).
- *Significance* - the level of importance of this prediction. This is a string enumeration with the following options:
 - High - this field value has a distinct advantage over other values from the list. Only one element in the prediction list can have this level.
 - Medium - the value of the field is close to several other high values in the list. For example, two values in the list have a probability of 0.41 and 0.39, and all the others are significantly smaller.
 - None - irrelevant values with low probabilities.

Expanding the training model logic

The above chain of classes calls and creates instances of each other through the IOC of the *Terrasoft.Core.Factories.ClassFactory* container.

If you need to replace the logic of any component, you need to implement the appropriate interface. When you start the application, you must bind the interface in your own implementation.

Interfaces for logic expansion:

IMLModelTrainerJob – the implementation of this interface will enable you to change the set of models for training.

IMLModelTrainer – responsible for the logic of loading data for training and updating the status of models.

IMLServiceProxy - the implementation of this interface will enable yo to execute queries to arbitrary predictive services.

Auxiliary classes for forecasting

Auxiliary (utility) classes for forecasting enable you to implement two basic cases:

1. Prediction at the time of creating or updating an entity record on the server.
2. Prediction when the entity is changed on the edit page.

While predicting on the Creatio server side, a business process is created that responds to the entity creation/change signal, reads a set of fields, and calls the prediction service. If you get the correct result, it stores the set of field values with probabilities in the *MLClassificationResult* table. If necessary, the business process records a separate value (for example, with the highest probability) in the corresponding field of the entity.

MLEntityPredictor

A utility class that helps to predict the value of a field based on a particular model (either one or several models) for a particular entity.

Some of the public methods include:

PredictEntityValueAndSaveResult(Guid modelId, Guid entityId) – based on the model *Id* and entity *Id*, performs predictions and records the results in the resulting entity field. Works with any machine learning task: classification, scoring, numeric field prediction.

ClassifyEntityValues(List<Guid> modelIds, Guid entityId) – based on the model (or list of several models created for the same object) *Id* and entity *Id* performs classification and returns the glossary, whose key is the model object, and the values are the predicted values.

MLPredictionSaver

The utility class that assists to save the prediction results in the Creatio object.

Some of the public methods include:

SaveEntityPredictedValues(Guid schemaUid, Guid entityId, Dictionary<MLModelConfig, List<ClassificationResult>> predictedValues, Func<Entity, string, ClassificationResult, bool> onSetEntityValue) - saves the (*MLEntityPredictor.ClassifyEntityValues*) classification results in the Creatio object. By default, it saves only the result, whose *Significance* equals to "High". You can still override this behavior using the passed *onSetEntityValue* delegate. If the delegate returns *false*, the value will not be recorded in the Creatio object.

Creating data queries for the machine learning model

Beginner

Easy

Medium

Advanced

Introduction

Use the [Terrasoft.Core.DB.Select](#) class instance for queries of training data or data for predicting machine learning service (see “**Machine learning service**” and “**How to implement custom prediction model**”). It is dynamically imported by the *Terrasoft.Configuration.ML.QueryInterpreter*.

The *QueryInterpreter* interpreter does not allow the use of lambda expressions.

Use the provided *userConnection* variable as an argument of the *Terrasoft.Core.UserConnection* type in the *Select* constructor when building query expression. The column with the “*Id*” alias (the unique id of the target object instance) is a required in the query expression.

The *Select* expression can be complex. Use the following practices to simplify it:

- Dynamic adding of types for the interpreter.
- Using local variables.
- Using the *Terrasoft.Configuration.QueryExtensions* utility class.

Dynamic adding of types for the interpreter

You can dynamically add types for the interpreter. For this the *QueryInterpreter* class provides the *RegisterConfigurationType* and *.RegisterType* methods. You can use them directly in the expression. For example, instead of direct using the type id:

```
new Select(userConnection)
    .Column("Id")
    .Column("Body")
    .From("Activity")
    .Where("TypeId").IsEqual(Column.Parameter("E2831DEC-CFC0-DF11-B00F-001D60E938C6"));
```

you can use the name of a constant from dynamically registered enumeration:

```
RegisterConfigurationType("ActivityConsts");
new Select(userConnection)
    .Column("Id")
    .Column("Body")
    .From("Activity")
    .Where("TypeId").IsEqual(Column.Parameter(ActivityConsts.EmailTypeID));
```

Using local variables

You can use local variables to avoid code duplication and more convenient structuring. Constraint: the type of the variable must be statically calculated and defined by the *var* word.

For example, the query with repetitive use of delegates:

```
new Select(userConnection)
    .Column("Id")
    .Column("Body")
    .From("Activity")
    .Where("CreatedOn").IsGreater(Func.DateAddMonth(-1, Func.CurrentDateTime()))
    .And("StartDate").IsGreater(Func.DateAddMonth(-1, Func.CurrentDateTime()));
```

you can write in a following way:

```
var monthAgo = Func.DateAddMonth(-1, Func.CurrentDateTime());

new Select(userConnection)
    .Column("Id")
    .Column("Body")
    .From("Activity")
    .Where("StartDate").IsGreater(monthAgo)
    .And("ModifiedOn").IsGreater(monthAgo);
```

Using the *Terrasoft.Configuration.QueryExtensions* utility class

The *Terrasoft.Configuration.QueryExtensions* utility class provides several extending methods for the *Terrasoft.Core.DB.Select*. This enables to build more compact queries.

As the *object sourceColumn* argument you can use following types (they will be transformed to the *Terrasoft.Core.DB.QueryColumnExpression*) for all extending methods:

- *System.String* – the name of the column in the “TableAlias.ColumnName as ColumnAlias” format (where the TableAlias and ColumnAlias are optional) or “*” – all columns.
- *Terrasoft.Core.DB.QueryColumnExpression* – will be added without changes.

- *Terrasoft.Core.DB.IQueryColumnExpressionConvertible* – will be converted.
- *Terrasoft.Core.DB.Select* – will be considered as subquery.

An exception will be thrown if the type is not supported.

Terrasoft.Configuration.QueryExtensions use cases

1. The public static Select Cols(this Select select, params object[] sourceColumns) method

Adds specified columns or subexpressions to the query.

Using the Cols() extension method, instead of the following expression:

```
new Select(userConnection)
    .Column("L", "Id")
    .Column("L", "QualifyStatusId")
    .Column("L", "LeadTypeId")
    .Column("L", "LeadSourceId")
    .Column("L", "LeadMediumId").As("LeadChannel")
    .Column("L", "BusinessPhone").As("KnownBusinessPhone")
    .From("Lead").As("L");
```

you can write:

```
new Select(userConnection).Cols(
    "L.Id",
    "L.QualifyStatusId",
    "L.LeadTypeId",
    "L.LeadSourceId",
    "L.LeadMediumId AS LeadChannel",
    "L.BusinessPhone AS KnownBusinessPhone")
    .From("Lead").As("L");
```

2. The public static Select Count(this Select select, object sourceColumn) method

Adds an aggregation column to calculate the number of non-empty values to the query.

For example, instead:

```
var activitiesCount = new Select(userConnection)
    .Column(Func.Count(Column.Asterisk()))
    .From("Activity")
```

you can write:

```
var activitiesCount = new Select(userConnection)
    .Count("*") // You can also specify the column name.
    .From("Activity")
```

3. The public static Select Coalesce(this Select select, params object[] sourceColumns) method

Adds a column with the function of determining the first value not equal to NULL to the query.

For example, instead:

```
new Select(userConnection)
    .Cols("L.Id")
    .Column(Func.Coalesce(
        Column.SourceColumn("L", "CountryId"),
        Column.SourceColumn("L", "CountryId"),
        Column.SourceColumn("L", "CountryId")))
    .As("CountryId")
    .From("Lead").As("L")
```

```
.LeftOuterJoin("Contact").As("C").On("L", "QualifiedContactId").IsEqual("C",
"Id")
    .LeftOuterJoin("Account").As("A").On("L", "QualifiedAccountId").IsEqual("A",
"Id");
```

you can write:

```
new Select(userConnection)
    .Cols("L.Id")
    .Coalesce("L.CountryId", "C.CountryId", "A.CountryId").As("CountryId")
    .From("Lead").As("L")
        .LeftOuterJoin("Contact").As("C").On("L", "QualifiedContactId").IsEqual("C",
"Id")
            .LeftOuterJoin("Account").As("A").On("L", "QualifiedAccountId").IsEqual("A",
"Id");
```

4. The public static **Select DateDiff(this Select select, DateDiffQueryFunctionInterval interval, object startDateExpression, object endDateExpression) method**

Adds a column that specifies the date difference to the query.

For example, instead:

```
new Select(_userConnection)
    .Cols("Id")
    .Column(Func.DateDiff(DateDiffQueryFunctionInterval.Day,
        Column.SourceColumn("L", "CreatedOn"),
    Func.CurrentDateTime()).As("LeadAge")
    .From("Lead").As("L");
```

you can write:

```
var day = DateDiffQueryFunctionInterval.Day;
new Select(userConnection)
    .Cols("L.Id")
    .DateDiff(day, "L.CreatedOn", Func.CurrentDateTime()).As("LeadAge")
    .From("Lead").As("L");
```

5. public static **Select IsNull(this Select select, object checkExpression, object replacementValue)**

Adds a column with the function replacing NULL value with a replacement expression.

For example, instead:

```
new Select(userConnection).Cols("Id")
    .Column(Func.IsNull(
        Column.SourceColumn("L", "CreatedOn"),
        Column.SourceColumn("L", "ModifiedOn")))
    .From("Lead").As("L");
```

you can write:

```
new Select(userConnection).Cols("L.Id")
    .IsNull("L.CreatedOn", "L.ModifiedOn")
    .From("Lead").As("L");
```

Connecting a custom web-service to the machine learning functionality

Beginner

Easy

Medium

Advanced

Introduction

Creatio 7.16.2 and up supports connecting custom web-services to the machine learning functionality.

You can implement typical machine learning problems (classification, scoring, numerical regression) or other similar problems (for example, customer churn forecast) using a custom web-service. This article covers the procedure for connecting a custom web-service implementation of a prediction model to Creatio.

The main principles of the machine learning service operation are covered in the “**Machine learning service**” article.

The general procedure of connecting a custom web-service to the machine learning service is as follows:

1. Create a machine learning web-service engine.
2. Expand the problem type list of the machine learning service.
3. Implement a machine learning model.

Create a machine learning web-service engine

A custom web-service must implement a service contract for model training and making forecasts based on an existing prediction model. You can find a sample [Swagger](#) service contract of the Creatio machine learning service at:

<https://demo-ml.bpmonline.com/swagger/index.html#/MLService>

Required methods:

- **/session/start** – starting a model training session.
- **/data/upload** – uploading data for an active training session.
- **/session/info/get** – getting the status of the training session.
- **<training start custom method>** – a user-defined method to be called by Creatio when the data upload is over. The model training process must not be terminated until the execution of the method is complete. A training session may last for an indefinite period (minutes or even hours). When the training is over, the **/session/info/get** method will return the training session status: either *Done* or *Error* depending on the result. Additionally, if the model is trained successfully, the method will return a model instance summary (*ModelSummary*): metrics type, metrics value, instance ID, and other data.
- **<Prediction custom method>** – an arbitrary signature method that will make predictions based on a trained prediction model referenced by the ID.

Web-service development with the Microsoft Visual Studio IDE is covered in the “**Developing the configuration server code in the user project**” article.

Expanding the problem type list of the machine learning service

To expand the problem type list of the Creatio machine learning service, add a new record to the **MLProblemType** lookup. You must specify the following parameters:

- **Service endpoint Url** – the URL endpoint for the online machine learning service.
- **Training endpoint** – the endpoint for the training start method.
- **Prediction endpoint** – the endpoint for the prediction method.

You will need the ID of the new problem type record to proceed. Look up the ID in the [dbo.MLProblemType] DB table.

Implementing a machine learning model

To configure and display a machine learning model, you may need to extend the **MLModelPage** mini-page schema.

Implementing IMLPredictor

Implement the *Predict* method. The method accepts data exported from the system by object (formatted as *Dictionary<string, object>*, where *key* is the field name and *value* is the field value), and returns the prediction value. This method may use a proxy class that implements the *IMLServiceProxy* interface to facilitate web-service calls.

Implementing IMLEntityPredictor

Initialize the *ProblemTypeId* property with the ID of the new problem type record created in the *MLProblemType* lookup. Additionally, implement the following methods:

- **SaveEntityPredictedValues** – the method retrieves the prediction value and saves it for the system entity, for which the prediction process is run. If the returned value is of the *double* type or is similar to classification results, you can use the methods provided in the *PredictionSaver* auxiliary class.
- **SavePrediction** (optional) – the method saves the prediction value with a reference to the trained model instance and the ID of the entity (*entityId*). For basic problems, the system provides the *MLPrediction* and *MLClassificationResult* entities.

Extending IMLServiceProxy and MLServiceProxy (optional)

You can extend the existing *IMLServiceProxy* interface and the corresponding implementations in the prediction method of the current problem type. In particular, the *MLServiceProxy* class provides the *Predict* generic method that accepts contracts for input data and for prediction results.

Implementing IMLBatchPredictor

If the web-service is called with a large set of data (500 instances and more), implement the *IMLBatchPredictor* interface. You must implement the following methods:

- **FormatValueForSaving** – returns a converted prediction value ready for database storage. In case a batch prediction process is running, the record is updated using the *Update* method rather than Entity instances to speed up the process.
- **SavePredictionResult** – defines how the system will store the prediction value per entity. For basic ML problems, the system provides *MLPrediction* and *MLClassificationResult* objects.

Case description

Connect a custom implementation of predictive scoring to Creatio

Source code

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

Case implementation algorithm

1. Create a machine learning web-service engine

A sample implementation of a machine learning web service for ASP.Net Core 3.1 can be downloaded [here](#).

Implement the *MLService* machine learning web-service.

Declare the required methods:

- **/session/start**
- **/data/upload**
- **/session/info/get**
- **/fakeScorer/beginTraining**
- **/fakeScorer/predict**

The complete source code of the module is available below:

```
namespace FakeScoring.Controllers
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Net;
    using Microsoft.AspNetCore.Mvc;
    using Terrasoft.ML.Interfaces;
```

```
using Terrasoft.ML.Interfaces.Requests;
using Terrasoft.ML.Interfaces.Responses;

[ApiController]
[Route("")]
public class MLService : Controller
{
    public const string FakeModelInstanceId = "{BFC0BD71-19B1-47B1-8BC4-D761D9172667}";

    private List<ScoringOutput.FeatureContribution>
GenerateFakeContributions(DatasetValue record) {
    var random = new Random(42);
    return record.Select(columnValue => new ScoringOutput.FeatureContribution
{
    Name = columnValue.Key,
    Contribution = random.NextDouble(),
    Value = columnValue.Value.ToString()
}).ToList();
}

[HttpGet("ping")]
public JsonResult Ping() {
    return new JsonResult("Ok");
}

/// <summary>
/// Handshake request to service with the purpose to start a model training
session.
/// </summary>
/// <param name="request">Instance of <see
 cref="StartSessionRequest"/>.</param>
/// <returns>Instance of <see cref="StartSessionResponse"/>.</returns>
[HttpPost("session/start")]
public StartSessionResponse StartSession(StartSessionRequest request) {
    return new StartSessionResponse {
        SessionId = Guid.NewGuid()
    };
}

/// <summary>
/// Uploads training data.
/// </summary>
/// <param name="request">The upload data request.</param>
/// <returns>Instance of <see cref="JsonResult"/>.</returns>
[HttpPost("data/upload")]
public JsonResult UploadData(UploadDataRequest request) {
    return new JsonResult(string.Empty) {
        StatusCode = (int) HttpStatusCode.OK
    };
}

/// <summary>
/// Begins fake scorer training on uploaded data.
/// </summary>
/// <param name="request">The scorer training request.</param>
/// <returns>Simple <see cref="JsonResult"/>.</returns>
[HttpPost("fakeScorer/beginTraining")]
public JsonResult BeginScorerTraining(BeginScorerTrainingRequest request) {
    // Start training work here. It doesn't have to be done by the end of
this request.
    return new JsonResult(string.Empty) {
```

```
        StatusCode = (int) HttpStatusCode.OK
    };
}

/// <summary>
/// Returns current session state and model statistics, if training is
complete.
/// </summary>
/// <param name="request">Instance of <see
 cref="GetSessionInfoRequest"/>.</param>
/// <returns>Instance of <see cref="GetSessionInfoResponse"/> with detailed
state info.</returns>
[HttpPost("session/info/get")]
public GetSessionInfoResponse GetSessionInfo(GetSessionInfoRequest request) {
    var response = new GetSessionInfoResponse {
        SessionState = TrainSessionState.Done,
        ModelSummary = new ModelSummary {
            DataSetSize = 100500,
            Metric = 0.79,
            MetricType = "Accuracy",
            TrainingTimeMinutes = 5,
            ModelInstanceId = new Guid(FakeModelInstanceId)
        }
    };
    return response;
}

/// <summary>
/// Performs fake scoring prediction.
/// </summary>
/// <param name="request">Request object.</param>
/// <returns>Scoring rates.</returns>
[HttpPost("fakeScorer/predict")]
public ScoringResponse Predict(ExplainedScoringRequest request) {
    List<ScoringOutput> outputs = new List<ScoringOutput>();
    var random = new Random(42);
    foreach (var record in request.PredictionParams.Data) {
        var output = new ScoringOutput {
            Score = random.NextDouble()
        };
        if (request.PredictionParams.PredictContributions) {
            output.Bias = 0.03;
            output.Contributions = GenerateFakeContributions(record);
        }
        outputs.Add(output);
    }
    return new ScoringResponse { Outputs = outputs };
}
}
```

2. Expand the problem list of the machine learning service.

To expand the problem list:

1. Open the System Designer by clicking . Go to the [System setup] block → click [Lookups].
 2. Select the [ML problem types] lookup.
 3. Add a new record.

In the record, specify (Fig.1):

- [Name] – "Fake scoring".

- [Service endpoint Url] – "<http://localhost:5000/>".
- [Training endpoint] – "/fakeScorer/beginTraining".

Fig. 1. Setting up the parameters of the problem type

Name	Description	Service endpoint Url	Training endpoint
Numeric prediction			/regressor/beginTraining
Predictive scoring	Predictive scoring		/scorer/beginTraining
Fake scoring		http://localhost:5000/	/fakeScorer/beginTraining
Recommendation	Recommendation based on Coll...		/cf/beginTraining
Lookup prediction	Entity lookup field prediction		/classifier/beginTraining

The ID of the added record is `319c39fd-17a6-453a-bceb-57a398d52636`.

3. Implement a machine learning model

Execute the [Add] → [Additional] → [Schema of the Edit Page View Model] menu command on the [Schemas] tab in the [Configuration] section of the custom package. The newly created module should inherit the *MLModelPage* base page functionality defined in the *ML* package. Specify this schema as the parent one for a new schema.

Specify the following parameters for the created object schema (Fig. 2):

- [Title] – “FakeScoringMLModelPage”.
- [Name] – "UsrMLModelPage".
- [Parent object] – select “MLModelPage”.

Fig. 2. Setting up the mini-page view model schema

Title	FakeScoringMLModelPage
Name	UsrMLModelPage
Package	sdkMLExtensionPackage
Parent object	MLModelPage

Overload the **getIsScoring** base method to make the style of the new mini-page identical to the mini-page for creating a predictive scoring model. The complete source code of the module is available below:

Type your example code here. It will be automatically colorized when you [switch](#) to Preview or build the help system.

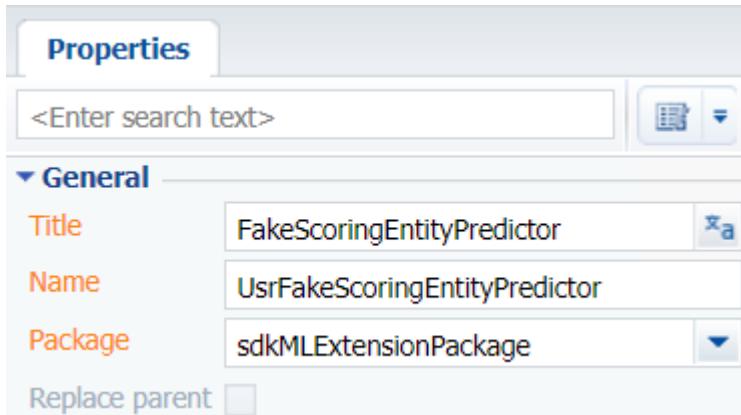
After making changes, save and publish the schema.

Go to the [Advanced settings] section -> [Configuration] -> Custom package -> the [Schemas] tab. Click [Add] -> [Source code]. Learn more about creating a schema of the [Source Code] type in the “[Creating the \[Source code\] schema](#)” article.

Specify the following parameters for the created object schema (Fig. 3):

- [Title] – “FakeScoringEntityPredictor”.
- [Name] – "UsrFakeScoringEntityPredictor".

Fig. 3. – Setting up the [Source Code] type object schema



Implement the *Predict* method that accepts data exported from the system by object and returns the prediction value. The method uses a proxy class that implements the *IMLServiceProxy* interface to facilitate web-service calls.

Initialize the *ProblemTypeId* property with the ID of the new problem type record created in the *MLProblemType* lookup and implement the **SaveEntityPredictedValues** and **SavePrediction** methods. The complete source code of the module is available below:

```
namespace Terrasoft.Configuration.ML
{
    using System;
    using System.Collections.Generic;
    using Core;
    using Core.Factories;
    using Terrasoft.ML.Interfaces;

    [DefaultBinding(typeof(IMLEntityPredictor), Name = "319C39FD-17A6-453A-BCEB-
57A398D52636")]
    [DefaultBinding(typeof(IMLPredictor<ScoringOutput>), Name = "319C39FD-17A6-453A-
BCEB-57A398D52636")]
    public class FakeScoringEntityPredictor: MLBaseEntityPredictor<ScoringOutput>,
    IMLEntityPredictor,
    IMLPredictor<ScoringOutput>
    {

        public FakeScoringEntityPredictor(UserConnection userConnection) :
        base(userConnection)
        {

            protected override Guid ProblemTypeId => new Guid("319C39FD-17A6-453A-BCEB-
57A398D52636");

            protected override ScoringOutput Predict(IMLServiceProxy proxy, MLModelConfig
model,
                Dictionary<string, object> data) {

```

```

        return proxy.FakeScore(model, data, true);
    }

    protected override List<ScoringOutput> Predict(MLModelConfig model,
        IList<Dictionary<string, object>> dataList, IMLServiceProxy proxy) {
        return proxy.FakeScore(model, dataList, true);
    }

    protected override void SaveEntityPredictedValues(MLModelConfig model, Guid
entityId,
        ScoringOutput predictedResult) {
        var predictedValues = new Dictionary<MLModelConfig, double> {
            { model, predictedResult.Score }
        };
        PredictionSaver.SaveEntityScoredValues(model.EntitySchemaId, entityId,
predictedValues);
    }

    protected override void SavePrediction(MLModelConfig model, Guid entityId,
ScoringOutput predictedResult) {
        PredictionSaver.SavePrediction(model.Id, model.ModelInstanceId,
entityId, predictedResult.Score);
    }
}
}

```

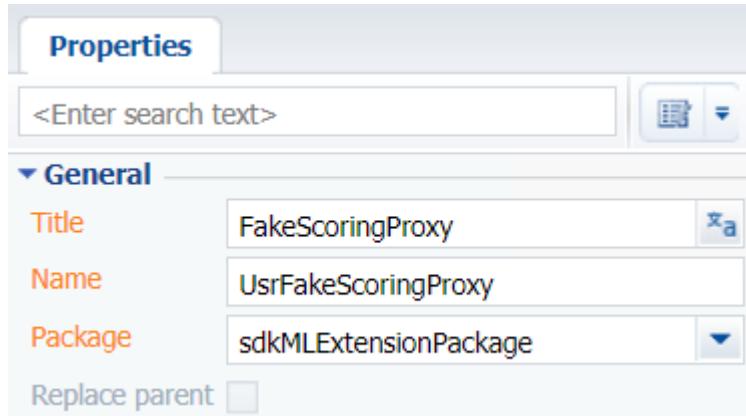
After making changes, save and publish the schema.

Go to the [Advanced settings] section -> [Configuration] -> Custom package -> the [Schemas] tab. Click [Add] -> [Source code]. Learn more about creating a schema of the [Source Code] type in the “**Creating the [Source code] schema**” article.

Specify the following parameters for the created object schema (Fig. 4):

- [Title] – "FakeScoringProxy".
- [Name] – "UsrFakeScoringProxy".

Fig. 4. – Setting up the [Source Code] type object schema



Extend the existing *IMLServiceProxy* interface and the corresponding implementations in the prediction method of the current problem type. In particular, the *MLServiceProxy* class provides the *Predict* generic method that accepts contracts for input data and prediction results.

The complete source code of the module is available below:

```

namespace Terrasoft.Configuration.ML
{
    using System;
    using System.Collections.Generic;

```

```
using System.Linq;
using Terrasoft.ML.Interfaces;
using Terrasoft.ML.Interfaces.Requests;
using Terrasoft.ML.Interfaces.Responses;

public partial interface IMLServiceProxy
{
    ScoringOutput FakeScore(MLModelConfig model, Dictionary<string, object> data,
bool predictContributions);
    List<ScoringOutput> FakeScore(MLModelConfig model, IList<Dictionary<string,
object>> dataList,
        bool predictContributions);
}

public partial class MLServiceProxy : IMLServiceProxy
{
    public ScoringOutput FakeScore(MLModelConfig model, Dictionary<string,
object> data, bool predictContributions) {
        ScoringResponse response = Predict<ExplainedScoringRequest,
ScoringResponse>(model.ModelInstanceId,
            new List<Dictionary<string, object>> { data },
model.PredictionEndpoint,
            100, predictContributions);
        return response.Outputs.FirstOrDefault();
    }

    public List<ScoringOutput> FakeScore(MLModelConfig model,
IList<Dictionary<string, object>> dataList,
        bool predictContributions) {
        int count = Math.Min(1, dataList.Count);
        int timeout = Math.Max(ScoreTimeoutSec * count, BatchScoreTimeoutSec);
        ScoringResponse response = Predict<ScoringRequest, ScoringResponse>
(model.ModelInstanceId,
            dataList, model.PredictionEndpoint, timeout, predictContributions);
        return response.Outputs;
    }
}
}
```

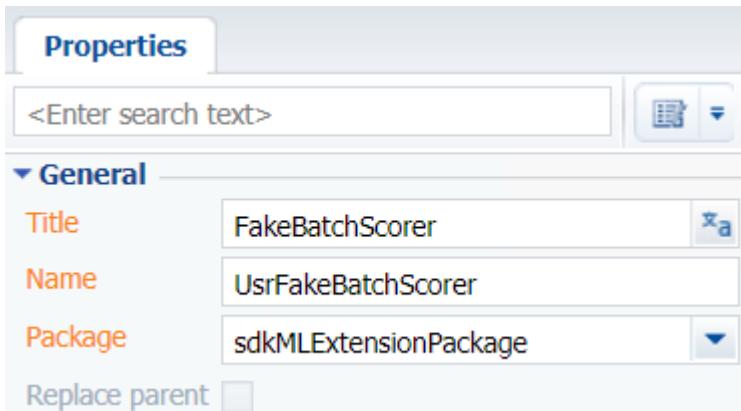
After making changes, save and publish the schema.

Go to the [Advanced settings] section -> [Configuration] -> Custom package -> the [Schemas] tab. Click [Add] -> [Source code]. Learn more about creating a schema of the [Source Code] type in the “**Creating the [Source code] schema**” article.

Specify the following parameters for the created object schema (Fig. 5):

- [Title] – "FakeBatchScorer".
- [Name] – "UsrFakeBatchScorer".

Fig. 5. – Setting up the [Source Code] type object schema



To use the batch prediction functionality, implement the *IMLBatchPredictor* interface, and the **FormatValueForSaving** and **SavePredictionResult** methods.

The complete source code of the module is available below:

```
namespace Terrasoft.Configuration.ML
{
    using System;
    using Core;
    using Terrasoft.Core.Factories;
    using Terrasoft.ML.Interfaces;

    [DefaultBinding(typeof(IMLBatchPredictor), Name = "319C39FD-17A6-453A-BCEB-
57A398D52636")]
    public class FakeBatchScorer : MLBatchPredictor<ScoringOutput>
    {

        public FakeBatchScorer(UserConnection userConnection) : base(userConnection)
    }

        protected override object FormatValueForSaving(ScoringOutput scoringOutput) {
            return Convert.ToInt32(scoringOutput.Score * 100);
        }

        protected override void SavePredictionResult(Guid modelId, Guid
modelInstanceId, Guid entityId,
            ScoringOutput value) {
            PredictionSaver.SavePrediction(modelId, modelInstanceId, entityId,
value);
        }
    }
}
```

After making changes, save and publish the schema.

Sending emails

Content

- **Sending emails from existing account**
- **Sending emails using the explicit account credentials**

Sending emails from existing account

Beginner

Easy

Medium

Advanced

Introduction

Creatio version 7.16 and up supports sending emails from an existing email account.

In Creatio, you can send emails using both customer and developer means. By developer means, you can send emails:

- Using an existing account
- Using the explicit account credentials

This article covers sending emails from existing accounts. Read more about sending emails using your account credentials in the “**Sending emails using the explicit account credentials**” article.

Sending an email from an existing account is done by using a business process. To set up a business process, use the [\[Auto-generated page\]](#) and [\[Script task\]](#) elements.

Preparing and sending an email

To send an email from an existing account:

1. Create a business process. The process must include [Auto-generated page] and [Script task] required elements.
2. Create a config file for the email.
3. Add an attachment (optional).
4. Run the business process to send the email.

Creating a config file for the email

To create a configuration file of the email, use the *Terrasoft.Mail.Sender.EmailMessage* class. Populate the parameters below to ensure the validity of your email:

```
var message = new Terrasoft.Mail.Sender.EmailMessage {
    // Sender email address.
    From = "Sender@email.com",
    // Recipient email addresses.
    To = List<string>{ "first@recipient.co", "second@recipient.co" },
    // Copy (optional)
    Cc = List<string>{ "first@recipient.co", "second@recipient.co" },
    // Hidden copy (optional)
    Bcc = List<string>{ "first@recipient.co", "second@recipient.co" },
    // The subject of the email.
    Subject = "Message subject",
    // Email body.
    Body = "Body",
    // Priority, Terrasoft.Mail.Sender.EmailPriority enumeration values.
    Priority = Terrasoft.Mail.Sender.EmailPriority.Normal
};
```

Adding attachments (optional)

You can add attachments to your email. To do this, populate the [Attachments] field. Attachments are essentially a list of *Terrasoft.Mail.Sender.EmailAttachments* instances.

```
// Creating an attachment.
var attachment = new Terrasoft.Mail.Sender.EmailAttachment {
    // Attachment ID.
    Id = new Guid("844F0837-EAA0-4F40-B965-71F5DB9EAE6E"),
    // Attachment name.
    Name = "attachName.txt",
    // Data.
    Data = byteData
```

```
};

// Adding the attachment to the message.
message.Attachments.Add(attachment);
```

Sending the email

To send the email, use the *Send* method of the *EmailSender* class with the email parameters and the connection configuration file.

```
// Sending the configured email message. To ignore access permissions when sending
// the message.
// set the ignoreRight parameter to "true".
emailSender.Send(message, ignoreRights);
```

Case description

Create a business process that will open a page with email parameters fields to send an email using an existing account.

Source code

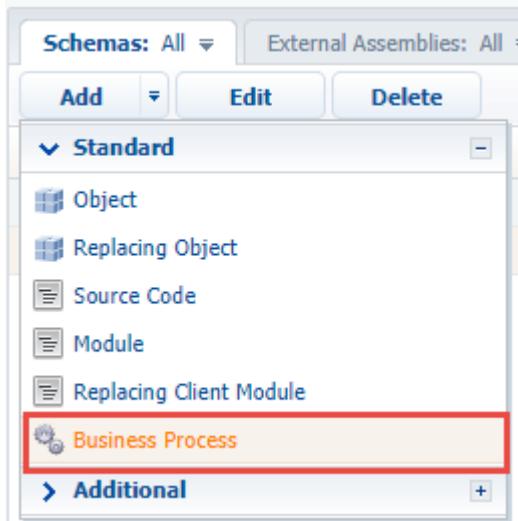
You can download the package with a sample case implementation using the following [link](#).

Case implementation algorithm

1. Create a business process

In the [Configuration] section execute the [Add] → [Business process] action (Fig. 1).

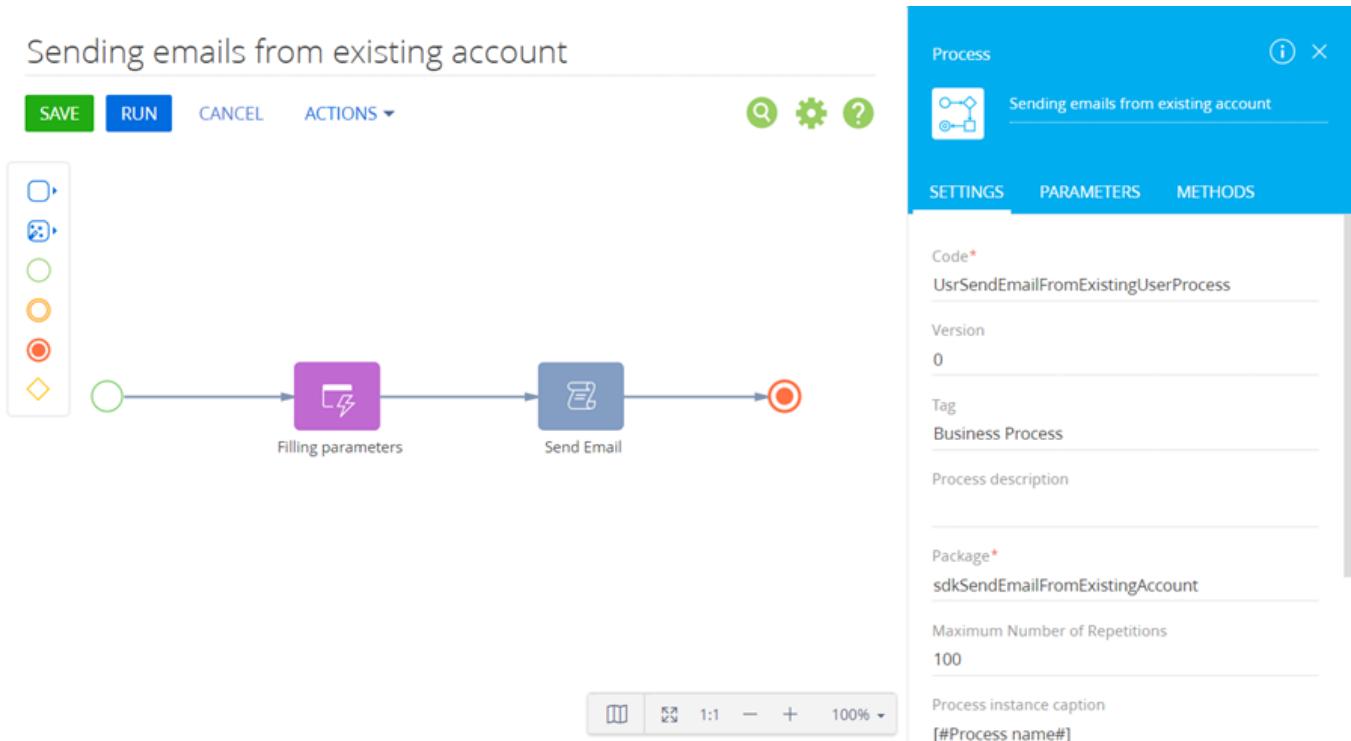
Fig. 1. [Add] → [Business process] action



In the opened Process Designer, set the following values for the properties in the setup area (Fig. 2):

- [Title] — "Sending emails from existing account".
- [Code] — "UsrSendEmailFromExistingUserProcess".

Fig. 2. The properties of the business process

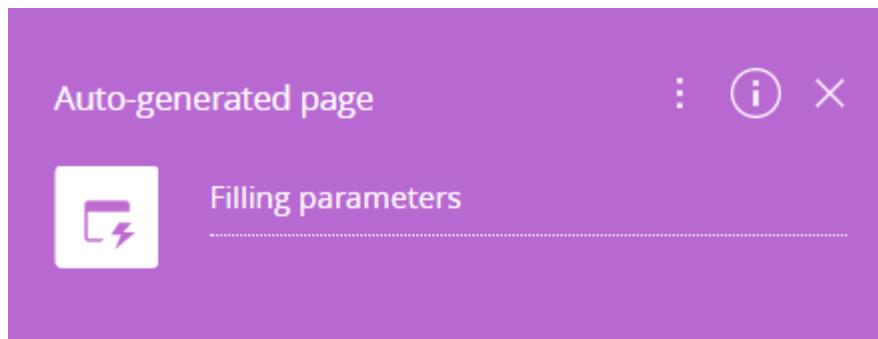


2. Add the [Auto-generated page] element

The [Auto-generated page] element enables the process to open an arbitrary page created by the user. For this element, add "Filling parameters" as a caption and set the following properties (Fig. 3):

- [Page title] — "Fill parameters for sending Email".
- [To whom should the page be shown?] – select [Formula] and specify [#System variable.Current user contact#].

Fig. 3. Auto-generated page properties



Page title

Fill parameters for sending Email

Buttons +

Page Items +

To whom should the page be shown?

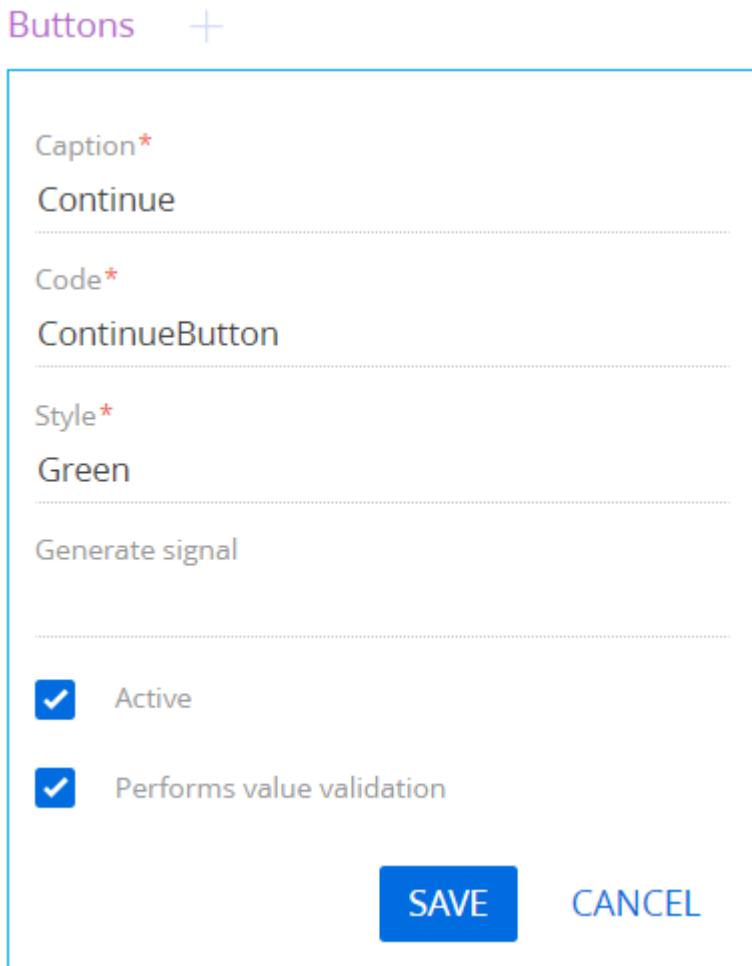
[#System variable.Current user contact#]

3. Add a button to the page

To add a [Continue] button in the [Buttons] block, click and specify the following parameters (Fig. 4):

- [Caption] — "Continue".
- [Code] — “ContinueButton”.
- [Style] — select "Green".
- Set the [Active] checkbox.
- Set the [Performs value validation] checkbox.

Fig. 4. Adding a button to the auto-generated page



Click [Save].

4. Add elements to the page.

To add an element that will contain the email sender's mailbox, click in the [Page Items] block, select the [Selection field] type and specify the following parameters (Fig. 5):

- [Title] – "Sender Mailbox".
- [Code] – "SenderMailbox".
- [Data source] – select "Mailbox synchronization settings".
- [View] – select "Drop down list".

Fig. 5. Adding an element to the auto-generated page

Page Items +

Title*

Sender Mailbox

Code*

SenderMailbox

Required

Data source*

Mailbox synchronization settings

View*

Drop down list

Value

SAVE CANCEL

Click [Save].

To add an element that will contain the email recipient's mailbox, click + in the [Page Items] block, select the [Text field] type and specify the following parameters (Fig. 6):

- [Title] – "Recipient (many recipients separated by semicolon ";")".
- [Code] – "Recipient".
- Set the [Required] checkbox.

Fig. 6. Adding an element to the auto-generated page

The screenshot shows a configuration dialog with the following fields:

- Title***: Recipient (many recipients separated by se...)
- Code***: Recipient
- Is multiline**: An unchecked checkbox.
- Required**: A checked checkbox.
- Value**: A large empty text area.

At the bottom right are two buttons: **SAVE** (in blue) and **CANCEL**.

Click [Save].

To add an element that will contain the subject of the email, click **+** in the [Page Items] block, select the [Text field] type and specify the following parameters (Fig. 7):

- [Title] – “Subject”.
- [Code] – “Subject”.
- Set the [Required] checkbox.

Fig. 7. Adding an element to the auto-generated page

The screenshot shows a configuration dialog with the following fields:

- Title***: A text input field containing "Subject".
- Code***: A text input field containing "Code".
- Value**: An empty text input field.
- Is multiline**: An unchecked checkbox.
- Required**: A checked checkbox.

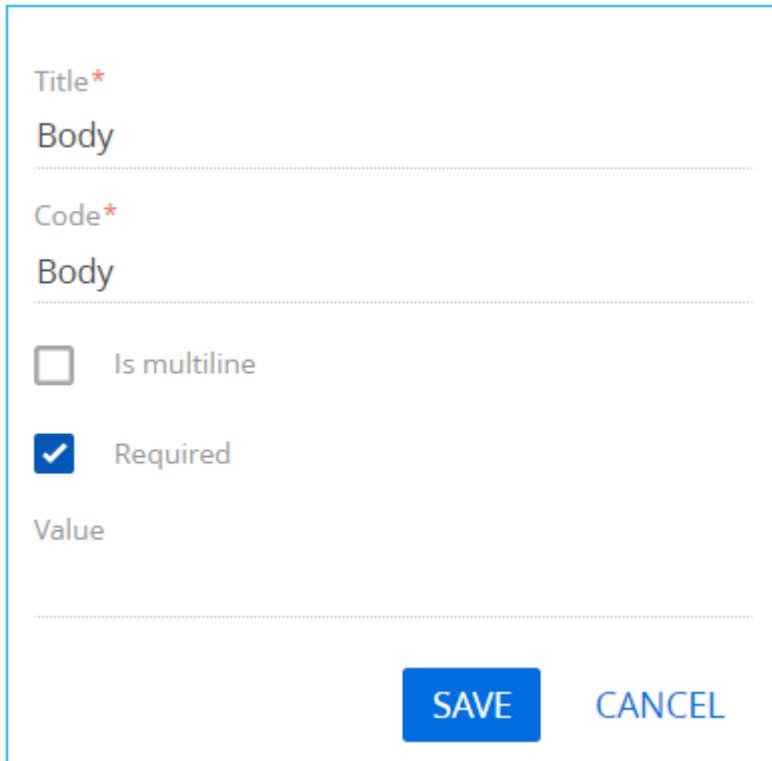
At the bottom right are two buttons: **SAVE** (in blue) and **CANCEL**.

Click [Save].

To add an element that will contain the body of the email, click **+** in the [Page Items] block, select the [Text field] type and specify the following parameters (Fig. 8):

- [Title] – “Body”.
- [Code] – “Body”.
- Set the [Required] checkbox.

Fig. 8. Adding an element to the auto-generated page



Click [Save].

5. Add a [ScriptTask] element

Set the value of the [Title] property of the [Script task] element to “Send Email” The element must execute the following program code:

```
// Id выбранного почтового ящика.
var mailboxSettingId = Get<Guid>("SenderId");
// Создание экземпляра EmailClientFactory.
var emailClientFactory = ClassFactory.Get<IEmailClientFactory>(
    new ConstructorArgument("userConnection", UserConnection));
// Создание экземпляра IEEmailSender.
var emailSender = ClassFactory.Get<IEEmailSender>(
    new ConstructorArgument("emailClientFactory", emailClientFactory),
    new ConstructorArgument("userConnection", UserConnection));

var entity =
UserConnection.EntitySchemaManager.GetInstanceByName("MailboxSyncSettings").CreateEnt
ity(UserConnection);
if (entity.FetchFromDB("Id", mailboxSettingId, new List<string> {
"SenderId" })) {
    // Получение почтового адреса отправителя из выбранного почтового ящика.
    var senderEmailAddress = entity.GetTypedColumnValue<string>
("SenderId");
    // Заполнение параметров отправляемого сообщения.
    var message = new Terrasoft.Mail.Sender.EmailMessage {
        // Email-адрес отправителя.
        From = senderEmailAddress,
        // Email-адреса получателей.
        To = Get<string>("Recipient").Split(';').ToList<string>(),
        // Копия (не обязательно).
        // Cc = List<string>{ "first@recipient.co", "second@recipient.co" },
        // Скрытая копия (не обязательно).
        // Bcc = List<string>{ "first@recipient.co", "second@recipient.co" },
    }
}
```

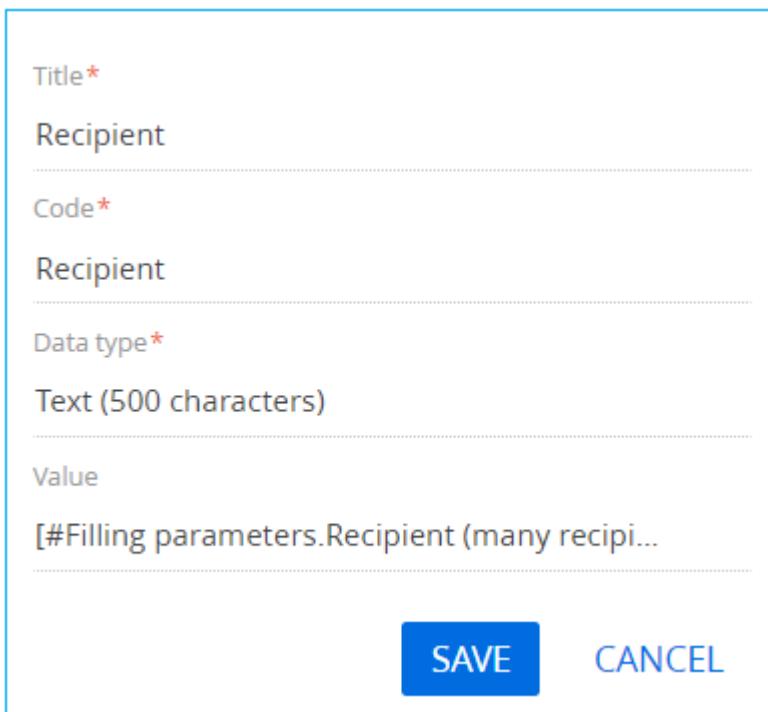
```
// Тема письма.  
Subject = Get<string>("Subject"),  
// Тело письма.  
Body = Get<string>("Body"),  
// Приоритет, значения из перечисления Terrasoft.Mail.Sender.EmailPriority.  
Priority = Terrasoft.Mail.Sender.EmailPriority.Normal  
};  
// Дополнительно можно прикреплять вложения (в примере используются тестовые  
значения).  
// Создание вложения.  
var attachment = new Terrasoft.Mail.Sender.EmailAttachment {  
    // Идентификатор вложения.  
    Id = Guid.NewGuid(),  
    // Название файла.  
    Name = "test.txt",  
    // Данные.  
    Data = Encoding.ASCII.GetBytes("some test text")  
};  
// Добавление вложения в письмо.  
message.Attachments.Add(attachment);  
// Отправка письма.  
emailSender.Send(message);  
}  
  
return true;
```

6 Add parameters

To add a business process parameter that will contain the email recipient's mailbox, execute the [Add parameters] –> [Text] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 9):

- [Title] – “Recipient”.
- [Code] – “Recipient”.
- [Value] – click  –> [Process parameter] and select the “Recipient (many recipients separated by semicolon ”;”)” process element.

Fig. 9. Adding a process parameter



The screenshot shows a configuration dialog for adding a process parameter. The fields are as follows:

- Title***: Recipient
- Code***: Recipient
- Data type***: Text (500 characters)
- Value**: [#Filling parameters.Recipient (many recipi...]

At the bottom right are two buttons: **SAVE** and **CANCEL**.

Click [Save].

To add a business process parameter that will contain the subject of the email, execute the [Add parameters] → [Text] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 10):

- [Title] – “Subject”.
- [Code] – “Subject”.
- [Value] – click ⚡ → [Process parameter] and select the “Subject” process element.

Fig. 10. Adding a process parameter

The screenshot shows a configuration dialog for adding a process parameter. The fields are as follows:

- Title***: Subject
- Code***: Subject
- Data type***: Text (500 characters)
- Value**: [#Filling parameters.Subject#]

At the bottom are two buttons: **SAVE** (in blue) and **CANCEL**.

Click [Save].

To add a business process parameter that will contain the body of the email, execute the [Add parameters] → [Text] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 11):

- [Title] – “Body”.
- [Code] – “Body”.
- [Value] – click ⚡ → [Process parameter] and select the “Body” process element.

Fig. 11. Adding a process parameter

The screenshot shows a configuration dialog with the following fields:

- Title***: Body
- Code***: Body
- Data type***: Text (500 characters)
- Value**: [#Filling parameters.Body#]

At the bottom right are two buttons: **SAVE** (blue) and **CANCEL**.

Click [Save].

To add a business process parameter that will contain the email sender's mailbox, execute the [Add parameters] —> [Other] —> [Unique identifier] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 12):

- [Title] – "Sender Mailbox".
- [Code] – “SenderMailbox”.
- [Value] – click —> [Process parameter] and select the “Sender Mailbox” process element.

Fig. 12. Adding a process parameter

The screenshot shows the 'Add parameters' dialog with the following configuration:

- Title***: Sender Mailbox
- Code***: SenderMailbox
- Data type***: Unique identifier
- Value**: [#Filling parameters.Sender Mailbox#]

At the bottom right are two buttons: **SAVE** (blue) and **CANCEL**.

Click [Save].

7. Add methods

To add business process methods, click  in the [Usings] block in the [Methods] tab of the process designer setup area, and specify `Terrasoft.Configuration` value in the [Name Space] field. Click [Save].

Using the same method, add the following namespaces:

- `Terrasoft.Mail.Sender`
- `Terrasoft.Core.Factories`
- `Terrasoft.Core`
- `Terrasoft.Mail`
- `IntegrationApi`
- `System.Linq`

Save all changes in the Process Designer.

8. Run the business process

To run the business process successfully, make sure that a user account for the email sender is available in the Creatio application. Learn more about adding a user account in the “[Integration with the MS Exchange service](#)” block of articles.

After the business process is run using the [Run] button, a page containing fields for specifying email parameters will open.

Fig. 13. The page for preparing an email

Fill parameters for sending Email

CONTINUE **CLOSE**

Sender Mailbox

Recipient (many recipients separated by semicolon ";")*

Subject*

Body*

To send an email from the corresponding email account, click [Continue].

Sending emails using the explicit account credentials

Beginner

Easy

Medium

Advanced

Introduction

Creatio version 7.16 and up supports sending emails using the explicit account credentials

In Creatio, you can send emails using both customer and developer means. By developer means, you can send emails:

- Using an existing account
- Using the explicit account credentials

This article covers sending emails using explicit account credentials. Read more about sending emails using an existing account in the "**Sending emails from existing account**" article.

Sending an email using the explicit account credentials is done by using a business process. To set up a business process, use the [\[Auto-generated page\]](#) and [\[Script task\]](#) elements.

Preparing and sending an email

To send an email using the explicit account credentials:

1. Create a business process. The process must include [Auto-generated page] and [Script task] required elements.
2. Create an instance of the *EmailClientFactory* class.
3. Create an instance of the *EmailSender* class.
4. Create a config file for connecting to the mailbox.
5. Create a config file of the email.
6. Add an attachment (if applicable).
7. Run the business process to send the email.

Creating an instance of the EmailClientFactory class.

To create an instance of the *EmailClientFactory* class, make sure you have *UserConnection* established.

```
var emailClientFactory = ClassFactory.Get<IEmailClientFactory>(
    new ConstructorArgument("userConnection", UserConnection));
```

Creating an instance of the EmailSender class.

To create an instance of the *EmailSender* class, pass the created *EmailClientFactory* instance and the *UserConnection* to the constructor.

```
var emailSender = ClassFactory.Get<IEmailSender>(
    new ConstructorArgument("emailClientFactory", emailClientFactory),
    new ConstructorArgument("userConnection", UserConnection));
```

Creating a config file for connecting to the mailbox

To create a config file for connecting to the mailbox, use the *EmailContract.DTO.Credentials* class. Populate the following parameters:

```
var credentialConfig = new EmailContract.DTO.Credentials {
    // Outbound mailserver IP address or domain
    ServiceUrl = "mail service host",
    // Can be left empty for some protocols
    Port = "Port",
    // Use SSL to encrypt the connection.
    UseSsl = false,
    // Mailbox user name
    UserName = "EmailUserName",
    // Mailbox user password
    Password = "UserPassword",
    // Mail server type (Exchange or IMAP/SMTP)
    ServerTypeId = EmailDomain.IntegrationConsts.ExchangeMailServerTypeId ||
        EmailDomain.IntegrationConsts.ImapMailServerTypeId,
    // Sender mailbox
    SenderEmailAddress = "sender@test.com"
};
```

The *ServiceUrl*, *UserName*, *Password*, *ServerTypeId*, *SenderMailbox* parameters are required.

Creating a config file of the email

To create a configuration file of the email that you are sending, use the *EmailContract.DTO.Email* class. Populate the parameters shown below to ensure the validity of your email:

```
var message = new EmailContract.DTO.Email {
    // Sender email address.
    Sender = "Sender@email.com",
    // Recipient email addresses.
    Recipients = List<string>{ "first@recipient.co", "second@recipient.co" },
    // The subject of the email.
    Subject = "Message subject",
    // Email body.
    Body = "Body",
    // Priority, EmailContract.EmailImportance enumeration values.
    Importance = EmailContract.EmailImportance.High
};
```

Adding attachments (if applicable)

You can add attachments to your email. To do this, populate the [Attachments] field. Attachments are essentially a list of *EmailContract.DTO.Attachment* instances.

```
// Creating an attachment.
var attachment = new EmailContract.DTO.Attachment {
    Name = "FileName",
    Id = "844F0837-EAA0-4F40-B965-71F5DB9EAE6E"
};
// Setting data for the attachment
attachment.SetData(byteData);
// Adding the attachment to the message.
message.Attachments.Add(attachment);
```

Sending the email

To send the email, use the *Send* method of the *EmailSender* class with the passed arguments of the email and the connection configuration file.

```
// Sending a configured email message with connection parameters
// for the mailbox of the sender To ignore access permissions when sending the
// message.
// set the ignoreRight parameter to "true".
emailSender.Send(message, credentialConfig, ignoreRights);
```

Case description

Create a business process that will open a page containing fields mapped to the email parameters to send an email using the explicit account credentials.

Source code

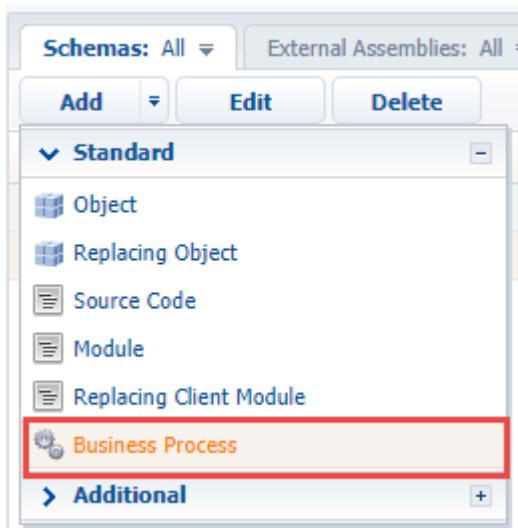
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Create a business process

In the [Configuration] section execute the [Add] → [Business process] action (Fig. 1).

Fig. 1. [Add] → [Business process] action



In the opened process designer set the following values for the properties in the setup area (Fig. 2):

- [Title] – "Sending emails using the explicit account credential".
- [Code] – “UsrSendEmailWithCredentialsProcess”.

Fig. 2. The properties of the business process

The screenshot displays the process designer with a workflow diagram on the left and a properties panel on the right. The workflow consists of three main steps: a green start event, a purple 'Filling parameters' activity, and a blue 'Send Email' activity, followed by a red end event. The 'Filling parameters' activity has a caption 'Filling parameters'. On the right, the properties panel is open for a process named 'Sending emails using the explicit account ...'. The 'SETTINGS' tab is selected, showing the following details:

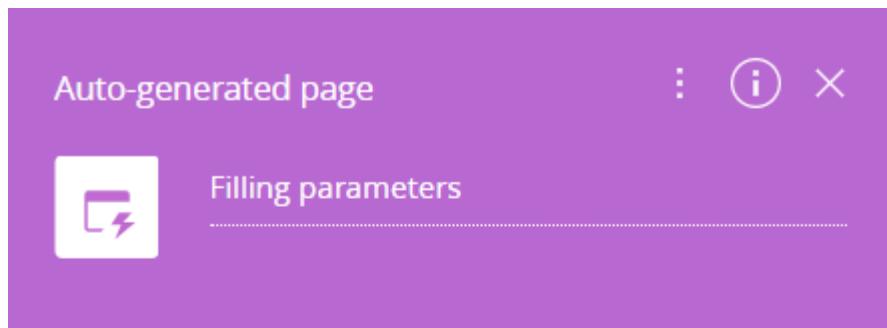
Code*	UsrSendEmailWithCredentialsProcess
Version	0
Tag	Business Process
Process description	
Package*	sdkSendEmailWithExplicitCredentials
Maximum Number of Repetitions	100
Process instance caption	!#Process.name#1

2. Add the [Auto-generated page] element

The [Auto-generated page] element enables the process to open an arbitrary page created by the user. For this element, add “Filling parameters” as a caption and set the following properties (Fig. 3):

- [Page title] – "Fill parameters for sending Email".
- [To whom should the page be shown?] – select [Formula] and specify [#System variable.Current user contact#].

Fig. 3. Auto-generated page properties



Page title

Fill parameters for sending Email

Buttons +

Page Items +

To whom should the page be shown?

[#System variable.Current user contact#]

3. Add a button to the page

To add a [Continue] button, click in the [Buttons] block and specify the following parameters (Fig. 4):

- [Caption] — "Continue".
- [Code] — “ContinueButton”.
- [Style] — select "Green".
- Set the [Active] checkbox.
- Set the [Performs value validation] checkbox.

Fig. 4. Adding a button to the auto-generated page

The screenshot shows a configuration dialog for a button element. The fields are as follows:

- Caption***: Continue
- Code***: ContinueButton
- Style***: Green
- Generate signal**: (checkbox)
- Active**: (checkbox) checked
- Performs value validation**: (checkbox) checked

At the bottom are two buttons: **SAVE** (blue) and **CANCEL**.

Click [Save].

4. Add elements to the page.

To add an element that will contain the email sender's mailbox, click **+** in the [Page Items] block, select the [Text field] type and specify the following parameters (Fig. 5):

- [Title] – "Sender Mailbox".
- [Code] – “SenderMailbox”.
- Set the [Required] checkbox.

Fig. 5. Adding an element to the auto-generated page

Page Items +

The screenshot shows a configuration dialog for a page item. At the top left is the title "Page Items" with a plus sign icon. Below it is a blue-bordered input field. Inside the field, there are several parameters:

- Title***: The value is "Sender Mailbox".
- Code***: The value is "SenderMailbox".
- Is multiline**: An unchecked checkbox.
- Required**: A checked checkbox.
- Value**: An empty text area.

At the bottom right of the dialog are two buttons: a blue "SAVE" button and a light blue "CANCEL" button.

Click [Save].

To add an element that will contain the name of the email sender, click in the [Page Items] block, select the [Text field] type and specify the following parameters (Fig. 6):

- [Title] – “User Name”.
- [Code] – “UserName”.
- Set the [Required] checkbox.

Fig. 6. Adding an element to the auto-generated page

The screenshot shows a configuration dialog with the following fields:

- Title***: User Name
- Code***: UserName
- Is multiline**: An unchecked checkbox.
- Required**: A checked checkbox.
- Value**: An empty input field.

At the bottom right are two buttons: **SAVE** (blue) and **CANCEL** (light blue).

Click [Save].

To add an element that will contain the password for email sender's mailbox, click **+** in the [Page Items] block, select the [Text field] type and specify the following parameters (Fig. 7):

- [Title] – “Password”.
- [Code] – “Password”.
- Set the [Required] checkbox.

Fig. 7. Adding an element to the auto-generated page

The screenshot shows a configuration dialog with the following fields:

- Title***: A text input field containing "Password".
- Code***: A text input field containing "Code".
- Is multiline**: An unchecked checkbox.
- Required**: A checked checkbox.
- Value**: An empty text input field.

At the bottom right are two buttons: **SAVE** (in blue) and **CANCEL**.

Click [Save].

To add an element that will contain the mail server address of the email sender, click **+** in the [Page Items] block, select the [Text field] type and specify the following parameters (Fig. 8):

- [Title] – "Service Url".
- [Code] – "ServiceUrl".
- Set the [Required] checkbox.

Fig. 8. Adding an element to the auto-generated page

The screenshot shows a configuration dialog with the following fields:

- Title***: Service Url
- Code***: ServiceUrl
- Value**: (empty)
- Is multiline**: Unchecked
- Required**: Checked

At the bottom right are two buttons: **SAVE** (blue) and **CANCEL**.

Click [Save].

To add an element that will contain the port number of the email sender's mail provider, click **+** in the [Page Items] block, select the [Integer] type and specify the following parameters (Fig. 9):

- [Title] – “Port”.
- [Code] – “Port”.

Fig. 9. Adding an element to the auto-generated page

The screenshot shows a configuration dialog with the following fields:

- Title***: Port
- Code***: Port
- Value**: (empty)
- Required**: Unchecked

At the bottom right are two buttons: **SAVE** (blue) and **CANCEL**.

Click [Save].

To add an element that will contain the cryptographic protocol to ensure a secure connection, click **+** in the [Page

Items] block, select the [Boolean] type and specify the following parameters (Fig. 10):

- [Title] – "Use Ssl".
- [Code] – “UseSsl”.

Fig. 10. Adding an element to the auto-generated page

The screenshot shows a configuration dialog with the following fields:

- Title***: Use Ssl
- Code***: UseSsl
- Value**: (empty)

At the bottom right are two buttons: **SAVE** (in blue) and **CANCEL**.

Click [Save].

To add an element that will contain the email recipient's mailbox, click **+** in the [Page Items] block, select the [Text field] type and specify the following parameters (Fig. 11):

- [Title] – "Recipient (many recipients separated by semicolon ";")".
- [Code] – “Recipient”.
- Set the [Required] checkbox.

Fig. 11. Adding an element to the auto-generated page

The screenshot shows a configuration dialog with the following fields:

- Title***: Recipient (many recipients separated by se...)
- Code***: Recipient
- Is multiline**:
- Required**:
- Value**: (empty)

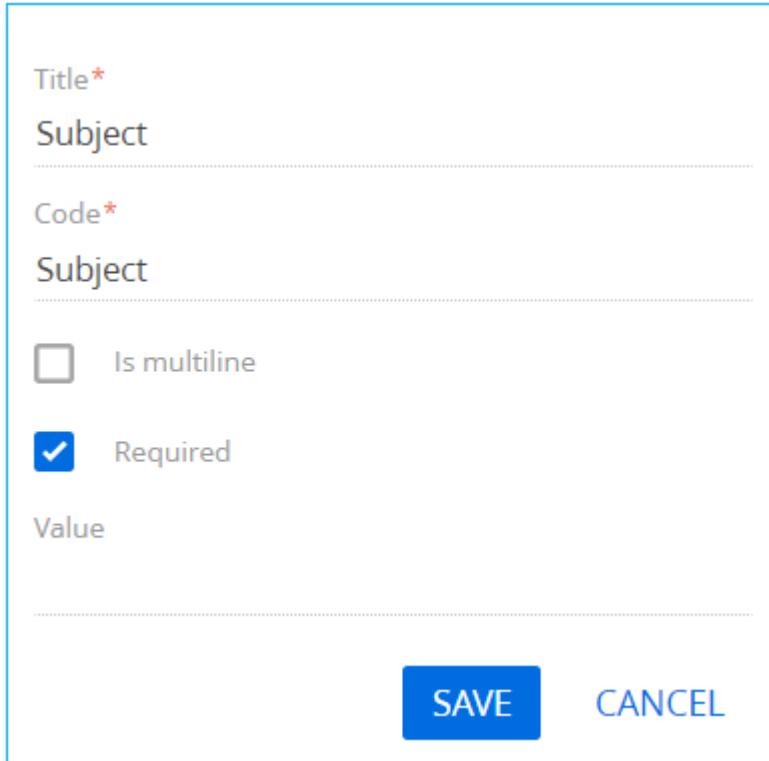
At the bottom right are two buttons: **SAVE** (in blue) and **CANCEL**.

Click [Save].

To add an element that will contain the subject of the email, click  in the [Page Items] block, select the [Text field] type and specify the following parameters (Fig. 12):

- [Title] – “Subject”.
- [Code] – “Subject”.
- Set the [Required] checkbox.

Fig. 12. Adding an element to the auto-generated page



The screenshot shows a configuration dialog for adding a page item. The 'Title' field is set to 'Subject'. The 'Code' field is also set to 'Subject'. A 'Required' checkbox is checked. The 'Value' field is empty. At the bottom, there are 'SAVE' and 'CANCEL' buttons.

Click [Save].

To add an element that will contain the body of the email, click  in the [Page Items] block, select the [Text field] type and specify the following parameters (Fig. 13):

- [Title] – “Body”.
- [Code] – “Body”.
- Set the [Required] checkbox.

Fig. 13. Adding an element to the auto-generated page

The screenshot shows a configuration dialog with the following fields:

- Title***: A text input field containing "Body".
- Code***: A text input field containing "Body".
- Is multiline**: An unchecked checkbox.
- Required**: A checked checkbox.
- Value**: A text input field.

At the bottom right are two buttons: **SAVE** (in blue) and **CANCEL**.

Click [Save].

To add an element that will contain the type of the email sender's provider, click **+** in the [Page Items] block, select the [Selection field] type and specify the following parameters (Fig. 14):

- [Title] – "Type of mail server".
- [Code] – "ServerTypeId".
- Set the [Required] checkbox.
- [Data source] – select "Mail service provider type".
- [View] – select "Drop down list".

Fig. 14. Adding an element to the auto-generated page

The screenshot shows a configuration dialog for a ScriptTask element. It includes the following fields:

- Title***: Type of mail server
- Code***: ServerTypeId
- Required**: A checked checkbox.
- Data source***: Mail service provider type
- View***: Drop down list
- Value**: An empty input field.

At the bottom right are two buttons: **SAVE** (in blue) and **CANCEL**.

Click [Save].

5. Add a [ScriptTask] element

Set the value of the [Title] property of the [Script task] element to “Send Email”. The element must execute the following program code:

```
// Создание экземпляра EmailClientFactory.
var emailClientFactory = ClassFactory.Get<EmailClientFactory>(new
ConstructorArgument("userConnection", UserConnection));
// Установка параметров подключения к почтовому сервису.
var credentialConfig = new EmailContract.DTO.Credentials {
    ServiceUrl = Get<string>("ServiceUrl"),
    Port = Get<int>("Port"),
    UseSsl = Get<bool>("UseSsl"),
    UserName = Get<string>("UserName"),
    Password = Get<string>("Password"),
    ServerTypeId = Get<Guid>("ServerTypeId"),
    SenderEmailAddress = Get<string>("SenderMailbox")
};
// Создание экземпляра IEmailSender.
var emailSender = ClassFactory.Get<IEmailSender>(new
ConstructorArgument("emailClientFactory", emailClientFactory),
    new ConstructorArgument("userConnection", UserConnection));
// Установка параметров отправляемого сообщения.
var message = new EmailContract.DTO.Email {
    Sender = credentialConfig.SenderEmailAddress,
    Recipients = Get<string>("Recipient").Split(';').ToList<string>(),
    Subject = Get<string>("Subject"),
```

```
Body = Get<string>("Body"),
Importance = EmailContract.EmailImportance.Normal,
// Если тело письма было сформировано в формате HTML.
IsHtmlBody = true,
// Дополнительно можно указать список получателей в копии, в т.ч. скрытых
// CopyRecipients = new List<string> { "user@mail.service" },
// BlindCopyRecipients = new List<string> { "user@mail.service" }
};

// Создание вложения.
var attachment = new EmailContract.DTO.Attachment {
    // Идентификатор вложения.
    Id = Guid.NewGuid().ToString(),
    // Название файла.
    Name = "test.txt",
};

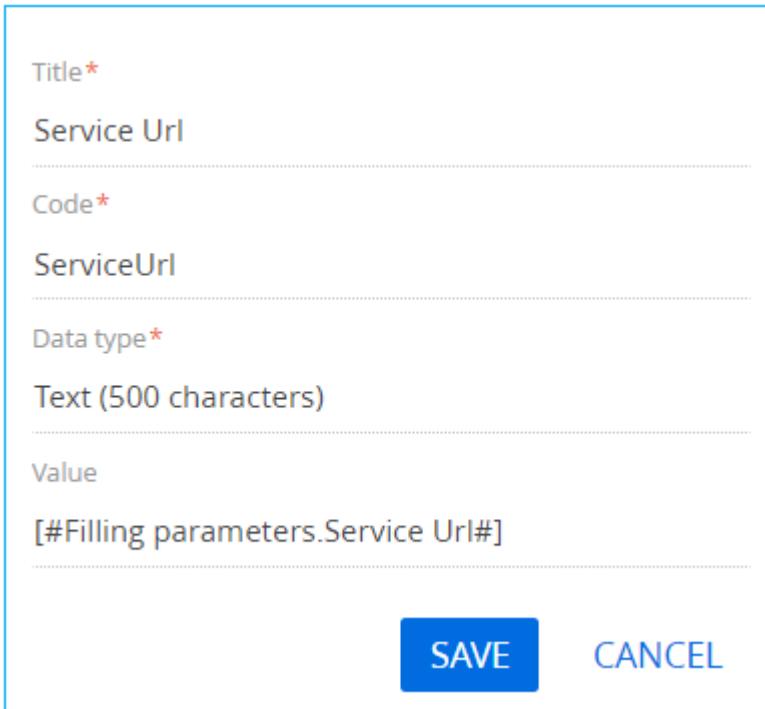
// Данные (используется тестовое значение данных в виде текста).
byte[] data = Encoding.ASCII.GetBytes("some test text");
// Добавление данных во вложение.
attachment.SetData(data);
// Добавление вложения в сообщение.
message.Attachments.Add(attachment);
// Отправка email-сообщения.
emailSender.Send(message, credentialConfig);
return true;
```

6 Add parameters

To add a business process parameter that will contain the address of the sender's mail server, execute the [Add parameters] → [Text] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 15):

- [Title] – "Service Url".
- [Code] – "ServiceUrl".
- [Value] – click  → [Process parameter] and select the "Service Url" process element.

Fig. 15. Adding a process parameter



The screenshot shows a configuration dialog for adding a process parameter. The fields are as follows:

- Title***: Service Url
- Code***: ServiceUrl
- Data type***: Text (500 characters)
- Value**: [#Filling parameters.Service Url#]

At the bottom right are two buttons: **SAVE** and **CANCEL**.

Click [Save].

To add a business process parameter that will contain the port number of the sender's email provider, execute the [Add parameters] → [Integer] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 16):

- [Title] – “Port”.
- [Code] – “Port”.
- [Value] – click ⚡ → [Process parameter] and select the “Port” process element.

Fig. 16. Adding a process parameter

The screenshot shows a dialog box for adding a process parameter. It has the following fields:

- Title***: Port
- Code***: Port
- Data type***: Integer
- Value**: [#Filling parameters.Port#]

At the bottom right are two buttons: **SAVE** (blue) and **CANCEL** (light blue).

Click [Save].

To add a business process parameter that will contain the cryptographic protocol for a secure connection, execute the [Add parameters] → [Boolean] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 17):

- [Title] – "Use Ssl".
- [Code] – "UseSsl".
- [Value] – click ⚡ → [Process parameter] and select the “Use SSL” process element.

Fig. 17. Adding a process parameter

The screenshot shows a configuration dialog box with the following fields:

- Title***: Use Ssl
- Code***: UseSsl
- Data type***: Boolean
- Value**: [#Filling parameters.Use Ssl#]

At the bottom right are two buttons: **SAVE** (blue) and **CANCEL**.

Click [Save].

To add a business process parameter that will contain the email sender's name, execute the [Add parameters] → [Text] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 18):

- [Title] – “User Name”.
- [Code] – “UserName”.
- [Data type] – select “Text (250 characters)”.
- [Value] – click → [Process parameter] and select the “User Name” process element.

Fig. 18. Adding a process parameter

The screenshot shows a configuration dialog box with the following fields:

- Title***: User Name
- Code***: UserName
- Data type***: Text (250 characters)
- Value**: [#Filling parameters.User Name#]

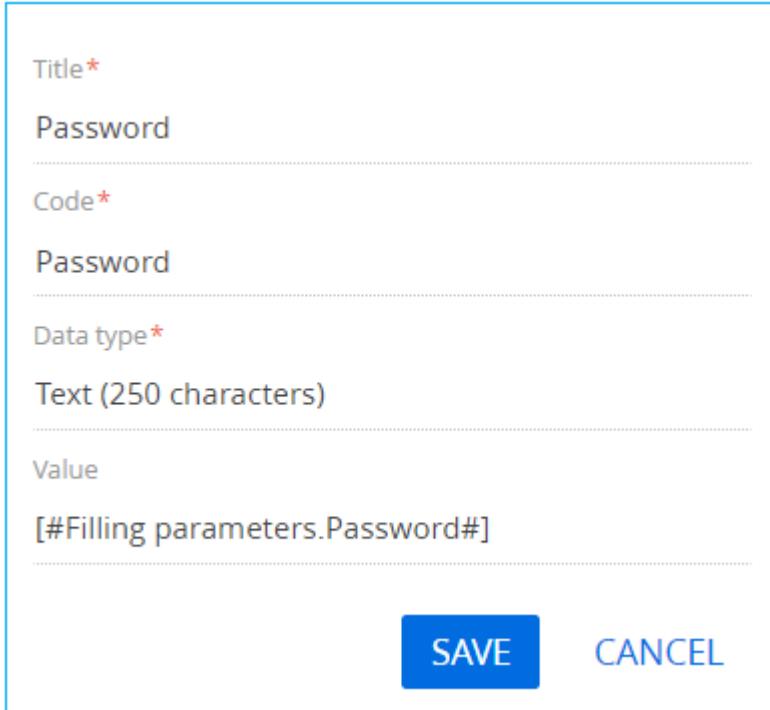
At the bottom right are two buttons: **SAVE** (blue) and **CANCEL**.

Click [Save].

To add a business process parameter that will contain the password for the email sender's mailbox, execute the [Add parameters] → [Text] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 19):

- [Title] – “Password”.
- [Code] – “Password”.
- [Data type] – select “Text (250 characters)”.
- [Value] – click  → [Process parameter] and select the “Password” process element.

Fig. 19. Adding a process parameter



The dialog box contains the following fields:

- Title*: Password
- Code*: Password
- Data type*: Text (250 characters)
- Value: [#Filling parameters.Password#]

At the bottom are two buttons: a blue **SAVE** button and a white **CANCEL** button.

Click [Save].

To add a business process parameter that will contain the email recipient's mailbox, execute the [Add parameters] → [Text] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 20):

- [Title] – “Recipient”.
- [Code] – “Recipient”.
- [Value] – click  → [Process parameter] and select the “Recipient (many recipients separated by semicolon ";")” process element.

Fig. 20. Adding a process parameter

The screenshot shows a configuration dialog for a process parameter. The fields are as follows:

- Title***: Recipient
- Code***: Recipient
- Data type***: Text (500 characters)
- Value**: [#Filling parameters.Recipient (many recipi...]

At the bottom right are two buttons: **SAVE** (in blue) and **CANCEL**.

Click [Save].

To add a business process parameter that will contain the type of the email sender's mail provider, execute the [Add parameters] → [Other] → [Unique identifier] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 21):

- [Title] – "Type of mail server".
- [Code] – "ServerTypeId".
- [Value] – click → [Process parameter] and select the "Type of mail server" process element.

Fig. 21. Adding a process parameter

The screenshot shows a configuration dialog for a process parameter named "Type of mail server". The fields are as follows:

- Title***: Type of mail server
- Code***: ServerTypeId
- Data type***: Unique identifier
- Value**: [#Filling parameters.Type of mail server#]

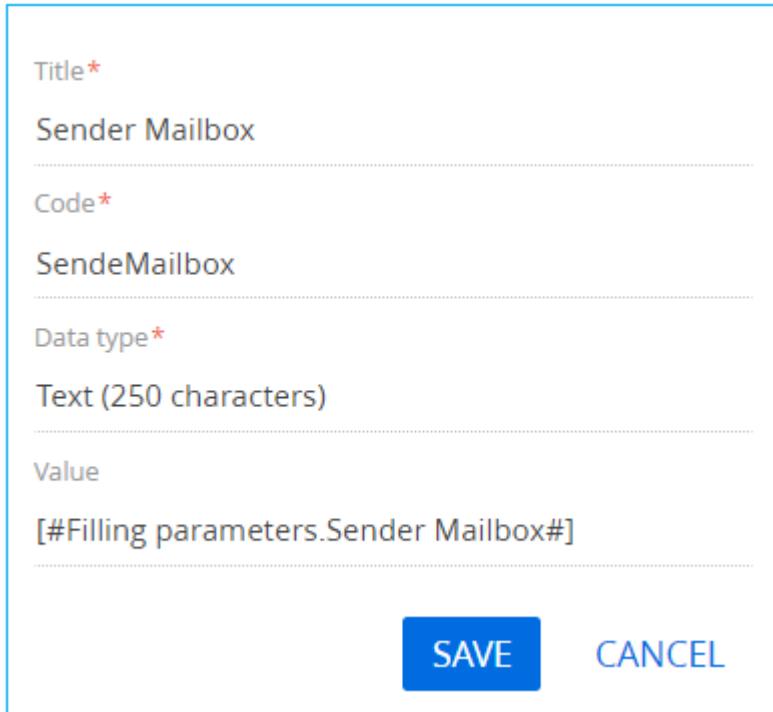
At the bottom right are two buttons: **SAVE** (in blue) and **CANCEL**.

Click [Save].

To add a business process parameter that will contain the email sender's mailbox, execute the [Add parameters] → [Text] action in the [Parameters] tab of the setup area and specify the following parameter properties (Fig. 22):

- [Title] – "Sender Mailbox".
- [Code] – "SenderMailbox".
- [Data type] – select "Text (250 characters)".
- [Value] – click  → [Process parameter] and select the "Sender Mailbox" process element.

Fig. 22. Adding a process parameter



The dialog box contains the following fields:

- Title***: Sender Mailbox
- Code***: SendeMailbox
- Data type***: Text (250 characters)
- Value**: [#Filling parameters.Sender Mailbox#]

At the bottom are two buttons: **SAVE** (highlighted in blue) and **CANCEL**.

Click [Save].

7. Add methods

To add business process methods, click  in the [Usings] block in the [Methods] tab of the process designer setup area, and specify `Terrasoft.Mail.Sender` value in the [Name Space] field. Click [Save].

Using the same method, add the following namespaces:

- `Terrasoft.Core.Factories`
- `System.Linq`

Save all changes in the Process Designer.

8. Business process launch

After the business process is run using the [Run] button, a page containing fields for specifying email parameters will open (Fig. 23).

Fig. 23. The page for preparing an email

Fill parameters for sending Email

CONTINUE

[CLOSE](#)

Sender Mailbox*

User Name*

Password*

Service Url*

Port 0

Use Ssl

Recipient (many
recipients separated by
semicolon ";"*)

Subject*

Body*

Type of mail server*

To send an email using the explicit account credentials, click [Continue].

Contact data enrichment from emails

Beginner

Easy

Medium

Advanced

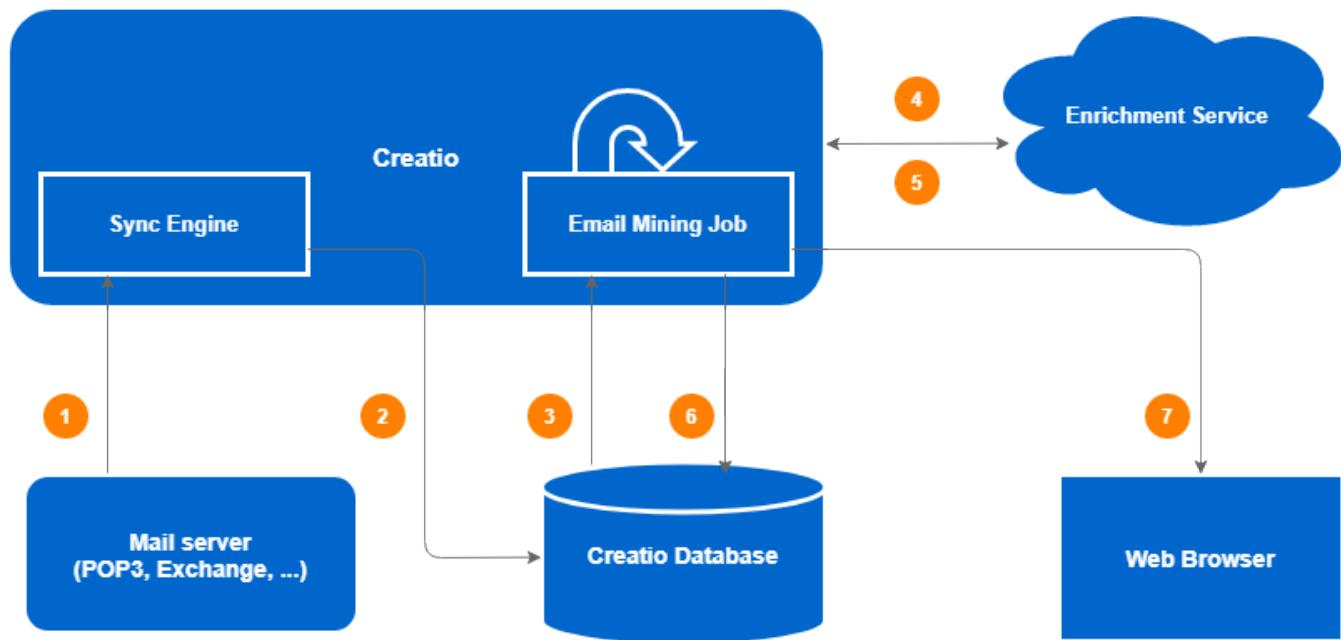
Introduction

Data enrichment from emails is available in Creatio since version 7.10.0. The system scans emails and identifies information that can be used to enrich contact data.

The enrichment process

The main data enrichment stages (Fig. 1):

Fig. 1. Receiving contact/account data from an email

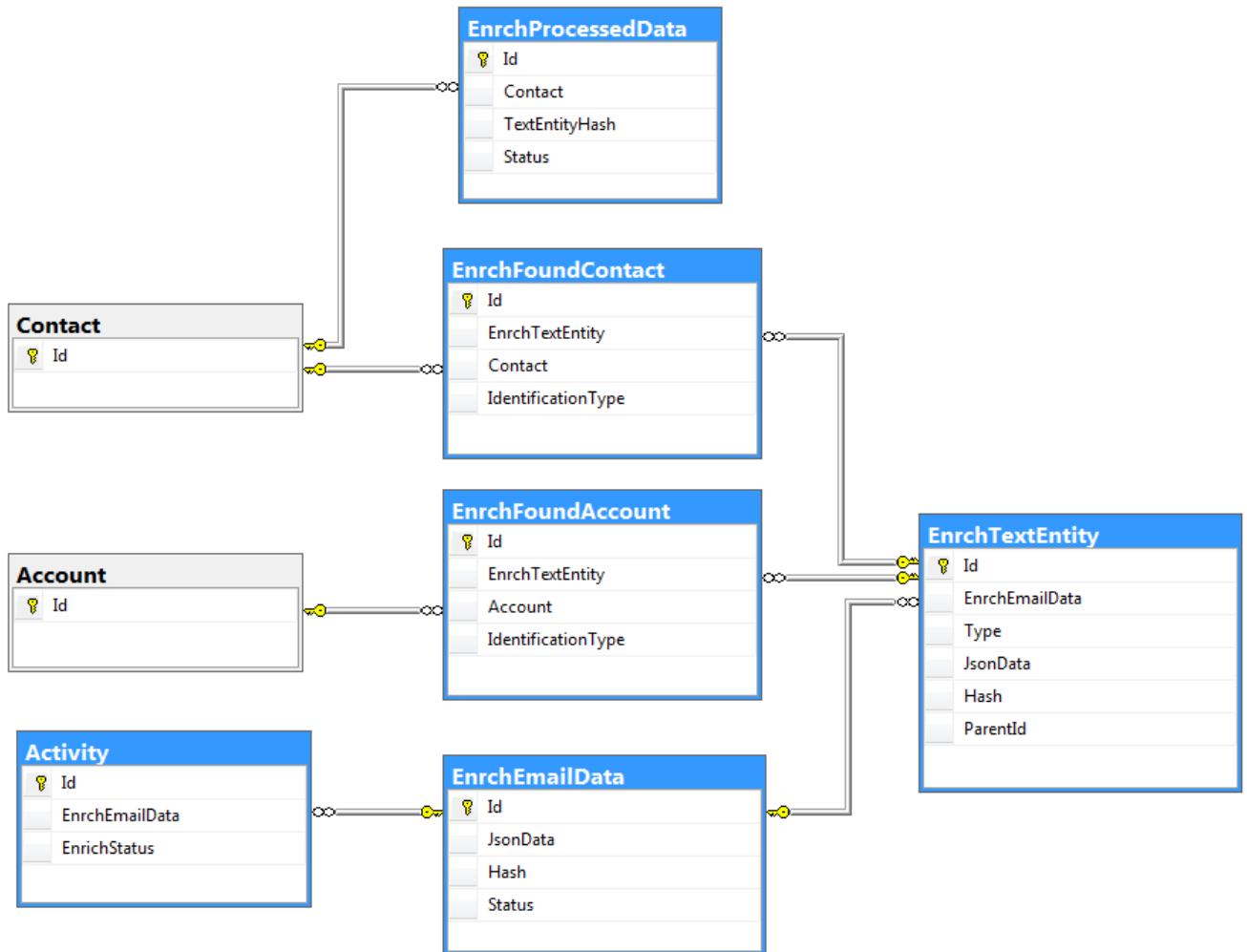


1. The existing **Sync Engine synchronization method** connects to the mail server. The mail server sends new emails to the Sync Engine.
2. Sync Engine saves the received emails in the database as activities with the *Email* type.
3. The Creatio planner periodically performs a task that starts the *Email Mining Job* process. This process selects some of the last (by creation date) activities with the *Email* type that were not previously processed by it. From each activity record, the body of the email and its format (plain-text or html) are selected.
4. The *Email Mining Job* process for each selected email sends an http request to the *Enrichment Service* cloud service.
5. *Enrichment Service* performs the following actions:
 - selects a chain of individual emails (replies) from the email;
 - selects a signature for each email (signature);
 - separates the entity (entity extraction) from the signature – contact (name, position), telephones, emails and web addresses, social networks, other means of communication, addresses, organization names.

Enrichment Service returns the gathered in the http-response as a specific structure in the JSON format.

6. The *Email Mining Job* process parses the structure received from the service and stores it in Creatio tables (see Fig. 2).

Fig. 2. The data structure for storing entities selected from the email



The main purpose of the tables shown in Fig. 2:

- *EnrichTextEntity* – stores information about one entity selected from an email. The *Type* field defines the type of this entity (contact, communication, address, organization, etc.). The data itself is stored in the *JsonData* field.
- *EnrichEmailData* – defines a set of information for enrichment selected from a single email.
- *EnrichFoundContact* – a contact in Creatio, identified by the data selected from the email. Stores a link to the Creatio contact and *EnrichTextEntity* of the *Contact* type.
- *EnrichFoundAccount* – stores information about the identified Creatio account (similar to the *EnrichFoundContact* table).
- *Activity* – the fields added to the existing activity table show the connection between the *Email* activity and the *EnrichEmailData* objects with the current status of the information extraction process.
- *EnrichProcessedData* – contains information about processed data, either accepted or rejected by the user in the enrichment process.

7. The *Email Mining Job* process notifies the user about the extraction process being finished. Messages are sent via the websocket channels to users who see the messages being processed in the communication panel. If the email contains information that may be used to enrich an associated contact, (or used to create a new contact altogether), the corresponding icon is displayed in the application interface in the upper right corner of the email (Fig. 3).

Fig. 3. Enrichment availability icon

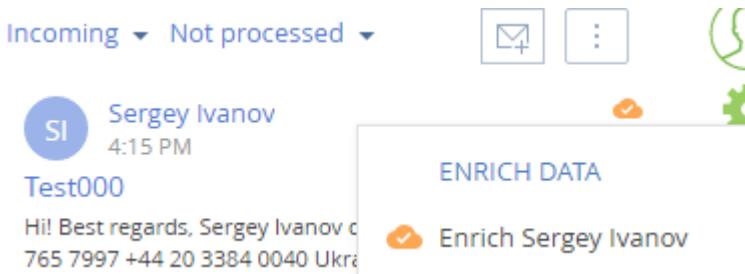


Sergey Ivanov
Yesterday



An email like that will enable the user enrich or create a new contact of the system (Fig. 4).

Fig. 4. Enrichment action



System settings

Enrichment system settings:

- *TextParsingService* – the address of the *Enrichment Service* cloud-service for data enrichment. Filled automatically for on-demand users. Required field.
- *CloudServicesAPIKey* – the key for accessing the cloud service API. Filled automatically for on-demand users. Required field.
- *EmailMiningPackageSize* – the number of emails processed at once. The *Email Mining Job* process will process as many emails each time as it is specified in this system setting. Default value – 10.
- *EmailMiningPeriodMin* – the frequency (in minutes) of running the *Email Mining Job* process.

If the value of *EmailMiningPeriodMin* is less than or equal to zero, then the process will not be scheduled and the functionality will be disabled. To re-enable the process, set the value of the setting $>=1$, restart the application pool of the application, go to the login page and enter the application.

- *EmailMiningIdentificationActualPeriod* – the period of relevance (in days) of contacts / accounts identification. If the specified period has expired and a new email is processed for the previously identified contact, the identification will be made again.

Identification sequence

Identifying contacts

1. Search by full name.
2. Search by name and last name.
3. Search by email addresses. Only those email addresses that do not belong to free or temporary email services are taken into account.
4. Search by phone. The search takes place only for the last digits of the contact's phone numbers.

If at any of the identification stages a data duplicate is detected, the identification process will be stopped.

Identifying accounts

1. Search by the [Name] or [Alternative name] columns (case-insensitive).
2. Search by web address.
3. Search by email addresses. Only those email addresses that do not belong to free or temporary email services are taken into account. From the email address, the domain is allocated and the search for the communication facilities of the account for the filter starts with one of the following domain variants:
http://<domain>, https://<domain>, http://www.<Domain>, https://www.<domain>, www.<domain>, <domain>.

If at any of the identification stages a data duplicate is detected, the identification process will be stopped.

Hashing information

The information extracted from the email is [hashed](#). As a result, in the *EnrchTextEntity* and *EnrchEmailData* tables, a hash value is written in the *Hash* field that uniquely identifies the given unit or set of extracted data in the system. This allows for two important improvements: resource savings when re-identifying contacts / accounts from a set of extracted information and grouping the information received for a contact.

Re–identification of contacts/accounts

For example, the system received an email with the signature of “John Smith Jr.”, the telephone number “123–45–67” and the address “71 Pilgrim Avenue Chevy Chase, MD 20815”. For the current data set, the system computed a hash of “Hash1” and recorded it in the *Hash* field of the *EnrchEmailData* table based on its contents. The identification of the contact revealed the contact “John Smith” in the system and recorded the result in the *EnrchFoundContact* table.

After a while the system received another email with a signature, which mentions the “John Smith Jr.” contact with the same phone and address. The system calculated the same hash for the current data set – “Hash1”, because the incoming hash data has not changed. Instead of creating new records in the *EnrchEmailData* and *EnrchTextEntity* tables and re–identifying this contact, the system found the previously created record in the Hash field of the *EnrichEmailData* table and wrote a reference to this record in the *Activity* table.

This process saves the amount of data stored and does not produce resource–intensive contact identification requests.

Grouping the highlighted information for a contact

Since each unit of the allocated information in *EnrchTextEntity* has a hash code based on its contents, when enriching the data of an existing contact, it becomes possible to use the information found in all the email in which it participated. When you select the data for enrichment, it is grouped by the Hash field and will not be duplicated.

Static content bundling service

Beginner

Easy

Medium

Advanced

Introduction

All custom content (e.g., the source code of custom schemas, css–styles) is generated in a special Creatio directory to improve performance. The benefits of this approach are described in a separate article – “[Client static content in the file system](#)”. However, having to process a considerable amount of files, the browser sends a large number of requests while loading the application, which significantly increases the loading time. To ensure stability, similar files are collected into bundles.

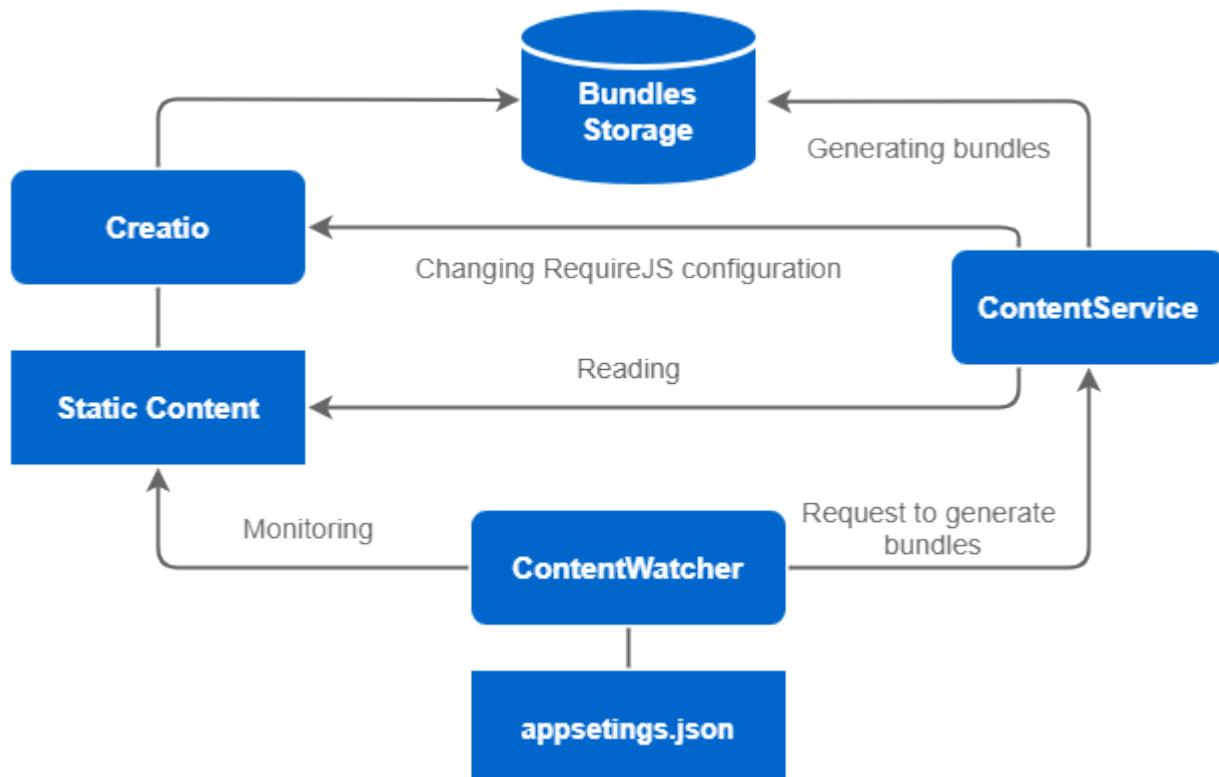
For this particular purpose, the static content bundling service is implemented in Creatio by default.

Technology

The “watcher” app (ContentWatcher) monitors the files in a static content directory and notifies the web–service about any changes. Upon request (either manual or from ContentWatcher), the web–service (ContentService) re–generates bundle–files and modifies a specific Creatio configuration file so that it uses bundles instead of static content.

Fig. 1 illustrates the basic work principles of the service.

Fig. 1. Basic work principles of the static content bundling service



A web-service can be installed without the “watcher” app (ContentWatcher). In this case, all requests to the web-service (ContentService) for bundling or [minification](#) can only be done manually.

Both service components and the Creatio application can be installed on separate computers, provided that the components have network access to static content.

ContentService

ContentService is a .NET Core 2.1 web-service which performs the following operations (access points):

- / – tests the service efficiency (the GET method).
- /process-content – generates minified bundle-files (the POST method).
- /clear-bundles – clears bundle-files (the POST method).
- /minify-content – minifies content (the POST method).

In addition to file-related actions, these operations also modify the `_ContentBootstrap.js` configuration file.

Operation parameters:

- *ContentPath* – static content path. This is a required parameter.
- *OutputPath* – the ContentService output path (local or network) for saving bundles or minified files. The default value is *ContentPath* + "/bundles".
- *SchemasBundlesRequireUrl* – a path used in the *RequireJs* configuration file for bundles or minified files. If the value of *OutputPath* is changed to anything other than the default one, this parameter must also be specified. The default value of *OutputPath* is *ContentPath* + "/bundles".
- *BundleMinLength* – a threshold for the size of a single bundle. The default value is 204800B (200KB).
- *MinifyJs* – determines if JavaScript files can be minified (*true/false*). The default value is *true*.
- *MinifyCss* – determines if CSS files can be minified (*true/false*). The default value is *true*.
- *MinifyConfigs* – determines if *RequireJs* configuration files can be minified (*true/false*). The default value is *true*.
- *ApplyBundlesForSchemas* – determines if schemas can be bundled (*true/false*). The default value is *true*.
- *ApplyBundlesForSchemasCss* – determines if CSS schemas can be bundled (*true/false*). The default value is *false*.
- *ApplyBundlesForAloneCss* – determines if separate CSS files (CSS files without a connected JavaScript-module) can be bundled (*true/false*). The default value is *false*.

- *ApplyBundlesForResources* – determines if schema resources can be bundled (*true/false*). The default value is *true*.

ContentWatcher

ContentWatcher is .NET Core 2.1 application, which is run like a service (it can also be run using .NET Core CLI Tools). The primary task of ContentWatcher is to monitor any changes in a file specified in the *fileFilter* parameter. The path to the file itself is specified in the *directory* parameter (e.g., `--directory='c:\temp' --fileFilter='readme.txt'`). By default, the *fileFilter* parameter value is `"_MetaInfo.json"`. If this file is changed, ContentWatcher considers this to be an update of all static content. When changes are detected, *ContentWatcher* notifies *ContentService* to re-generate bundle-files.

Launch parameters (specified in *appsettings.json*):

- *ContentServiceUrl* – a link to the ContentService application.
- *CloudServiceId* – service identifier for reading the AzManager database settings. This parameter is optional – it reads the settings of Creatio cloud infrastructure.
- *DefaultFileFilter* – default name of the monitored file.
- *AzManagerConnectionString* – the connection string to the AzManager database. This parameter is optional – it reads the settings of Creatio cloud infrastructure.
- *ConfigurationRefreshInterval* – service configuration update interval.
- *ContentWorkers* – an array of configuration objects that *ContentWatcher* monitors. Properties of configuration objects:
 - *name* – name for the monitored site (displayed in service logs).
 - *directory* – path to the monitored file.
 - *fileFilter* – name of the monitored file.
 - *ContentWorkerArguments* – an array of additional parameters, passed to ContentService. The elements of the array are key-value objects. Object properties:
 - *key* – the key property. “contentPath” by default.
 - *value* – path to the directory with static content files of the custom site.

An example of *appsettings.json* configuration file:

```
{
  "ContentServiceUrl": "http://localhost:9563/process-content",
  "ConfigurationRefreshInterval": "60000",
  "DefaultFileFilter": "_MetaInfo.json",
  "CloudServiceId": "151",
  "AzManagerConnectionString": "Data Source=azserver\\mssql2016;Initial Catalog=azmanager;Integrated Security=SSPI;",
  "ContentWorkers": [
    {
      "name": "My Creatio",
      "directory": "C:/Creatio/Terrasoft.WebApp/conf",
      "fileFilter": "_MetaInfo.json",
      "ContentWorkerArguments": [
        {
          "key": "contentPath",
          "value": "C:/Creatio/Terrasoft.WebApp/conf/content"
        }
      ]
    }
  ]
}
```

InfluxDb metrics

The ContentWatcher and ContentService can record successful and unsuccessful operations and their metrics in InfluxDb. To do this, specify the *InfluxDbConnectionString* string in the corresponding *appsettings.json* file with the connection parameters:

- *url* – InfluxDb server address. Required parameter.

- *db* – metrics will be recorded to this database. Optional parameter. The default value is “content”.
- *user* – username for connecting to the server. Optional parameter, blank by default.
- *password* – password for connecting to the server. Optional parameter, blank by default.
- *version* – InfluxDb server version. Available values: Latest, v_1_3, v_1_0_0, v_0_9_6, v_0_9_5, v_0_9_2, v_0_8_x. Optional parameter. The default value is “Latest”.

An example:

```
"InfluxDbConnectionString": "url=http://1.2.3.4:5678; db=content; user=User1; password=QwErTy; version=v_1_3"
```

Metrics that are written by *ContentWatcher*:

- *watcher_notifying_duration* – the duration of the ContentService notification.
- *watcher_error* – errors that occur while loading the settings, monitoring the file, or notifying *ContentService*.

Metrics that are written by *ContentService*:

- *service_processing_duration* – content processing duration.
- *service_error* – errors that occur while generating bundles, deleting temporary folders, or cleaning bundles.

Docker deployment instructions

System requirements

- A Linux OS server (stable versions of Ubuntu or Debian are recommended), with a version of docker installed and configured. The requests to the image repository (<https://hub.docker.com/>) must be allowed from this server.
- Both Docker and Docker Compose must be installed on the server (see the [Docker documentation](#)).

Service configuration structure

- *ets/content-watcher/appsettings.json* – ContentWatcher configuration file.
- *Docker-compose.yml* – docker-compose utility configuration file.
- *.env* – the file containing environment variables for running the components.

Service installations steps

1. [Download](#) the service configuration files and unzip them in a random directory (e.g., */opt/services*). The archive is available via this [link](#).
2. Specify the necessary parameters in *./docker-compose/.env*:
 - *ContentServicePort* – ContentService will run on the port, specified here.
 - *ContentPath* – sites and their contents specified here will be mounted to a container.
3. Configure the list of ContentWatcher tracking sites in *./docker-compose/etc/content-watcher/appsettings.json* (see above).
4. Execute the following command from the configuration file directory (e.g., */opt/services*):

```
sudo docker-compose pull
```

The docker-images of the services will be downloaded from hub.docker.com as a result.

If the server denies access to the Internet, manually download all necessary images on a computer with open access (see the “*docker-compose.yml*” configuration file). Use the *sudo docker export* and *sudo docker import* commands to transfer the images to the target computer as files.

5. Execute the following command from the configuration file directory (e.g., */opt/services*):

```
sudo docker-compose up -d
```

The services will start as a result.

6. To test the installation, send a GET request to: {Server IP address}: {ContentServicePort}, where:

- Server IP address – IP address of the server with installed services.
- ContentServicePort – ContentService will run on the port, specified here. The port here should match the port specified in the *./docker-compose/.env* file (see paragraph 2).

Example:

10.17.17.7:9999

Expected response:

"Service is running"

Class libraries and REST API

Contents

- [.NET class libraries of platform core \(on-line documentation\)](#)
- [JavaScript API for platform core \(on-line documentation\)](#)
- [Integrations and external API](#)

Integrations and external API

Contents

- [Introduction](#)
- [Choosing the method of integration with Creatio](#)
- [Authentication of external requests](#)
- [Integration via OData protocol](#)
- [DataService web service \(on-line documentation\)](#)
- [The ProcessEngineService.svc web service](#)
- [Integration examples with Creatio](#)

Integration with Creatio and public API

[Beginner](#)

[Easy](#)

[Medium](#)

[Advanced](#)

Creatio has a wide range of integrations with custom third-party applications.

First, accessing Creatio by a third-party application requires authentication. For more information on accessing Creatio, see "[External requests authentication to Creatio services \(on-line documentation\)](#)". For more information on Creatio primary authentication service, see "[The AuthService.svc authentication service \(on-line documentation\)](#)".

Starting with version 7.10, authentication is protected from CSRF attacks. For more information, see "[Protection from CSRF attacks during integration with Creatio \(on-line documentation\)](#)".

Brief description and comparison of Creatio basic integration methods is available in the "[Choosing the method of integration with Creatio](#)" article.

Wide range of integration capabilities is available through API provided by the DataService web service. For more information on create, read, update and delete operations (the CRUD operations), as well as the API, see the "[DataService](#)" article.

If the third-party system uses the OData protocol, it can also be used for integration with Creatio. For more information, see the "[Integration via OData protocol](#)" article.

Using the iframe HTML element for integration is covered in the "[Integration of third-party sites via iframe](#)" article.

For more information on the "Web-to-Object" integrations, see the "[Web-To-Object. Integration via landings and web-forms](#)".

Use methods of the ProcessEngineService.svc web service to trigger Creatio processes via third-party applications. Description of methods and public web service API are available in the following article: "[The ProcessEngineService.svc web service](#)".

Contents

- [Choosing the method of integration with Creatio](#)
- [External requests authentication to Creatio services \(on-line documentation\)](#)
- [The AuthService.svc authentication service \(on-line documentation\)](#)
- [Protection from CSRF attacks during integration with Creatio \(on-line documentation\)](#)
- [DataService](#)
- [Integration via OData protocol](#)

- **Integration of third-party sites via iframe**
- **Web-To-Object. Integration via landings and web-forms**
- **The ProcessEngineService.svc web service**

Choosing the method of integration with Creatio

Beginner **Easy** **Medium** **Advanced**

Introduction

Creatio enables range of methods for integration with third-party software products. Choosing the method of integration depends on the needs of the client, the type and architecture of third-party software products and the developer's skills. A comparison of the main characteristics of the supported methods of integration with Creatio is given in Table 1.

Table 1. Comparison of main methods of integration with Creatio

DataService	OData	Configuration service	Web-to-Object	iframe
<i>Supported formats of the data exchange</i>				
XML, JSON, JSV, CSV	XML, JSON	XML, JSON	JSON	No
<i>Tasks are being solved</i>				
CRUD operations with Creatio objects, data filtering and use of built-in Creatio macros	CRUD-operations with objects, adding and removing links, obtaining metadata, collections, object fields, sorting, etc.	All user tasks that can be solved within the open Creatio API	Only adding objects	Only interaction with user interface of the integrated third-party web application (web page).
<i>Complexity</i>				
High	Medium	Medium	Medium	Low
<i>Ways of authentication</i>				
Forms	Basic, Forms	Anonymous, Forms — depends on service implementation.	Forms	Forms (with Creatio)
<i>Availability of auxiliary custom libraries</i>				
Creatio .dll libraries can be used only for .NET applications	Enabled http://www.odata.org/libraries	No need	No need	No need
<i>Developer</i>				

A brief description and the main advantages and disadvantages of each of the methods are given below.

Integration with DataService

The DataService web service is the main link between the Creatio client and server parts. It helps to transfer the data that were entered by user via user interface to server side of the application for further processing and saving to the database. More information about the DataService web service can be found in the “**DataService web service**” article.

Key benefits and integration options

- Data exchange with XML, JSON, JSV, CSV.
- Available operations of **creating, reading, updating** and **deleting** the Creatio objects (CRUD operations). You can use **built-in macros** and **data filtering**. **Batch processing** is available for complex queries.
- User authorization is required for access.

Disadvantages

- High complexity of query building.
- Required in-depth knowledge for development.
- Auxiliary libraries for popular application and mobile platforms are disabled.

Example

An example of the source code of a simple application for sending adding data request to the DataService web service is given below. A request is made to create a new contact, for which the main columns are filled. Detail description of this example can be found in the “**DataService. Adding records**”

```
using System;
using System.Text;
using System.IO;
using System.Net;
using System.Collections.Generic;
using Terrasoft.Nui.ServiceModel.DataContract;
using Terrasoft.Core.Entities;
using System.Web.Script.Serialization;

namespace DataServiceInsertExample
{
    class Program
    {
        private const string baseUri = @"http://userapp.creatio.com";
        private const string authServiceUri = baseUri +
@"/ServiceModel/AuthService.svc/Login";
        private const string insertQueryUri = baseUri +
@"/0/DataService/json/reply/InsertQuery";
        private static CookieContainer AuthCookie = new CookieContainer();

        private static bool TryLogin(string userName, string userPassword)
        {
            bool result = false;
            // TODO: Implementation of authentication.
            return result;
        }

        static void Main(string[] args)
        {
            if (!TryLogin("User1", "User1"))
```

```
{  
    return;  
}  
// Generate a JSON object for requesting the addition of data.  
// Use the InsertQuery data contract.  
var insertQuery = new InsertQuery()  
{  
    RootSchemaName = "Contact",  
    ColumnValues = new ColumnValues()  
};  
var columnExpressionName = new ColumnExpression  
{  
    ExpressionType = EntitySchemaQueryExpressionType.Parameter,  
    Parameter = new Parameter  
    {  
        Value = "John Smith",  
        DataValueType = DataValueType.Text  
    }  
};  
var columnExpressionPhone = new ColumnExpression  
{  
    ExpressionType = EntitySchemaQueryExpressionType.Parameter,  
    Parameter = new Parameter  
    {  
        Value = "+12 345 678 00 00",  
        DataValueType = DataValueType.Text  
    }  
};  
var columnExpressionJob = new ColumnExpression  
{  
    ExpressionType = EntitySchemaQueryExpressionType.Parameter,  
    Parameter = new Parameter  
    {  
        Value = "11D68189-CED6-DF11-9B2A-001D60E938C6",  
        DataValueType = DataValueType.Guid  
    }  
};  
insertQuery.ColumnValues.Items = new Dictionary<string, ColumnExpression>  
();  
insertQuery.ColumnValues.Items.Add("Name", columnExpressionName);  
insertQuery.ColumnValues.Items.Add("Phone", columnExpressionPhone);  
insertQuery.ColumnValues.Items.Add("Job", columnExpressionJob);  
var json = new JavaScriptSerializer().Serialize(insertQuery);  
byte[] jsonArray = Encoding.UTF8.GetBytes(json);  
  
// Sending an Http-request.  
var insertRequest = HttpWebRequest.Create(insertQueryUri) as  
HttpWebRequest;  
insertRequest.Method = "POST";  
insertRequest.ContentType = "application/json";  
insertRequest.CookieContainer = AuthCookie;  
insertRequest.ContentLength = jsonArray.Length;  
using (var requestStream = insertRequest.GetRequestStream())  
{  
    requestStream.Write(jsonArray, 0, jsonArray.Length);  
}  
using (var response = (HttpWebResponse)insertRequest.GetResponse())  
{  
    using (StreamReader reader = new  
StreamReader(response.GetResponseStream()))  
    {  
        Console.WriteLine(reader.ReadToEnd());  
    }  
}
```

```
        }
    }
}
}
```

Integration with OData

[Open Data \(OData\)](#) protocol is an open web protocol for data request and update based on the REST architectural approach using Atom/XML and JSON standards. Any third-party application that supports HTTP messaging and can process XML or JSON data can have the access to the Creatio data and objects. Data is available as resources addressed through a URI. Access to the data and modification are performed by standard HTTP commands (GET, PUT/PATCH, POST and DELETE). More information about OData protocol can be found in the “**Creatio integration over the OData protocol (on-line documentation)**” article.

Key benefits and integration options

- Data exchange with XML, JSON.
- A lot of operations with Creatio objects, including CRUD operations.
- Convenient functions for working with strings, dates and time.
- A large number of custom libraries for working with OData for popular application and mobile platforms.
- User authorization is required for access.

More information about OData integration can be found in the “**Creatio integration over the OData protocol (on-line documentation)**” article.

Disadvantages

- Complexity of query building.
- Requires advanced skills for development.

Case example

Example of method source code for adding a new contact record using OData is given below. Detail description of this example can be found in the “**Working with Creatio objects over the OData protocol using Http request (on-line documentation)**” article.

```
// POST <Creatio URL>/0/ServiceModel/EntityDataService.svc/ContactCollection/
public static void CreateCreatioEntityByOdataHttpExample()
{
    // Create an xml message that contains information about the object being
    created.
    var content = new XElement(dsmd + "properties",
        new XElement(ds + "Name", "John Smith"),
        new XElement(ds + "Dear", "John"));
    var entry = new XElement(atom + "entry",
        new XElement(atom + "content",
            new XAttribute("type", "application/xml"), content));
    Console.WriteLine(entry.ToString());
    // Create a service request that will add a new object to the collection of
    contacts.
    var request = (HttpWebRequest)HttpWebRequest.Create(serverUri +
    "ContactCollection/");
    request.Credentials = new NetworkCredential("CreatioUserName",
    "CreatioUserPassword");
    request.Method = "POST";
    request.Accept = "application/atom+xml";
    request.ContentType = "application/atom+xml;type=entry";
    // Write an xml message to the request thread.
    using (var writer = XmlWriter.Create(request.GetRequestStream()))

```

```
{  
    entry.WriteTo(writer);  
}  
// Receiving a response from the service about the result of the operation.  
using (WebResponse response = request.GetResponse())  
{  
    if (((HttpWebResponse)response).StatusCode == HttpStatusCode.Created)  
    {  
        // Processing the result of the operation.  
    }  
}  
}  
}
```

Integration with custom configuration web service

Creatio enables to create custom web services in the configuration that can implement specific integration tasks. Configuration web service is a RESTful service implemented on the basis on WCF technology. More information about creation of custom configuration web service can be found in the “[Creating a user configuration service](#)” article.

Key benefits and integration options

- Data exchange is implemented by developer in any convenient way.
- Developer can implement any operation with Creatio objects, including CRUD operations.
- User authorization is not required for access.

Disadvantages

- The entire functionality of the service need to be developed.
- Required in-depth knowledge for development.

Case example

The complete source code of the configuration service is available below: Service adds the "changed!" word to the incoming parameter and sends a new value in the HTTP answer. Detail description of this example can be found in the “[Creating a user configuration service](#)” article.

```
namespace Terrasoft.Configuration.CustomConfigurationService  
{  
    using System;  
    using System.ServiceModel;  
    using System.ServiceModel.Web;  
    using System.ServiceModel.Activation;  
  
    [ServiceContract]  
    [AspNetCompatibilityRequirements(RequirementsMode =  
AspNetCompatibilityRequirementsMode.Required)]  
    public class CustomConfigurationService  
    {  
        [OperationContract]  
        [WebInvoke(Method = "POST", RequestFormat = WebMessageFormat.Json, BodyStyle = WebMessageBodyStyle.Wrapped,  
ResponseFormat = WebMessageFormat.Json)]  
        public string GetTransformValue(string inputParam)  
        {  
            // Change the value of the incoming parameter.  
            var result = inputParam + " changed!";  
            return result;  
        }  
    }  
}
```

Integration with Web-To-Object mechanism.

Web-to-Object is a mechanism of implementation of simple one-way integrations with Creatio. It enables you to create records of the Creatio sections (leads, cases, orders, etc.) by sending required data to the Web-to-Object service.

More common cases of using Web-to-Object service:

- Creatio integration with custom landings and web forms. The service call is executed from a specifically configured user web page (landing) after the visitor sends the data of the filled form.
- Integration with external systems that are involved in creation of Creatio objects.

More information about the Web-To-Object can be found in the **“Web-To-Object. Integration via landings and web-forms”**

Key benefits and integration options

- Data transfer with JSON.
- Easy creation of Creatio objects.
- To access you need only URL of the service and Id.
- Required only basic knowledge for development.

Disadvantages

- Data transfer only in the Creatio.
- Limited number of objects to use. Service needs to be modified to use custom objects.

Case example

To use service, send the POST query by the address:

```
[Path to Creatio application]/0/ServiceModel/GeneratedObjectWebFormService.svc/SaveWebFormObjectData
```

Query content type – application/json. Insert required cookies and a JSON object that contains data of the web form, to the query content. Example of the JSON object that contains data for creation of a new contact:

```
{
    "formData": {
        "formId": "d66ebbf6-f588-4130-9f0b-0ba3414dafb8",
        "formFieldsData": [
            { "name": "Name", "value": "John Smith" },
            { "name": "Email", "value": "j.smith@creatio.com" },
            { "name": "Zip", "value": "00000" },
            { "name": "MobilePhone", "value": "0123456789" },
            { "name": "Company", "value": "Creatio" },
            { "name": "Industry", "value": "" },
            { "name": "FullJobTitle", "value": "Sales Manager" },
            { "name": "UseEmail", "value": "" },
            { "name": "City", "value": "Boston" },
            { "name": "Country", "value": "USA" },
            { "name": "Commentary", "value": "" },
            { "name": "BpmHref", "value": "http://localhost/Landing/" },
            { "name": "BpmSessionId", "value": "0ca32d6d-5d60-9444-ec34-5591514b27a3" }
        ]
    }
}
```

Integration of third-party sites via iframe

The most simple way to integrate external solutions to Creatio. The third-party web application can be implemented

to Creatio with the iframe HTML element. This enables to view third-party web resources (web pages, video, etc.) from Creatio. Examples of integration via iframe can be found in the “[Integration of third-party sites via iframe](#)” and “[Developing an advanced marketplace application](#)” ([Marketplace development documentation](#)).

Key benefits and integration options

- Convenience of viewing the third-party web resources directly from the Creatio.
- Requires only basic skills for development.

Disadvantages

- Needs to be modified for data transfer (or use another integration method).
- Some sites prohibit uploading of their pages into the iframe element.

Case example

An example of the source code of the view model schema of Creatio edit page with the implemented iframe element is given below. On the page, the iframe element displays a web site which URL is specified in the object associated with the page. Detail description of this example can be found in the “[Developing an advanced marketplace application](#)” article. Another approach is described in the “[Integration of third-party sites via iframe](#)” article.

```
define("tsaWebData1Page", [], function() {
    return {
        entitySchemaName: "tsaWebData",
        diff: /**SCHEMA_DIFF*/ [
            // ...
            // A container with an embedded HTML iframe element.
            {
                "operation": "insert",
                "name": "IFrameStat",
                "parentName": "TabData",
                "propertyName": "items",
                "values": {
                    "id": "testiframe",
                    "itemType": Terrasoft.ViewItemType.CONTAINER,
                    "selectors": {"wrapEl": "#stat-iframe"},
                    "layout": { "colSpan": 24, "rowSpan": 1, "column": 0, "row": 0 },
                    "html": "<iframe id='stat-iframe' class='stat-iframe' width='100%' height='550px' + style = 'border: 1px solid silver;'></iframe>",
                    "afterrerender": {
                        "bindTo": "addUrl"
                    }
                }
            }
        ] /**SCHEMA_DIFF*/,
        methods: {
            // The event handler for the full data load.
            onEntityInitialized: function() {
                this.callParent(arguments);
                this.addUrl();
            },
            // The method of adding a URL to an HTML iframe element.
            addUrl: function() {
                var iframe = Ext.get("stat-iframe");
                if (!iframe) {
                    window.console.error("The tab with iframe element was not found");
                    return;
                }
                var siteName = this.get("tsaName");
```

```
        if (!siteName) {
            window.console.error("The website name was not provided");
            return;
        }
        var url = "https://www.similarweb.com/website/" + siteName;
        this.set("tsaURL", url);
        iframe.dom.src = url;
    }
}
);
});
```

Authentication of external requests

Beginner

Easy

Medium

Advanced

Contents

- **Introduction**
- **AuthService.svc request**
- **AuthService.svc response**
- **Example of a software implementation of the authentication**
- **Disabling the CSRF protection**

Introduction

All external requests to Creatio web services must be authenticated.

We have the following authentication types in Creatio:

- Anonymous authentication
- Basic authentication
- Forms authentication (Cookie based)

The “Forms authentication” is a best practice for integration with the application.

To perform the “Forms authentication”, Creatio uses a separate “AuthService.svc” web service.

Starting with version 7.10, authentication is protected from CSRF attacks.

CSRF (Cross Site Request Forgery) – is a type of attacks for web site visitors based on possible HTTP problems.

You need to perform authentication using the “AuthService.svc” service when integrating with third party applications. After you successfully authenticate, AuthService returns the authentic cookie that must be added to your request. It also returns a cookie containing the CSRF token. This token must be placed in the request header. You can find examples of an authentic cookie below and in the "**Integration via OData protocol**" and "**DataService ('DataService web service' in the on-line documentation)**" articles.

AuthService.svc request

Example of an authentication HTTP request

```
POST /ServiceModel/AuthService.svc/Login HTTP/1.1
Host: http://myserver.com/CreatioWebApp
Content-Type: application/json
ForceUseSession: true
{
    "UserName": "Supervisor",
    "UserPassword": "Supervisor"
}
```

Web service request structure:

- query string;
- request headers;
- request body.

Query string

To perform authentication, call the “Login” AuthService.svc. method.

Query string structure

```
http(s)://[Creatio application address]/ServiceModel/AuthService.svc/Login
```

Example of a query string

```
https://myserver.com/CreatioWebApp/ServiceModel/AuthService.svc/Login
```

Request headers

Content-Type

Content-Type: application/json

Encoding and resource type passed in the request body.

ForceUseSession

ForceUseSession: true

The “ForceUseSession” header accounts for using the existing session.

Request body

The request body must pass the Creatio user credentials. The credentials are passed as a JSON object.

Properties of the JSON object with credentials

UserName

The user name of a Creatio user.

UserPassword

The password of a Creatio user.

Example of a request body:

```
{  
    "UserName": "Supervisor",  
    "UserPassword": "Supervisor"  
}
```

AuthService.svc response

Structural elements of the request response:

- HTTP status code;
- response headers;
- response body.

HTTP status code

As a request response, the server returns an HTTP status code encrypted in 3 digits. The first digit specifies the status class. The second and the third digits determine the response ordinal number.

When executing the authentication request, the server returns the following status codes:

200 OK The request has been completed successfully and the resource value is not equal to zero. In this case, the

request body should contain the authentication status code. If it contains 0, the authentication is successful. In case of unsuccessful authentication, the authentication status code will equal 1 and the request body will contain a message about the cause of the unsuccessful authentication.

403 Forbidden The server cannot provide access to the resource specified in the request (for example, if a method name is spelled incorrectly). Request body can contain additional information.

Response headers

The response to a POST request contains authentication cookies. You need to save these cookies on the side of the client or on the client computer to use them in your further Creatio web service queries.

Response body

Example of an authentication service response

```
{  
    "Code": 0,  
    "Message": "",  
    "Exception": null,  
    "PasswordChangeUrl": null,  
    "RedirectUrl": null  
}
```

The response body will contain a JSON object of the authentication status.

The primary properties of the JSON object:

Code

Authentication status code

If the code contains a “0” value, the authentication is successful. Otherwise, it is failed.

Message

The message notifying of an unsuccessful authentication.

Exception

The object that contains a detailed description of the exception connected with the unsuccessful authentication.

Example of a software implementation of the authentication

Create a C# console application in Visual Studio and give it a name, e.g., RequestAuthentication. You can download the full code with case implementation using the following [link](#).

Example of a software implementation of the authentication

```
// Sends a request to the authentication service and processes the  
// response.  
public void TryLogin() {  
    var authData = @"  
        ""UserName"":"""+_userName + @"""",  
        ""UserPassword"":"""+_userPassword + @"""  
    ";  
    var request = CreateRequest(_authServiceUrl, authData);  
    _authCookie = new CookieContainer();  
    request.CookieContainer = _authCookie;  
    // Upon successful authentication, we save authentication cookies  
    for  
        // further use in requests to Creatio. In case of failure
```

```
// authentication application console displays a message about the
reason
// of the mistake.
using (var response = (HttpWebResponse)request.GetResponse())
{
    if (response.StatusCode == HttpStatusCode.OK)
    {
        using (var reader = new
StreamReader(response.GetResponseStream()))
        {
            var responseMessage = reader.ReadToEnd();
            Console.WriteLine(responseMessage);
            if (responseMessage.Contains("\"Code\":1"))
            {
                throw new
UnauthorizedAccessException($"Unauthorized {_userName} for {_appUrl}");
            }
        }
        string authName = ".ASPxAUTH";
        string authCookieValue = response.Cookies[authName].Value;
        _authCookie.Add(new Uri(_appUrl), new Cookie(authName,
authCookieValue));
    }
}
// Create request to the authentication service.
private HttpWebRequest CreateRequest(string url, string requestData =
null)
{
    HttpWebRequest request =
(HttpWebRequest)WebRequest.Create(url);
    request.ContentType = "application/json";
    request.Method = "POST";
    request.KeepAlive = true;
    if (!string.IsNullOrEmpty(requestData))
    {
        using (var requestStream = request.GetRequestStream())
        {
            using (var writer = new
StreamWriter(requestStream))
            {
                writer.Write(requestData);
            }
        }
    }
    return request;
}
// Method realizes protection from CSRF attacks: copies cookie, which
contents CSRF-token
// and pass it to the header of the next request.
private void AddCsrfToken(HttpWebRequest request) {
```

```
var cookie = request.CookieContainer.GetCookies(new Uri(_appUrl))
["BPMCSRF"];
if (cookie != null) {
    request.Headers.Add("BPMCSRF", cookie.Value);
}
}
```

Disabling the CSRF protection

We recommend disabling the CSRF protection only if you use basic authentication.

In Creatio, you can disable the CSRF protection both for the whole application or for separate services and methods.

1. Disabling the CSRF protection for all application services

In the .\Web.Config and .\Terrasoft.WebApp\Web.Config files, disable the UseCsrfToken setting:

```
<add value="false" key="UseCsrfToken" />
```

2. Disabling the CSRF protection for one application service

Use the DisableCsrfTokenValidationForPaths setting in the .\Web.Config file:

```
<add key="DisableCsrfTokenValidationForPaths" value="/ServiceModel/
MsgUtilService.svc" />
```

3. Disabling the CSRF protection for several methods of different services

You can do it using the same DisableCsrfTokenValidationForPaths setting in the .\Web.Config file.

```
<add key="DisableCsrfTokenValidationForPaths"
value="/MsgUtilService.svc/Ping,/AuthService.svc/Login" />
```

Integration via OData protocol

Contents

- **Creatio integration via the OData 3 protocol**
- **Creatio integration via the OData 4 protocol**

Creatio integration via the OData 3 protocol

- **Introduction**
- **Implementation of OData 3 in Creatio**
- **OData 3 request structure**
- **Request response structure**
- **Request types**

Introduction

OData (Open Data Protocol) is an ISO/IEC confirmed OASIS standard determining the best practices for building and using REST API. It enables creating REST-based services that allow web clients to publish and edit resources using simple HTTP requests. Such resources should be identified with a URL and defined in the data model.

Creatio supports OData 3 and OData 4 protocols. OData 4 replaces OData 3 and considerably enhances the capacity of the previous protocol. These two protocols are not compatible by data format returned by the server. You can learn more about the protocols in [Odata documentation](#).

To integrate with Creatio, use OData 4. Using the OData 4 protocol is covered in the “[Creatio integration via the OData 4 protocol](#)” article.

You need to **authenticate** your requests to Creatio.

Implementation of OData 3 in Creatio

The *EntityDataService.svc* web service will provide access to Creatio objects via the OData 3 protocol.

EntityDataService.svc server address:

`https://mycreatio.com/0/ServiceModel/EntityDataService.svc`

OData 3 request structure

Structural elements of the request:

- String
- Headers
- Body

Query string

Query string structure:

`method my_Creatio_site/0/ServiceModel/EntityDataService.svc/data_resource?$parameters`

Example of a POST request string

```
// Add an instance of the "AcademyURL" collection object.  
POST  
https://mycreatio.com/0/ServiceModel/EntityDataService.svc/AcademyURLCollection
```

Example of a GET request string

```
// Receive an instance of the "Employee" collection objects, whose  
"Full Job Title" field contains the "Developer" value. The "Name" field  
value of the nested "Account" object collection should be other than  
"Our company".  
GET  
https://mycreatio.com/0/ServiceModel/EntityDataService.svc/EmployeeCollection?$filter=FullJobTitle eq 'Developer' and Account/Name ne 'Our  
company'
```

method

Creatio supports the following request methods:

- GET – receiving data.
- POST – adding data.
- PATCH – modifying data.
- DELETE – deleting data.

data_resource

Data resource is the path to the source of information obtained in the request response.

Data resource structure:

`objects_collectionCollection(guid'object_id')/object_field`

When integrating with Creatio via OData 3, add *Collection* to the first name of the collection objects in the request string (e.g., *ContactCollection*). Use the `guid'object_id'` construction to specify the identifier of the collection object instance (e.g., `guid'oooooooo-oooo-oooo-oooo-oooooooooooo'`).

Structural elements of the data resource:

objects_collection (required)

The name of the database table (name of the object collection).

You can receive the list of database tables if you run the following query:

For MySQL

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
```

For Oracle

```
SELECT * FROM ALL_TABLES
```

For PostgreSQL

```
SELECT table_name FROM information_schema.tables
```

object_id (optional)

The string identifier of the database table record (identifier of the collection object instance).

object_field (optional)

The database table record field (field of the collection object instance).

parameters

Non required OData 3 parameters that you can use in the Creatio GET request string:

\$value

Field value.

\$count

`$count`

The number of elements in the build.

\$skip

`$skip=n`

N of the first elements that should not be in the build.

\$top

`$top=n`

N of the first elements that should be in the build.

\$select

`$select=field1,field2,...`

List of fields that should be in the build.

\$orderby

`$orderby=field asc`

`$orderby=field desc`

Sorting of field values that are in the build.

\$expand

`$expand=field1,field2,...`

Extension of the connected fields.

\$filter

`$filter=field template 'field_value'`

Filtering of fields that should be in the build.

You can use the following operators when working with parameters:

`?`

Specifying the parameter.

`$`

Specifying the parameter name.

`&`

Using 2 or more parameters.

Request header

Headers that you can use in the requests to Creatio:

Accept

`Accept: application/atom+xml; type=entry`

Possible data type of the server response. The server returns its response in the XML format.

Content-Type

`Content-Type: application/json; odata=verbose`

Encoding and resource type passed in the request body.

ForceUseSession

`ForceUseSession: true`

The “ForceUseSession” header accounts for using the existing session. You do not need using AuthService.svc in your request to the authentication service.

BPMCSRF

`BPMCSRF: authentication_cookie_value`

Authentication cookie

The “Content-Type” and “Accept” headers are not required for the GET requests.

Example of POST request headers

```
Accept: application/atom+xml; type=entry
Content-Type: application/json; odata=verbose
ForceUseSession: true
BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO
```

Example of GET request headers

```
ForceUseSession: true
BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO
```

Request body

Request body structure:

```
{  
    "field1": "value1",  
    "field2": "value2",  
    ...  
}
```

Structural elements of the request body:

field1, field2, ... (required)

Field names that are passed in the request body.

value1, value2, ... (required)

field1, field2, ...values that are passed in the request body.

Example of a POST request body

```
{  
    // Add the Academy value in the Name field.  
    "Name": "Academy",  
    // Add https://academy.creatio.com/ value in the Description field.  
    "Description": "https://academy.creatio.com/"  
}
```

Request response structure

Structural elements of the query response:

- Status code
- Body

Status code

As a request response, the server returns a status code encrypted in 3 digits. The first digit specifies the status class. The second and the third digits determine the response ordinal number. The server returns the following HTTP code classes when integrating with Creatio via the OData 3 protocol:

1. 2xx (success) – successful acceptance and processing of the client request.
2. 3xx (redirection) – you need to execute another request to successfully fulfill the operation.
3. 4xx (client error) – a request building error on the side of the client. The server must return a hypertext explanation of receiving the 4xx class code.

Receiving the 4xx class code is a result of the unsuccessful request execution.

You can receive the following status codes from the server when sending requests via the OData 3 protocol:

200 OK The GET, PUT, MERGE or PATCH request has been completed successfully. The response body must contain the value of the object or property specified in the request URL.

201 Created The POST request has successfully created an object or a link. The response body must contain the updated object.

202 Accepted The request to modify data has been accepted but has not yet finished. The response body must contain [header Location](#) in addition to [header Retry-After](#). The response body must be empty. The server must return the 303 response code with [header Location](#), containing the final URL to receive the request result. The body and the header of the final URL must be formatted similar to the previous request for modifying data.

204 No Content Request to modify data. The requested resource has a zero value. The response body must be empty.

3xx Redirection Request to modify data. The redirection means that the client needs to take further actions to

execute the request. The response must contain [header Location](#) with URL that enables receiving the result.

4xx Client Error Incorrect requests. The server returns a code in response to client errors and requests to nonexistent resources, such as entities, entity collections or properties. If the response body is defined for the error code, the error body is as defined for the corresponding [format](#).

404 Not Found Object or collection specified in the URL does not exist. The response body must be empty.

Response body

Response body structure:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xml:base="http://mycreatio.com/0/ServiceModel/EntityDataService.svc/"
      xmlns="http://www.w3.org/2005/Atom"
      xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
      xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
    <id>http://mycreatio.com/0/ServiceModel/EntityDataService.svc/data_resource</id>
    <title type="text">data_resource</title>
    <updated>date and time of request</updated>
    <link rel="self" title="data_resource" href="data_resource" />
    <entry>
        metadata_data
        <content type="application/xml">
            <m:properties>
                <d:object1 field1>object1 field_value1</d:object1 field1>
                <d:object1 field2>object1 field_value2</d:object1 field2>
                ...
                </m:properties>
            </content>
        </entry>
        <entry>
            metadata_data
            <content type="application/xml">
                <m:properties>
                    <d:object2 field1>object2 field_value1</d:object2 field1>
                    <d:object2 field2>object2 field_value2</d:object2 field2>
                    ...
                    </m:properties>
                </content>
            </entry>
            ...
    </feed>
```

Structural elements of the response body:

entry

Instance of the collection object.

metadata_data

Metadata of the collection object instance.

object1 field1, object1 field2, ..., object2 field1, object2 field2, ...

Names of the field1, field2, ... fields of the object1, object2, ...collection object instances.

object1 field_value1, object1 field_value2, ..., object2 field_value1, object2 field_value2, ...

Values of the field1, field2, ... fields of the object1, object2, ...collection object instances.

Example of a response to a POST request

Status: 201 Created

```
<?xml version="1.0" encoding="utf-8"?>
<entry
  xml:base="http://mycreatio.com/0/ServiceModel/EntityDataService.svc/"
  xmlns="http://www.w3.org/2005/Atom"
  xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
  xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
>

<id>http://mycreatio.com/0/ServiceModel/EntityDataService.svc/AcademyUR
LCollection(guid'b634e2d0-6baf-4a13-b9e5-869b717f6406')</id>
  <category term="Terrasoft.Configuration.AcademyURL"
  scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme"
/>
  <link rel="edit" title="AcademyURL"
  href="AcademyURLCollection(guid'b634e2d0-6baf-4a13-b9e5-869b717f6406')"
/>
  <link
    rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Crea
    tedBy" type="application/atom+xml;type=entry" title="CreatedBy"
    href="AcademyURLCollection(guid'b634e2d0-6baf-4a13-b9e5-
    869b717f6406')/CreatedBy" />
  <link
    rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Modi
    fiedBy" type="application/atom+xml;type=entry" title="ModifiedBy"
    href="AcademyURLCollection(guid'b634e2d0-6baf-4a13-b9e5-
    869b717f6406')/ModifiedBy" />
  <title />
  <updated>2020-04-01T05:46:43Z</updated>
  <author>
    <name />
  </author>
  <content type="application/xml">
    <m:properties>
      <d:Id m:type="Edm.Guid">b634e2d0-6baf-4a13-b9e5-
      869b717f6406</d:Id>
        <d:Name>Academy</d:Name>
        <d:CreatedOn m:type="Edm.DateTime">2020-04-
      01T05:46:41.7819089Z</d:CreatedOn>
        <d:CreatedBy m:type="Edm.Guid">410006e1-ca4e-4502-a9ec-
      e54d922d2c00</d:CreatedBy>
        <d:ModifiedOn m:type="Edm.DateTime">2020-04-
      01T05:46:41.7819089Z</d:ModifiedOn>
        <d:ModifiedById m:type="Edm.Guid">410006e1-ca4e-4502-a9ec-
      e54d922d2c00</d:ModifiedById>
        <d:ProcessListeners
      m:type="Edm.Int32">0</d:ProcessListeners>
        <d>Description>https://academy.creatio.com/</d>Description>
      </m:properties>
    </content>
  </entry>
```

Example of a response to a GET request

Status: 200 OK

```
<?xml version="1.0" encoding="utf-8"?>
<feed
  xml:base="http://mycreatio.com/0/ServiceModel/EntityDataService.svc/"
  xmlns="http://www.w3.org/2005/Atom"
  xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
  xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
>

<id>http://mycreatio.com/0/ServiceModel/EntityDataService.svc/EmployeeC
ollection</id>
  <title type="text">EmployeeCollection</title>
  <updated>2020-03-31T06:32:52Z</updated>
  <link rel="self" title="EmployeeCollection"
    href="EmployeeCollection" />
  <entry>

<id>http://mycreatio.com/0/ServiceModel/EntityDataService.svc/EmployeeC
ollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')</id>
    <category term="Terrasoft.Configuration.Employee"
      scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme"
    />
    <link rel="edit" title="Employee"
      href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')"
    />
    <link
      rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Crea
      tedBy" type="application/atom+xml;type=entry" title="CreatedBy"
      href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-
      0943fa08fd02')/CreatedBy" />
    <link
      rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Modi
      fiedBy" type="application/atom+xml;type=entry" title="ModifiedBy"
      href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-
      0943fa08fd02')/ModifiedBy" />
    <link
      rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Cont
      act" type="application/atom+xml;type=entry" title="Contact"
      href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-
      0943fa08fd02')/Contact" />
    <link
      rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Orgs
      tructureUnitCollectionByHead" type="application/atom+xml;type=feed"
      title="OrgStructureUnitCollectionByHead"
      href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-
      0943fa08fd02')/OrgStructureUnitCollectionByHead" />
    <link
      rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Orgs
      tructureUnit" type="application/atom+xml;type=entry"
```

```
title="OrgStructureUnit" href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/OrgStructureUnit" />
    <link
        rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Job"
        type="application/atom+xml;type=entry" title="Job"
        href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/Job" />
    <link
        rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Owner"
        type="application/atom+xml;type=entry" title="Owner"
        href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/Owner" />
    <link
        rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/ReasonForDismissal"
        type="application/atom+xml;type=entry" title="ReasonForDismissal"
        href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/ReasonForDismissal" />
    <link
        rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Account"
        type="application/atom+xml;type=entry" title="Account"
        href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/Account" />
    <link
        rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Manager"
        type="application/atom+xml;type=entry" title="Manager"
        href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/Manager" />
    <link
        rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/EmployeeCollectionByManager"
        type="application/atom+xml;type=feed" title="EmployeeCollectionByManager"
        href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/EmployeeCollectionByManager" />
    <link
        rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/EmployeeCareerCollectionByEmployee"
        type="application/atom+xml;type=feed" title="EmployeeCareerCollectionByEmployee"
        href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/EmployeeCareerCollectionByEmployee" />
    <link
        rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/EmployeeFileCollectionByEmployee"
        type="application/atom+xml;type=feed" title="EmployeeFileCollectionByEmployee"
        href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/EmployeeFileCollectionByEmployee" />
    <link
        rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/EmployeeInFolderCollectionByEmployee"
        type="application/atom+xml;type=feed" title="EmployeeInFolderCollectionByEmployee"
        href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/EmployeeInFolderCollectionByEmployee" />
```

```
<link
rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/EmployeeInTagCollectionByEntity" type="application/atom+xml;type=feed"
title="EmployeeInTagCollectionByEntity"
href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/EmployeeInTagCollectionByEntity" />
<link
rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/SalaryCollectionByEmployee" type="application/atom+xml;type=feed"
title="SalaryCollectionByEmployee"
href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/SalaryCollectionByEmployee" />
<link
rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/VwEmployeesHierarchyCollectionByEmployee"
type="application/atom+xml;type=feed"
title="VwEmployeesHierarchyCollectionByEmployee"
href="EmployeeCollection(guid'c31c7862-fe33-4a13-9bbc-0943fa08fd02')/VwEmployeesHierarchyCollectionByEmployee" />
<title />
<updated>2020-03-31T06:32:52Z</updated>
<author>
<name />
</author>
<content type="application/xml">
<m:properties>
<d:Id m:type="Edm.Guid">c31c7862-fe33-4a13-9bbc-0943fa08fd02</d:Id>
<d:Name>William Walker</d:Name>
<d:CreatedOn m:type="Edm.DateTime">2017-03-30T14:50:04Z</d:CreatedOn>
<d:CreatedBy m:type="Edm.Guid">76929f8c-7e15-4c64-bdb0-adc62d383727</d:CreatedBy>
<d:ModifiedOn m:type="Edm.DateTime">2020-02-14T06:30:46.234Z</d:ModifiedOn>
<d:ModifiedBy m:type="Edm.Guid">410006e1-ca4e-4502-a9ec-e54d922d2c00</d:ModifiedBy>
<d:ProcessListeners
m:type="Edm.Int32">0</d:ProcessListeners>
<d>ContactId m:type="Edm.Guid">227aab3b-7c0c-4181-abf9-81585563ab23</d>ContactId>
<d:OrgStructureUnitId m:type="Edm.Guid">d436a9ce-9690-4415-9e03-e8061d7cab5</d:OrgStructureUnitId>
<d:Notes></d:Notes>
<d:JobId m:type="Edm.Guid">11d68189-ced6-df11-9b2a-001d60e938c6</d:JobId>
<d:FullJobTitle>Developer</d:FullJobTitle>
<d:OwnerId m:type="Edm.Guid">76929f8c-7e15-4c64-bdb0-adc62d383727</d:OwnerId>
<d:CareerStartDate m:type="Edm.DateTime">2019-09-08T00:00:00</d:CareerStartDate>
```

```
<d:CareerDueDate m:type="Edm.DateTime">0001-01-01T00:00:00</d:CareerDueDate>
<d:ProbationDueDate m:type="Edm.DateTime">2020-01-09T00:00:00</d:ProbationDueDate>
<d:ReasonForDismissalId m:type="Edm.Guid">00000000-0000-0000-000000000000</d:ReasonForDismissalId>
<d:AccountId m:type="Edm.Guid">a0bf3e92-f36b-1410-0499-00155d043204</d:AccountId>
<d:ManagerId m:type="Edm.Guid">3e5bd47e-1ebd-41db-a9a6-a3560dcee3cb</d:ManagerId>
</m:properties>
</content>
</entry>
</feed>
```

Request types

Reading data

Primary rules of GET requests

- Using parameters is allowed.
- Request body is not available.
- Response body is available.

GET request structure:

```
GET my_Creatio_site/0/ServiceModel/EntityDataService.svc/data_resource?$parameters
ForceUseSession: true
BPMCSRFAuthentication_cookie_value
```

Structure of a response body to a GET request

```
<?xml version="1.0" encoding="utf-8"?>
<feed xml:base="http://mycreatio.com/0/ServiceModel/EntityDataService.svc/" xmlns="http://www.w3.org/2005/Atom"
      xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
      xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
    <id>http://mycreatio.com/0/ServiceModel/EntityDataService.svc/data_resource</id>
    <title type="text">data_resource</title>
    <updated>date and time of request</updated>
    <link rel="self" title="data_resource" href="data_resource" />
    <entry>
        metadata_data
        <content type="application/xml">
            <m:properties>
                <d:object1 field1>object1 field_value1</d:object1 field1>
                <d:object1 field2>object1 field_value2</d:object1 field2>
                ...
            </m:properties>
        </content>
    </entry>
    <entry>
        metadata_data
        <content type="application/xml">
            <m:properties>
                <d:object2 field1>object2 field_value1</d:object2 field1>
                <d:object2 field2>object2 field_value2</d:object2 field2>
                ...
            </m:properties>
        </content>
    </entry>
</feed>
```

```
</m:properties>
</content>
</entry>
...
</feed>
```

The structural elements of the response body to a GET request have the following values:

entry

Instance of the collection object.

metadata_data

Metadata of the collection object instance.

object1 field1, object1 field2, ..., object2 field1, object2 field2, ...

Names of the field1, field2, ... fields of the object1, object2, ...collection object instances.

object1 field_value1, object1 field_value2, ..., object2 field_value1, object2 field_value2, ...

Values of the field1, field2, ... fields of the object1, object2, ...collection object instances.

Adding data

Primary rules of POST requests:

- Using parameters is not allowed.
- Request body is available.
- Response body is available.

POST request structure:

```
POST
my_Creatio_site/0/ServiceModel/EntityDataService.svc/objects_collectionCollection

Accept: application/atom+xml; type=entry
Content-Type: application/json; odata=verbose
ForceUseSession: true
BPMCSR: authentication_cookie_value
```

Only the object collection where you create a new collection object instance can be the data resource for the POST request.

Structural elements of the POST request string:

objects_collection (required)

The name of the collection where you need to create a new collection object instance.

Structure of the POST request body:

```
{
  "field1": "value1",
  "field2": "value2",
  ...
}
```

Structural elements of the POST request body:

field1, field2, ... (required)

Field names of the created collection object instance that you need to populate.

value1, value2, ... (required)

Field values of the field1, field2, ... of the created collection object instance that you need to populate.

Structure of a response body to a POST request:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xml:base="http://mycreatio.com/0/ServiceModel/EntityDataService.svc/"
      xmlns="http://www.w3.org/2005/Atom"
      xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
      xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
    <id>http://mycreatio.com/0/ServiceModel/EntityDataService.svc/data_resource</id>
    <title type="text">data_resource</title>
    <updated>date and time of request</updated>
    <link rel="self" title="data_resource" href="data_resource" />
    <entry>
        metadata_data
        <content type="application/xml">
            <m:properties>
                <d:object1 field1>object1 field_value1</d:object1 field1>
                <d:object1 field2>object1 field_value2</d:object1 field2>
                ...
            </m:properties>
        </content>
    </entry>
    <entry>
        metadata_data
        <content type="application/xml">
            <m:properties>
                <d:object2 field1>object2 field_value1</d:object2 field1>
                <d:object2 field2>object2 field_value2</d:object2 field2>
                ...
            </m:properties>
        </content>
    </entry>
    ...
</feed>
```

The structural elements of the response body to a POST request have the following values:

entry

Instance of the collection object.

metadata_data

Metadata of the created collection object instance.

object1 field1, object1 field2, ..., object2 field1, object2 field2, ...

Field names of the created collection object instance.

object1 field_value1, object1 field_value2, ..., object2 field_value1, object2 field_value2, ...

Field values of the field1, field2, ... of the created collection object instance.

Modifying data

Primary rules of PATCH requests:

- Using parameters is not allowed.
- Request body is available.
- Response body is not available.

Only the object collection where you modify data of the object instance can be the data resource for the PATCH request.

PATCH request structure:

```
PATCH
my_Creatio_site/0/ServiceModel/EntityDataService.svc/objects_collectionCollection(guid'object_id')
```

```
Accept: application/atom+xml; type=entry
Content-Type: application/json; odata=verbose
ForceUseSession: true
BPMCSR: authentication_cookie_value
```

Structural elements of the PATCH request string:

objects_collection (required)

The name of the collection whose object instance needs to be modified.

object_id (required)

Identifier of the modified collection object instance.

Structure of the PATCH request body:

```
{
    "field1": "value1",
    "field2": "value2",
    ...
}
```

Structural elements of the PATCH request body:

field1, field2, ... (required)

Field names of the modified collection object instance.

value1, value2, ... (required)

New field values of the field1, field2, ... of the created collection object instance.

Deleting data

Primary rules of DELETE requests:

- Using parameters is not allowed.
- Request body is not available.
- Response body is not available.

Only the object collection where you delete the object instance can be the data resource for the DELETE request.

DELETE request structure:

```
DELETE
my_Creatio_site/0/ServiceModel/EntityDataService.svc/objects_collectionCollection(guid'object_id')
```

```
Accept: application/atom+xml; type=entry
Content-Type: application/json; odata=verbose
ForceUseSession: true
BPMCSR: authentication_cookie_value
```

Structural elements of the DELETE request string:

objects_collection (required)

The name of the collection whose object instance needs to be deleted.

object_id (required)

Identifier of the deleted collection object instance.

Creatio integration via the OData 4 protocol

- **Introduction**
- **Implementation of OData 4 in Creatio**

- Request structure
- Request response structure
- Request types

Introduction

OData (Open Data Protocol) is an ISO/IEC confirmed OASIS standard determining the best practices for building and using REST API. It enables creating REST-based services that allow web clients to publish and edit resources using simple HTTP requests. Such resources should be identified with a URL and defined in the data model.

Creatio supports OData 3 and OData 4 protocols. OData 4 replaces OData 3 and considerably enhances the capacity of the previous protocol. These two protocols are not compatible by data format returned by the server. You can learn more about the protocols in [OData documentation](#).

To integrate with Creatio, use OData 4.

You need to **authenticate** your requests to Creatio.

Implementation of OData 4 in Creatio

Feature-UseODataV4 is the parameter responsible for using the OData 4 protocol in Creatio. For Creatio 7.15.3 and up, this parameter is included by default.

To include this parameter for Creatio 7.15.2 and lower versions, add the following string to the configuration file ..\Terrasoft.WebApp\Web.Config:

```
<add key="Feature-UseODataV4" value="true" />
```

The *odata* web service will provide access to Creatio objects via the OData 4 protocol.

Odata server address:

<https://mycreatio.com/0/odata>

Request structure

Structural elements of the request:

- String
- Headers
- Body

Query string

Query string structure:

```
method my_Creatio_site/0/odata/data_resource?$parameters
```

Example of a POST request string

```
// Add an instance of the "Contact" collection object  
POST https://mycreatio.com/0/odata/Contact
```

Example of a GET request string

```
// Receive an instance of the "Employee" collection objects, whose  
"Full Job Title" field contains the "Developer" value. The "Name" field  
value of the nested "Account" object collection should be other than  
"Our company".  
GET https://mycreatio.com/0/odata/Employee?$filter=FullJobTitle eq  
'Developer' and Account/Name ne 'Our company'
```

method

Creatio supports the following request methods:

- GET – receiving data.
- POST – adding data.
- PATCH – modifying data.
- DELETE – deleting data.

data_resource

Data resource is the path to the source of information obtained in the request response.

Data resource structure:

objects_collection(object_id)/object_field

Structural elements of the data resource:

objects_collection (required)

The name of the database table (name of the object collection).

You can receive the list of database tables if you run the following query:

For MySQL

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
```

For Oracle

```
SELECT * FROM ALL_TABLES
```

For PostgreSQL

```
SELECT table_name FROM information_schema.tables
```

object_id (optional)

The string identifier of the database table record (identifier of the collection object instance).

object_field (optional)

The database table record field (field of the collection object instance).

parameters

Non required parameters that you can use in the Creatio GET request string:

\$value

Field value.

\$count

\$count=true

The number of elements in the build.

\$skip

\$skip=n

N of the first elements that should not be in the build.

\$top

\$top=n

N of the first elements that should be in the build.

\$select

`$select=field1,field2,...`

List of fields that should be in the build.

\$orderby

`$orderby=field asc`

`$orderby=field desc`

Sorting of field values that are in the build.

\$expand

`$expand=field1,field2,...`

Extension of the connected fields.

\$filter

`$filter=field template 'field_value'`

Filtering of fields that should be in the build.

You can use the following operators when working with parameters:

`?`

Specifying the parameter.

`$`

Specifying the parameter name.

`&`

Using 2 or more parameters.

Request header

Headers that you can use in the requests to Creatio:

`Accept`

`Accept: application/json`

Possible data type of the server response.

`Content-Type`

`Content-Type: application/json; charset=utf-8`

Encoding and resource type passed in the query body.

`ForceUseSession`

`ForceUseSession: true`

The “ForceUseSession” title accounts for using the existing session. You do not need using AuthService.svc in your request to the authentication service.

`BPMCSRF`

`BPMCSRF: authentication_cookie_value`

Authentication cookie

The “Content-Type” and “Accept” headers are not required for the GET requests.

Example of POST request headers

`Accept: application/json`

`Content-Type: application/json; charset=utf-8`

`ForceUseSession: true`

BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO

Example of GET request headers

```
ForceUseSession: true  
BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO
```

Request body

Request body structure:

```
{  
    "field1": "value1",  
    "field2": "value2",  
    ...  
}
```

Structural elements of the request body:

field1, field2, ... (required)

Field names that are passed in the request body.

value1, value2, ... (required)

field1, field2, ...values that are passed in the request body.

Example of a POST request body

```
{  
    // Add the New User value in the Name field.  
    "Name": "New User",  
    // Add the Customer manager value in the JobTitle field.  
    "JobTitle": "Customer manager"  
}
```

Request response structure

Structural elements of the query response:

- Status code
- Body

Status code

As a request response, the server returns a status code encrypted in 3 digits. The first digit specifies the status class. The second and the third digits determine the response ordinal number. The server returns the following HTTP code classes when integrating with Creatio via the OData 4 protocol:

1. 2xx (success) – successful acceptance and processing of the client request.
2. 3xx (redirection) – you need to execute another request to successfully fulfill the operation.
3. 4xx (client error) – a request building error on the side of the client. The server must return a hypertext explanation of receiving the 4xx class code.
4. 5xx (internal server error) – unsuccessful execution of the request due to server issues.

Receiving the 4xx and 5xx class codes is a result of the unsuccessful request execution.

You can receive the following status codes from the server when sending requests via the OData 4 protocol:

200 OK The request that does not create a resource is successfully completed and the resource value does not equal zero. In this case, the response body must contain a resource value specified in the request URL. Information

returned with the response depends on the method used in the request

GET – the requested resource has been found and passed in the response body.

POST – resource that describes the server action to the request, is passed in the response body.

201 Created The request that successfully creates a resource. The response body must contain the created resource. Used for POST requests, that [create collection](#), [create multimedia object](#) (e.g., a picture) or [call the action through import](#).

202 Accepted The data processing request has been accepted but has not yet finished. There is no guarantee that the request will complete successfully ([asynchronous request processing](#)).

204 No Content The request has been processed successfully but there is no need to return any data. The requested resource has a zero value. The response only returned the headers, the response body should be empty.

3xx Redirection The redirection specifies that the client needs to take further action to execute the request. The response must contain [header Location](#) with URL that can be used to receive result. The response can also contain [header Retry-After](#), displaying time (in seconds). Thhis time characterizes the period that the client can wait before repeating the request that has been returned in [header Location](#).

304 Not Modified The client executes the GET request with [header If-None-Match](#) and the contents has not changed. The response should not contain other headers.

403 Forbidden The request is correct, but the server has refused to authorize it because the client has no permission to work with the requested resource. This may be caused by an invalid *BPMCSRF* cookie.

404 Not Found The server cannot find a resource specified in the URL. The response body can have additional information.

405 Method Not Allowed The resource specified in the request URL does not support the specified request method. The response should return the Allow header containing the list of request methods accessible for the resource.

406 Not Acceptable The resource specified in the request URL does not have any current view acceptable for the client as per [Accept](#), [Accept-Charset](#) and [Accept-Language](#) request headers. The service does not provide a default view.

410 Gone The requested resource is no longer available. The resource used the specified URL, but was deleted and is no longer available.

412 Precondition Failed In the request header, the client specified a condition that cannot be processed by the resource.

424 Failed Dependency The current request cannot be processed. The requested action depends on another action that has not been executed. The request has not been executed due to dependency failure.

501 Not Implemented The client is using a request method that is not implemented by OData 4 and cannot be processed. The response body must contain the description of the functionality that has not been implemented.

Response body

Response body structure:

```
{  
    "@odata.context": "http://my_Creatio_site/0/odata/$metadata#data_resource",  
    "value": [  
        {  
            "object1 field1": "object1 field_value1",  
            "object1 field2": "object1 field_value2",  
            ...  
        },  
        {  
            "object2 field1": "object2 field_value1",  
            "object2 field2": "object2 field_value2",  
            ...  
        },  
        ...  
    ]  
}
```

Structural elements of the response body:

`@odata.context`

Information on the type of the returned data. In addition to the data source path, the `data_resource` element can also contain the “`$entity`” parameter. This parameter indicates that the response returned a single instance of the collection object.

`value`

Contains the object collection. Is not available if the response contains a single instance of the collection object.

`Square brackets`

Object collection.

`Nested braces`

Instances of collection objects.

`object1 field1, object1 field2, ..., object2 field1, object2 field2, ...`

Names of the field1, field2, ... fields of the object1, object2, ... collection object instances.

`object1 field_value1, object1 field_value2, ..., object2 field_value1, object2 field_value2, ...`

Values of the field1, field2, ... fields of the object1, object2, ... collection object instances.

Example of a response to a POST request

Status: 201 Created

```
{  
    "@odata.context":  
    "http://mycreatio.com/0/odata/$metadata#Contact/$entity",  
    "Id": "37dab67d-199d-4275-9eb6-22bea86cb969",  
    "Name": "New User",  
    "OwnerId": "410006e1-ca4e-4502-a9ec-e54d922d2c00",  
    "CreatedOn": "2020-03-02T08:31:34.8076517Z",  
    "CreatedBy": "410006e1-ca4e-4502-a9ec-e54d922d2c00",  
    "ModifiedOn": "2020-03-02T08:31:34.8076517Z",  
    "ModifiedBy": "410006e1-ca4e-4502-a9ec-e54d922d2c00",  
    "ProcessListeners": 0,  
    "Dear": "",  
    "SalutationTypeId": "00000000-0000-0000-0000-000000000000",  
    "GenderId": "00000000-0000-0000-0000-000000000000",  
    "AccountId": "00000000-0000-0000-0000-000000000000",  
    "DecisionRoleId": "00000000-0000-0000-0000-000000000000",  
    "TypeId": "00000000-0000-0000-0000-000000000000",  
    "JobId": "00000000-0000-0000-0000-000000000000",  
    "JobTitle": "Customer manager",  
    "DepartmentId": "00000000-0000-0000-0000-000000000000",  
    "BirthDate": "0001-01-01T00:00:00Z",  
    "Phone": "",  
    "MobilePhone": "",  
    "HomePhone": "",  
    "Skype": "",  
    "Email": "",  
    "AddressTypeId": "00000000-0000-0000-0000-000000000000",  
    "Address": "",  
    "CityId": "00000000-0000-0000-0000-000000000000",
```

```
"RegionId": "00000000-0000-0000-0000-000000000000",
"Zip": "",
"CountryId": "00000000-0000-0000-0000-000000000000",
"DoNotUseEmail": false,
"DoNotUseCall": false,
"DoNotUseFax": false,
"DoNotUseSms": false,
"DoNotUseMail": false,
"Notes": "",
"Facebook": "",
"LinkedIn": "",
"Twitter": "",
"FacebookId": "",
"LinkedInId": "",
"TwitterId": "",
"TwitterAFDAId": "00000000-0000-0000-0000-000000000000",
"FacebookAFDAId": "00000000-0000-0000-0000-000000000000",
"LinkedInAFDAId": "00000000-0000-0000-0000-000000000000",
"PhotoId": "00000000-0000-0000-0000-000000000000",
"GPSN": "",
"GPSE": "",
"Surname": "User",
"GivenName": "New",
"MiddleName": "",
"Confirmed": true,
"IsNonActualEmail": false,
"Completeness": 0,
"LanguageId": "6ebc31fa-ee6c-48e9-81bf-8003ac03b019",
"Age": 0
}

}
```

Example of a response to a GET request

```
Status: 200 OK

{
  "@odata.context": "http://mycreatio.com/0/odata/$metadata#Employee",
  "value": [
    {
      "Id": "c31c7862-fe33-4a13-9bbc-0943fa08fd02",
      "CreatedOn": "2017-03-30T14:50:04Z",
      "CreatedBy": "76929f8c-7e15-4c64-bdb0-adc62d383727",
      "ModifiedOn": "2020-02-14T06:30:46.234Z",
      "ModifiedBy": "410006e1-ca4e-4502-a9ec-e54d922d2c00",
      "ProcessListeners": 0,
      "ContactId": "227aab3b-7c0c-4181-abf9-81585563ab23",
      "Name": "William Walker",
      "OrgStructureUnitId": "d436a9ce-9690-4415-9e03-
e8061d7cabbb5",
      "Notes": ""
    }
  ]
}
```

```
        "JobId": "11d68189-ced6-df11-9b2a-001d60e938c6",
        "FullJobTitle": "Developer",
        "OwnerId": "76929f8c-7e15-4c64-bdb0-adc62d383727",
        "CareerStartDate": "2019-09-08T00:00:00Z",
        "CareerDueDate": "0001-01-01T00:00:00Z",
        "ProbationDueDate": "2020-01-09T00:00:00Z",
        "ReasonForDismissalId": "00000000-0000-0000-0000-
000000000000",
        "AccountId": "a0bf3e92-f36b-1410-0499-00155d043204",
        "ManagerId": "3e5bd47e-1ebd-41db-a9a6-a3560dcee3cb"
    }
]
}
```

Request types

Reading data

Primary rules of GET requests

- Using parameters is allowed.
- Request body is not available.
- Response body is available.

GET request structure:

```
GET my_Creatio_site/0/odata/data_resource?$parameters

Accept: application/json
Content-Type: application/json; charset=utf-8
ForceUseSession: true
BPMCSRF: authentication_cookie_value
```

Structure of a response body to a GET request

```
{
    "@odata.context": "http://my_Creatio_site/0/odata/$metadata#data_resource",
    "value": [
        {
            "object1 field1": "object1 field_value1",
            "object1 field2": "object1 field_value2",
            ...
        },
        {
            "object2 field1": "object2 field_value1",
            "object2 field2": "object2 field_value2",
            ...
        },
        ...
    ]
}
```

The structural elements of the response body to a GET request have the following values:

`@odata.context`

Information on the type of the returned data. In addition to the data source path, the `data_resource` element can also contain the “`$entity`” parameter. This parameter indicates that the response returned a single instance of the collection object.

`value`

Contains the object collection. Is not available if the response contains a single instance of the collection object.

object1 field1, object1 field2, ..., object2 field1, object2 field2, ...

Names of the field1, field2, ... fields of the object1, object2, ...collection object instances.

object1 field_value1, object1 field_value2, ..., object2 field_value1, object2 field_value2, ...

Values of the field1, field2, ... fields of the object1, object2, ...collection object instances.

Adding data

Primary rules of POST requests:

- Using parameters is not allowed.
- Request body is available.
- Response body is available.

POST request structure:

```
POST my_Creatio_site/0/odata/objects_collection
```

Accept: application/json

Content-Type: application/json; charset=utf-8

ForceUseSession: true

BPMCSR: authentication_cookie_value

Only the object collection where you create a new collection object instance can be the data resource for the POST request.

Structural elements of the POST request string:

objects_collection (required)

The name of the collection where you need to create a new collection object instance.

Structure of the POST request body:

```
{  
    "field1": "value1",  
    "field2": "value2",  
    ...  
}
```

Structural elements of the POST request body:

field1, field2, ... (required)

Field names of the created collection object instance that you need to populate.

value1, value2, ... (required)

Field values of the field1, field2, ... of the created collection object instance that you need to populate.

Structure of a response body to a POST request:

```
{  
    "@odata.context":  
    "http://my_Creatio_site/0/odata/$metadata#data_resource/$entity",  
    "field1": "value1",  
    "field2": "value2",  
    ...  
}
```

The structural elements of the response body to a POST request have the following values:

@odata.context

Information on the type of the returned data. In addition to the data source path, the data_resource element can also contain the "\$entity" parameter. This parameter indicates that the response returned a singe instance of the

collection object.

field1, field2, ...

Field names of the created collection object instance.

value1, value2, ...

Field values of the field1, field2, ... of the created collection object instance.

Modifying data

Primary rules of PATCH requests:

- Using parameters is not allowed.
- Request body is available.
- Response body is not available.

Only the object collection where you modify data of the object instance can be the data resource for the PATCH request.

PATCH request structure:

```
PATCH my_Creatio_site/0/odata/objects_collection(object_id)
```

Accept: application/json

Content-Type: application/json; charset=utf-8

ForceUseSession: true

BPMCSR: authentication_cookie_value

Structural elements of the PATCH request string:

objects_collection (required)

The name of the collection whose object instance needs to be modified.

object_id (required)

Identifier of the modified collection object instance.

Structure of the PATCH request body:

```
{  
    "field1": "value1",  
    "field2": "value2",  
    ...  
}
```

Structural elements of the PATCH request body:

field1, field2, ... (required)

Field names of the modified collection object instance.

value1, value2, ... (required)

New field values of the field1, field2, ... of the created collection object instance.

Deleting data

Primary rules of DELETE requests:

- Using parameters is not allowed.
- Request body is not available.
- Response body is not available.

Only the object collection where you delete the object instance can be the data resource for the DELETE request.

DELETE request structure:

```
DELETE my_Creatio_site/0/odata/objects_collection(object_id)

Accept: application/json
Content-Type: application/json; charset=utf-8
ForceUseSession: true
BPMCSR: authentication_cookie_value
```

Structural elements of the DELETE request string:

objects_collection (required)

The name of the collection whose object instance needs to be deleted.

object_id (required)

Identifier of the deleted collection object instance.

DataService

Beginner

Easy

Medium

Advanced

General information

The DataService web service is used for handling the requests from the Creatio client side.

The full list and description of the DataService data contracts is displayed on table 1.

Table 1. The Creatio application DataService services

Service	Description
SchemaDesignerRequest	Schema designer request class. Not recommended to use.
EntitySchema	Object schema class. Not recommended to use.
ClientUnitSchema	Client schema class. Not recommended to use.
RemoveEntitySchemaRequest	Remove object schema request. Not recommended to use.
RemoveClientUnitSchemaRequest	Remove client schema request. Not recommended to use.
EntitySchemaRequest	Receive object schema instance request. Not recommended to use.
ClientUnitSchemaRequest	Receive client object schema instance request. Not recommended to use.
ProcessUserTaskSchemaRequest	Receive business process user action schema request. Not recommended to use.
UpdatePackageSchemaDataRequest	Receive package schema data update request. Not recommended to use.
ProcessSchemaRequest	Receive process schema instance request. Not recommended to use.
ContractProcessSchema	Process schema contract class. Not recommended to use.
RemoveProcessSchemaRequest	Remove process schema request. Not recommended to use.
InsertQuery	Add section record query class.
UpdateQuery	Update section record query class.
DeleteQuery	Delete section record query class.
SelectQuery	Select section record query class.
BatchQuery	Package query class.
UserProfile	User profile class. Not recommended to use.
QueryProfile	Query profile class. Not recommended to use.
QuerySysSettings	Receive system settings list request. Not recommended to use.
PostSysSettingsValue	Set system setting value class. Not recommended to use.

PostSysSettingsValues	Set system setting values class. Not recommended to use.
Filters	Filter class.
QueryModuleDescriptors	Receive module descriptors query class. Not recommended to use.
ClientLoggerDataContract	Client log data class. Not recommended to use.
PostClientLog	Client log post class. Not recommended to use.
UploadFile	File upload class. Not recommended to use.
GetTelephonyConfig	Receive telephony configuration settings class. Not recommended to use.
InsertSysSettingRequest	Add system setting query class. Not recommended to use.
UpdateSysSettingRequest	Edit system setting query class. Not recommended to use.
DeleteSysSettingRequest	Delete system setting query class. Not recommended to use.
GetTests	Receive all Unit tests class. Not recommended to use.
RunTests	Run Unit tests class. Not recommended to use.

The ProcessEngineService.svc web service

Основы Легкий Средний **Сложный**

Contents

- **Introduction**
- **ProcessEngineService.svc request**
- **ProcessEngineService.svc response**
- **Executing a separate element of the business process**

Introduction

Running business processes is one of the purpose of integration the external application with Creatio. The *ProcessEngineService.svc* web service that enables running business processes from the outside is implemented for this.

Example of using the *ProcessEngineService.svc* web service is covered in the "**How to run Creatio processes via web service**" article. You can find the full list of web service methods in the [ProcessEngineService Class Members documentation](#).

ProcessEngineService.svc request

Example of the request to the ProcessEngineService.svc service

```
GET  
/7.15.1.1295_SalesEnterprise/0/ServiceModel/ProcessEngineService.svc/Cu  
stomProcess/Execute?  
ResultParameterName=CustomProcessResult&processParam=15 HTTP/1.1  
Host: localhost  
BPMCSRF: bmJcGKDsmKnGmK0yKPUfR.  
Cookie: BPMSESSIONID=mdbspc31pbvldg4qwrotnoe1;  
BPMLOADER=2cqcd4ean14rmlkyllo35qfnt;  
.ASPXAUTH=90A57901E17893AF50528FB608B52A1FFCA8359E525D72B95A2C36835171F  
33B4E6B6421920F5F73EC7B6C707402E2243C142738E6DC1B0CB6B9C8DF1921B53BDBAD  
F3D076F0CB671FBC33EECEC1AD587D119ECA63FE49C8C4CC0DB4AD6A9C720778E249754  
C8BE34AD082AFD0A027A4CC7B6783003DD659FA077ED0C968602BF8841291B6380D3BC5
```

```
3C62C88D9EAA8D2096DB44B0EFD1428637F3D9554A58AE59E07D8777D0C8DC41290156F
176F596014D1E5D6648A038FBBA557048EF0DE9FF8AABA9AB3D34EC157F6C839BA24F8C
5D2D11FE72664D7791559D37A5C29AEDAF2F55A87E68B2250864B1C082A679267D25135
F780370635DBFCBFAC3F96EA32001E5AC80DBDB2C5C401C576875F875549872EED27946
23D2E0AF6E89B14230BC6847D19AF1951863035B43BB6C04105AA063EC9BDFC2EE5C8FC
4D13B0B30009443A6BC42F0E2434C2F924D13433D8392A051308CF98209428E6A70AB10
F4BBA46E7C328299EBCA6743C94D90BDC6C18F0AA6F8CCF52F014C29A17E38EFD7B061F
60853DB063977C9601385D4D648EB63D8B487;
UserName=83|117|112|101|114|118|105|115|111|114;
BPMCSRF=bmJcGKDsmKnGmK0yKPUfR.
```

To run a specific business process, call the *Execute()* method of the service.

Web service request structure:

- query string;
- request headers.

Query string

Query string structure

```
http(s)://[Creatio application
URL]/0/ServiceModel/ProcessEngineService.svc/ProcessSchemaName/Execute?
ResultParameterName=resultParameterName&inputParameterValue
```

Example of a query string

```
https://mycreatio.com/0/ServiceModel/ProcessEngineService.svc/CustomProcess/Execute?
ResultParameterName=CustomProcessResult&processParam=15
```

Query string parameters

ProcessSchemaName

You can find the name of the business process schema in the [Configuration] section.

resultParameterName (the parameter is not required).

The code of the process parameter that stores the result of the process execution. If this parameter is not specified, the web service will run the business process without waiting for the result of its execution.

If the parameter code is not available in the process being called, the web service will return a *null* value.

inputParameter (the parameter is not required).

The incoming parameter codes of the business process. If you pass several incoming parameters, use a “&” character to combine them.

inputParameterValue

The incoming parameter values of the business process.

Request header

BPMCSRF

BPMCSRF: authentication_cookie_value

Authentication cookie

Before calling the web service using third-party tools, authenticate the user who will run the requests. For this, use the AuthService.svc service (see "**Authentication of external requests**").

ForceUseSession

ForceUseSession: true

The “ForceUseSession” title accounts for using the existing session.

ProcessEngineService.svc response

Example of the response

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">
  [
    {"Id": "c38d46fd-e405-491a-a508-
01bb9760eecc", "Name": "Valerie E. Murphy", "Phone": "+3090"}, 
    {"Id": "516bc5b4-a79a-4e1d-9894-
0566617012d1", "Name": "Sharyn Mccraney", "Phone": "+1 33867 411 85
76"}, 
    {"Id": "41b67c9e-7c14-487c-a217-
057fd3735ba1", "Name": "Shela Andry", "Phone": "+1 33985 177 50
37"}, 
    {"Id": "f79fe861-88af-4e23-9bb7-
0625a6cee703", "Name": "Kate Roberts", "Phone": "+1 212 775
9012"}, 
  ]
</string>
```

The result of executing the *Execute()* method is returned as a string containing the JSON object (it is possible to get the *null*).

Deserialization of the JSON object and bringing the result to a specific type of data must be performed in the code that calls the web service.

Executing a separate element of the business process

To execute a separate element of the business process, call the *ExecProcElByUID* web service method. This method accepts the Id of the executed process element as a parameter.

Query string:

```
http[s]://[Creatio application
URL]/0/ServiceModel/ProcessEngineService.svc/ExecProcElByUID/ProcessElementUID
```

ProcessElementUID

Id of the executed element of the process.

You can only execute an element of the process that has been run.

If the *ExecProcElByUID* process element has already been completed when calling the method, this element will not be executed.

Integration examples with Creatio

Contents

- **Integration tools**
- **OData**
- **DataService**

Integration tools

Contents

- **Executing OData queries using Fiddler**
- **Working with requests in Postman**
- **Working with request collections in Postman**

Executing OData queries using Fiddler

Beginner

Easy

Medium

Advanced

Introduction

Integration with Creatio using the OData protocol requires executing HTTP requests to the *EntityDataService.svc*. Requests can be compiled in any programming language: C#, PHP, etc. However, it is recommended to use HTTP request debugging tools, such as [PostMan](#) or [Fiddler](#) for better understanding of general principles for request formatting. This article covers examples of requests composed with the help of Fiddler.

More information about OData protocol can be found in the “**Creatio integration over the OData protocol (on-line documentation)**” article.

Authentication

Before making requests to *EntityDataService.svc*, a third party application must be authenticated and receive the corresponding *cookies*. Creatio’s authentication uses a separate web service: *AuthService.svc*. For more information about this service, please see the “**The AuthService.svc authentication service (on-line documentation)**” article.

To execute a request to *AuthService.svc* using Fiddler, go to the [Composer] tab and execute the following (Fig. 4):

1. Select HTTP method POST.
2. Specify the authentication service URL generated according to the following mask:

```
http(s)://[Creatio application address]/ServiceModel/AuthService.svc/Login
```

Example:

```
https://012496-sales-enterprise.creatio.com/ServiceModel/AuthService.svc/Login
```

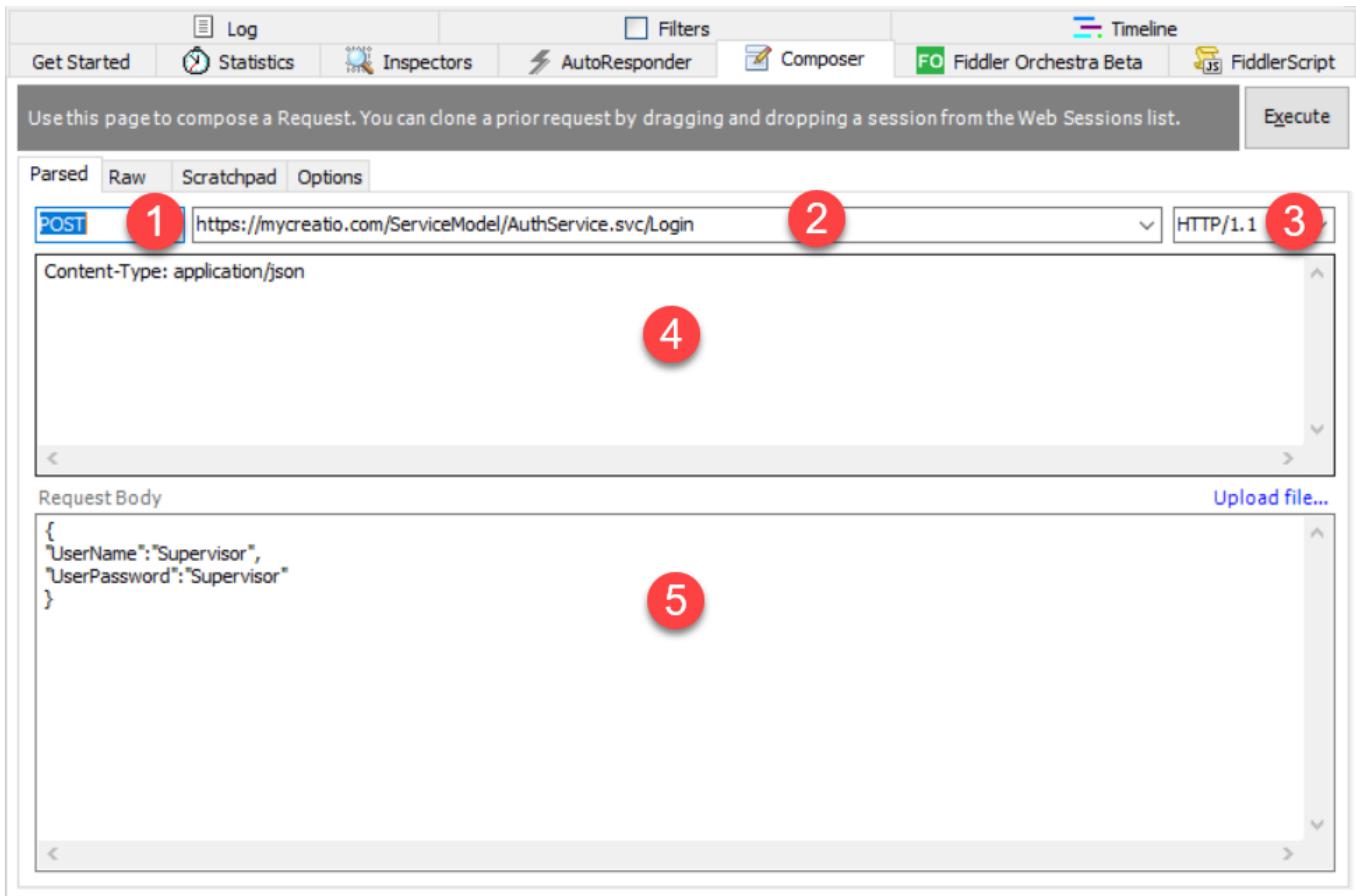
3. Specify HTTP protocol version 1.1.
4. Specify the type of the request body:

Content-Type: application/json

5. Add the request body – a JSON object with the authentication data (login and password):

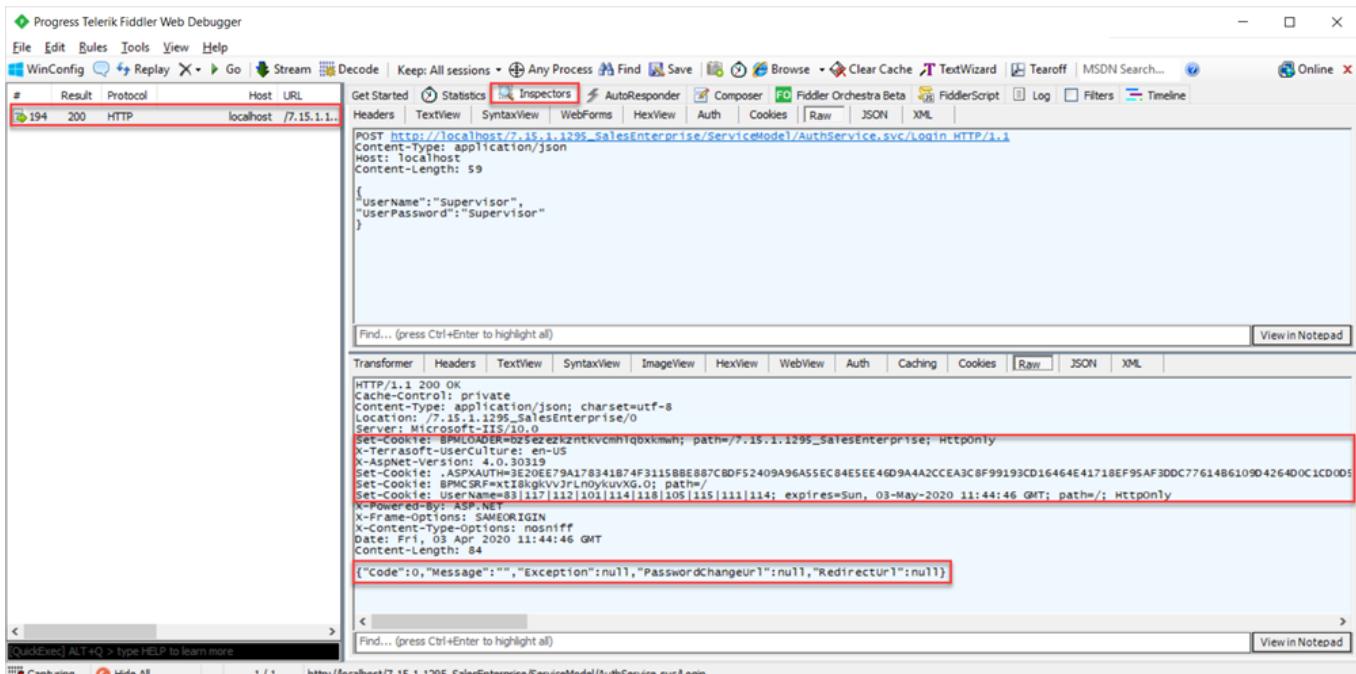
```
{
    "UserName": "Supervisor",
    "UserPassword": "Supervisor"
}
```

Fig. 4. Generating authentication request



Execute the request by clicking the [Execute] button. As a result, the Fiddler session window will display a response from the *AuthService.svc* service (Fig. 5). Double-click the reply string (1) to open the [Inspectors] tab with the response properties.

Fig. 5. Properties of HTTP response from the AuthService.svc



The *cookies* received in the HTTP response (BPMLOADER, .ASPxAuth and BPMCSRF) are to be used in all further requests to Creatio, that require authentication data (Fig. 5-2).

If the authentication has been successful, the response body will contain a JSON object whose `Code` property will be

set to "0" (Fig. 5, 3). In case of errors, JSON object properties will contain corresponding code and message.

If it is necessary to send a request and get a response in the JSON format, use the following key-value pairs in the request header:

```
Content-Type = "application/json"  
Accept = "application/json;odata=verbose"
```

Adding data

Example: add a new activity. Fill out the [Title], [Owner] and [Notes] columns.

To compose a request to add such data using Fiddler, go to the [Composer] tab and execute the following (Fig. 6):

1. Select HTTP method POST.
2. Specify the *EntityDataService.svc* service URL generated according to the following mask:

```
http(s)://[Creatio application  
address]/0/ServiceModel/EntityDataService.svc/ActivityCollection/
```

3. Specify HTTP protocol version 1.1.

4. Add the following in the request title:

- Query content type – application/json.
- Required cookies (BPMLOADER, .ASPXAUTH, BPMSESSIONID and BPMCSRF).
- CSRF token BPMCSRF that contains the value of the corresponding cookie (BPMCSRF).

Example of request's HTTP title:

```
Accept: application/atom+xml  
Content-Type: application/atom+xml;type=entry  
Cookie: BPMSESSIONID=cxa54p2dsb4wnqbbzvgyxcoo; BPMCSRF=6yCmyILS1IE8/toyQm9Ca.;  
BPMLOADER=rqqjjeqyfaudfyk4xu404j5f; .ASPXAUTH=697...A292D8164;  
BPMCSRF: 6yCmyILS1IE8/toyQm9Ca.
```

If protection from CSRF attacks is enabled, use both the BPMCSRF *cookie* and BPMCSRF token. For more information, see "**Protection from CSRF attacks during integration with Creatio (on-line documentation)**".

Protection from CSRF attacks is disabled on Creatio trial websites. Therefore, there is no need to use both BPMCSRF *cookie* and token in the request titles.

User session is created only upon the first request to the *EntityDataService.svc*, after which the BPMSESSIONID *cookie* will be returned in the response (Fig. 8, 2). Therefore, there is no need to add BPMSESSIONID *cookie* to the title of the first request (Fig. 6, 4).

If you do not add BPMSESSIONID *cookie* to each subsequent request, then each request will create a new user session. Significant frequency of requests (several or more requests a minute) will increase the RAM consumption which will decrease performance.

If it is necessary to send a request and get a response in the JSON format, use the following key-value pairs in the request header:

```
Content-Type = "application/json"  
Accept = "application/json;odata=verbose"
```

5. Add contents in XML format to the HTTP request body:

```
<?xml version="1.0" encoding="utf-8"?>  
<entry xmlns="http://www.w3.org/2005/Atom">  
    <content type="application/xml">  
        <properties  
            xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">  
                <Title  
                    xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices">process the incomming  
                    website form request</Title>
```

```

<Notes
xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices">please, email to
client@gmail.com and process the following request: clients request</Notes>
<OwnerId
xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices">64844c83-c6c2-4eee-
a0e9-e26cef529d2f</OwnerId>
</properties>
</content>
</entry>
```

This request fills out all required object columns.

If an object column is a lookup, specify the lookup database Id instead of lookup name. Add the “Id” suffix to the lookup column name in the request. In the current example, the lookup column is [Owner], and the “OwnerId” identifier is specified for it in the request.

You can view the OwnerId value in the browser, by opening corresponding record for editing (Fig. 7) obtain via a query.

Fig. 6. Composing an insert query

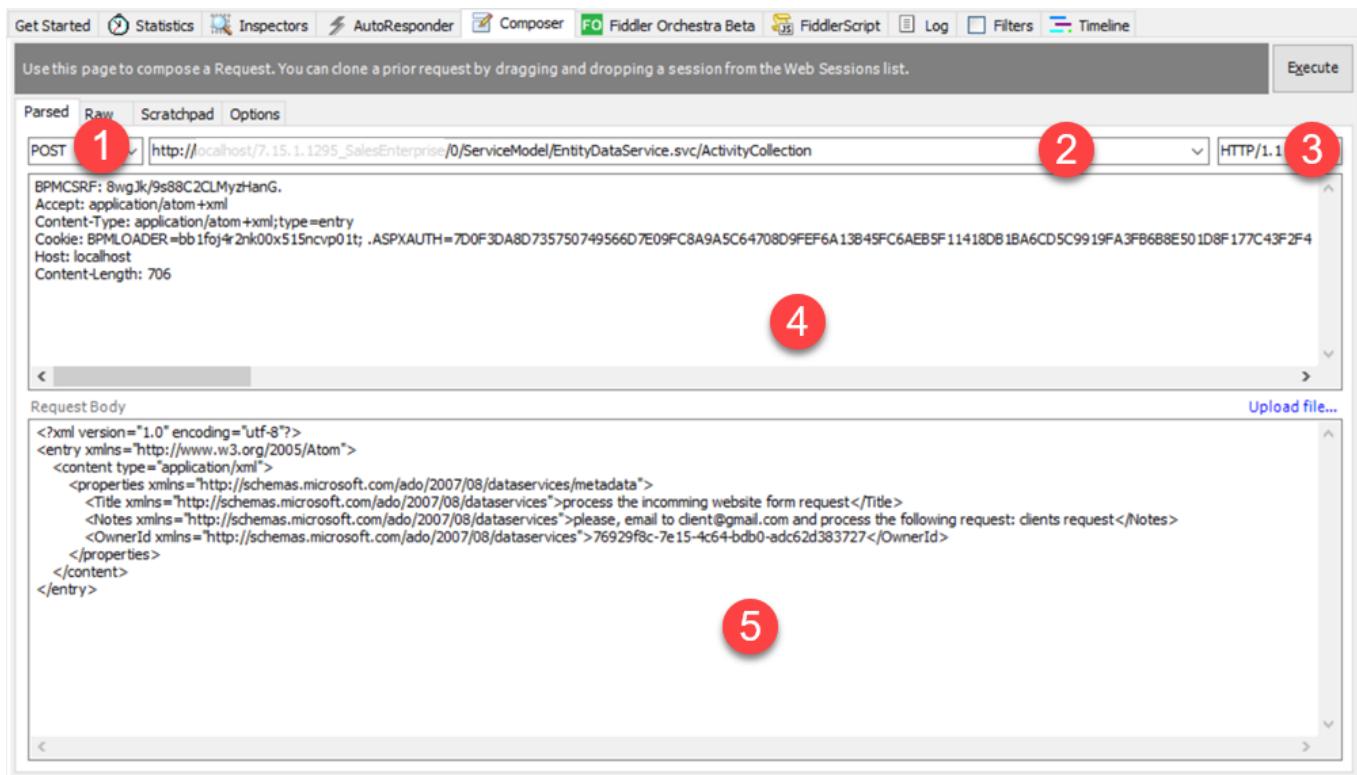


Fig. 7. Contact Id displayed in the browser

The screenshot shows the Creatio Sales Enterprise application interface. On the left is a vertical navigation bar with icons for Sales, Dashboards, Feed, Leads, Accounts, Contacts, Activities, Opportunities, Orders, Contracts, and Invoices. The main area displays a contact card for "Andrew Baker (sample)". The card includes a profile picture, a green progress bar at 100%, the time "8:11 AM, Boston", and a "NEXT STEPS (0)" section with icons for phone, email, and tasks. Below the card are sections for "CONTACT INFO", "CURRENT EMPLOYMENT", "MAINTENANCE", "TIMELINE", and "HISTO". Under "CONTACT INFO", fields include "Full name*" (Andrew Baker (sample)), "Full job title" (Specialist), "Mobile phone" (+1 617 221 5187), "Business phone" (+1 617 440 2031), "Email" (a.baker@ac.com), and "Recipient's name" (empty). Under "CURRENT EMPLOYMENT", it shows "Type" (Customer), "Title" (Mr.), "Owner" (Supervisor), "Gender" (Male), and "Preferred language" (empty). The "Communication options" section lists "Mobile phone" (+1 617 221 5187) and "Business phone" (+1 617 440 2031) with their respective icons. A sidebar on the right shows notifications for 6 messages and 1 task.

Execute the request by clicking the [Execute] button. As a result, the Fiddler session window will display a response from the *EntityDataService.svc* service (Fig. 8). Double-click the reply string (1) to open the [Inspectors] tab with the response properties.

Fig. 8. Properties of HTTP response from the EntityDataService.svc

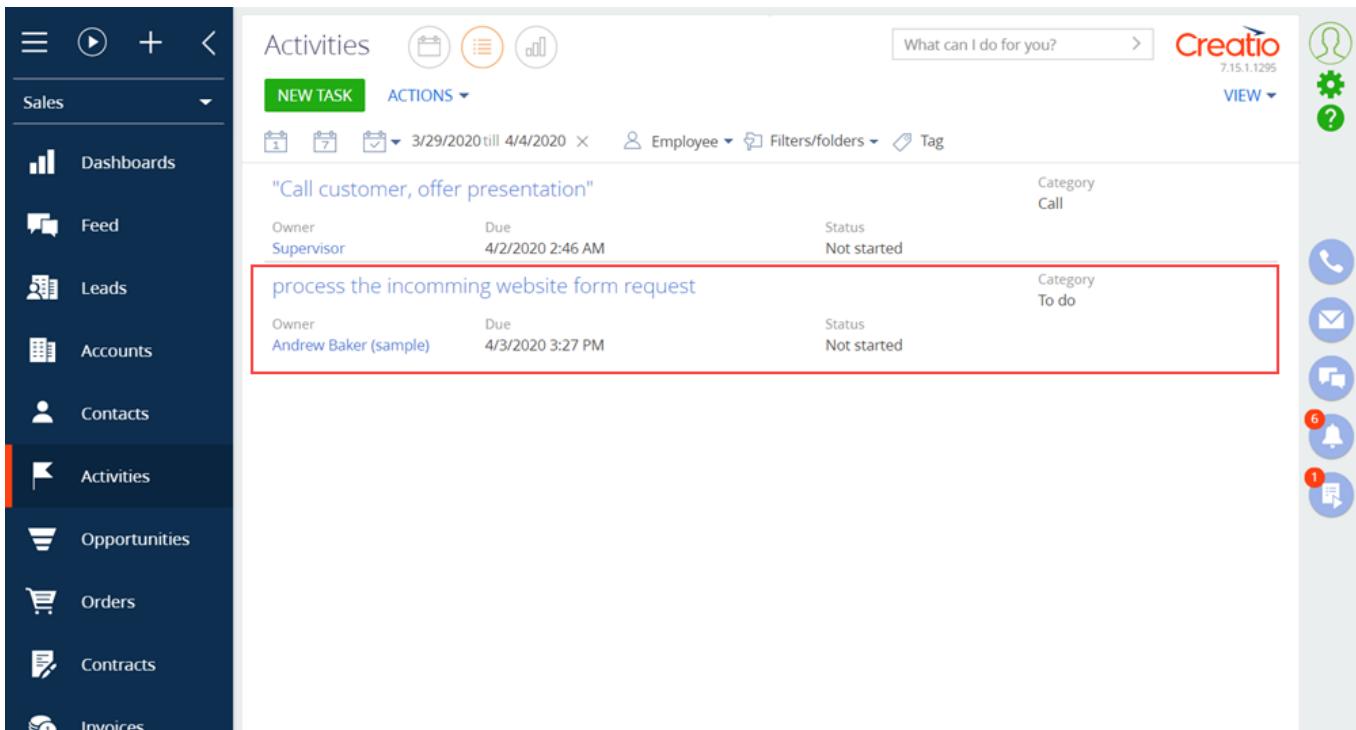
The screenshot shows the Fiddler Web Debugger interface. The main pane displays a list of network sessions. A specific session is highlighted with a red box and circled with a large red number '1'. The details tab for this session shows a POST request to `http://localhost:7.15.1.1295/SalesEnterprise/0/ServiceModel/EntityDataService.svc/ActivityCollection`. The response body is a large XML document representing an activity collection. A red box highlights the XML content, with a red number '2' circled around it. A red number '3' is circled around the bottom right corner of the XML block. The status bar at the bottom indicates the URL `http://localhost:7.15.1.1295.SalesEnterprise/0/ServiceModel/EntityDataService.svc/ActivityCollection`.

The response from the *EntityDataService.svc* service may contain the **BPMSESSIONID** cookie if the request is executed for the first time (Fig. 8, 2).

The response body contains the added record in the XML format (Fig. 8, 3). The <id> XML element contains the identifier of the added activity, that can be used in other requests that need to work with this record.

As a result, a new record will be added in the [Activities] section (Fig. 9).

Fig. 9. Results of the activity add request



Data selection

The data select query does not have body. Data filtering is done based on the URL parameter values. For more information on the data select queries with filters, see the **“Examples of requests for filter selection (on-line documentation)”** article.

To compose a request to select data using Fiddler, go to the [Composer] tab and execute the following (Fig. 10):

1. Select HTTP method POST.
2. Specify the *EntityDataService.svc* service URL generated according to the following mask:

```
http(s)://[Creatio application address]/0/ServiceModel/EntityDataService.svc/ActivityStatusCollection?  
$filter=Code%20eq%20'InProgress'
```

3. Specify HTTP protocol version 1.1.

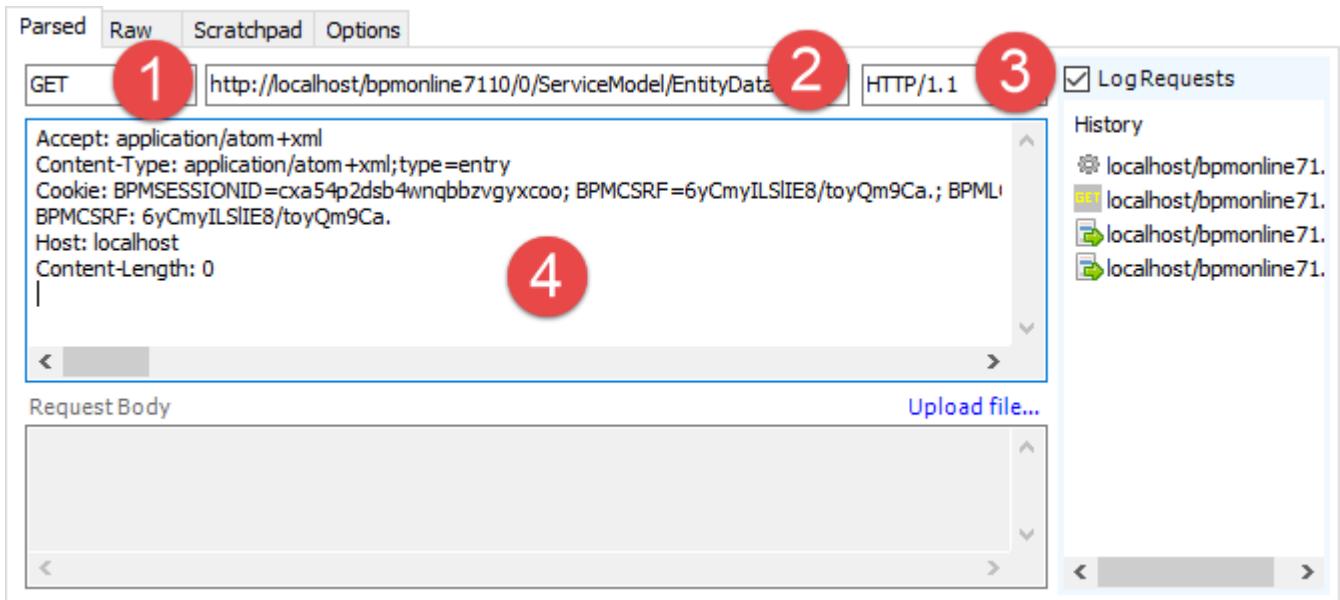
4. In the request title, specify the request body type as application/atom+xml. Add all necessary cookies to the request title (BPMLOADER, .ASPXAUTH, BPMSESSIONID, BPMCSRFT) and the BPMCSRFT token:

```
Accept: application/atom+xml  
Content-Type: application/atom+xml; type=entry  
Cookie: BPMSESSIONID=cxa54p2dsb4wnqbbzvgyxcoo; BPMCSRFT=6yCmyILS1IE8/toyQm9Ca.;  
BPMLOADER=rqqjjeqyfaudfyk4xu404j5f; .ASPXAUTH=697...A292D8164;  
BPMCSRFT: 6yCmyILS1IE8/toyQm9Ca.
```

If it is necessary to send a request and get a response in the JSON format, use the following key-value pairs in the request header:

```
Content-Type = "application/json"  
Accept = "application/json;odata=verbose"
```

Fig. 10. Composing data select query



Execute the request by clicking the [Execute] button. As a result, the Fiddler session window will display a response from the *EntityDataService.svc* service (Fig. 11). Double-click the reply string (1) to open the [Inspectors] tab with the response properties. The body of the HTTP response contains the selection result (2).

Fig. 11. Properties of HTTP response from the EntityDataService.svc

The screenshot shows the Telerik Fiddler Web Debugger interface. A red circle numbered 1 highlights the selected response in the list of sessions. The response details are shown in the main pane, with a red circle numbered 2 highlighting the 'Raw' tab containing the XML response body. The response body includes XML tags like <ActivityStatusCollection>, <ActivityTypeStatusEntryCollection>, and <ActivityCollectionByStatus>.

Update data

Example: modify the title of the added activity.

To compose a request to add data using Fiddler, go to the [Composer] tab and execute the following (Fig. 12):

1. Specify HTTP method POST.

Using HTTP methods PUT and DELETE will cause the "405 Method not allowed" error if HTTP extension WebDAV is disabled in the application's Web.Config file.

2. Specify the *EntityDataService.svc* service URL generated according to the following mask:

```
http(s)://[Creatio application address]/0/ServiceModel/EntityDataService.svc/ActivityCollection(guid'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX')
```

Use the unique identifier of the added record (a record Id looks like this: 9741ffe-ff81-46ba-8d99-1f488ec5502e), which can be obtained in an HTTP response to the add request. You can also view record Id in a browser, by opening a record for editing (Fig. 13).

3. Specify HTTP protocol version 1.1.

4. In the request title, specify the request body type as application/atom+xml. Add all necessary cookies to the request title (BPMLOADER, .ASPXAUTH, BPMSESSIONID, BPMCSRF) and the BPMCSRF token:

```
Accept: application/atom+xml
Content-Type: application/atom+xml;type=entry
Cookie: BPMSESSIONID=cxa54p2dsb4wnqbbzvgyxcoo; BPMCSRF=6yCmyILS1IE8/toyQm9Ca.; BPMLOADER=rqqjjeqyfaudfyk4xu404j5f; .ASPXAUTH=697...A292D8164; BPMCSRF: 6yCmyILS1IE8/toyQm9Ca.
```

If it is necessary to send a request and get a response in the JSON format, use the following key-value pairs in the request header:

Content-Type = "application/json"

Accept = "application/json;odata=verbose"

5. Add contents in the XML format to the request body.

```
<?xml version="1.0" encoding="utf-8"?>
<entry xmlns="http://www.w3.org/2005/Atom">
    <content type="application/xml">
        <properties
            xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata">
            <Title
                xmlns="http://schemas.microsoft.com/ado/2007/08/dataservices">process the incomming website form request (Updated)</Title>
        </properties>
    </content>
</entry>
```

It is recommended to specify only columns that can be modified.

Fig. 12. Composing data update query

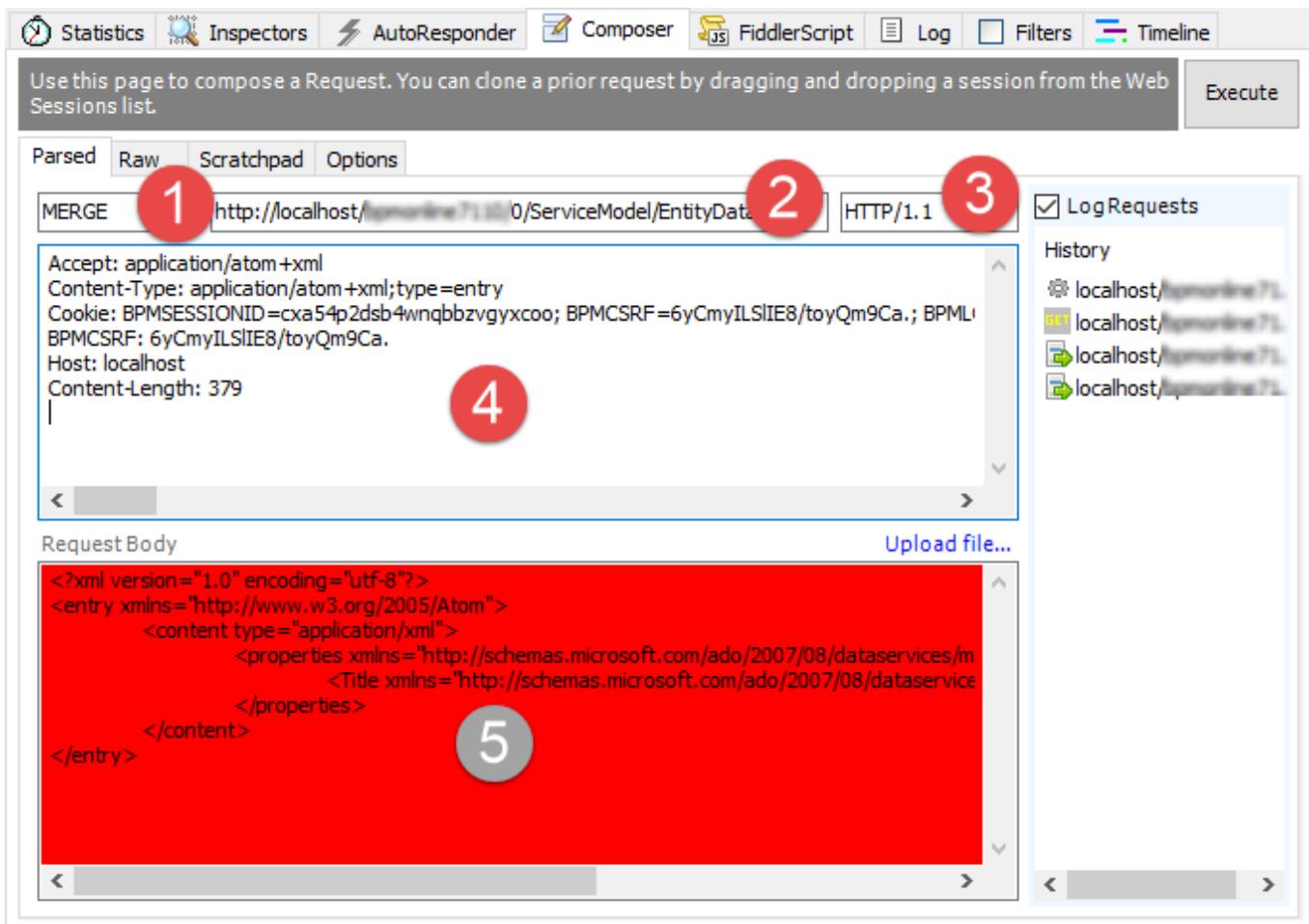


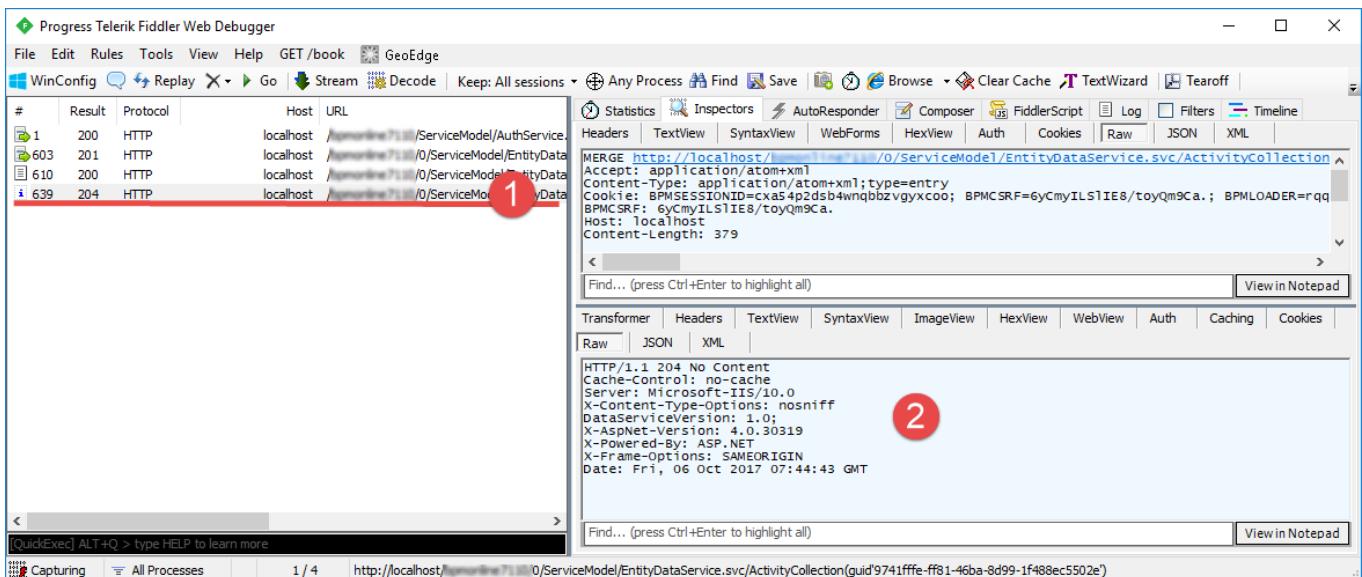
Fig. 13. Activity Id displayed in the browser

The screenshot shows the Creatio application interface with the following details:

- Task Card:** process the incoming website form request (Updated)
- Details:**
 - Start: 10/6/2017
 - Due: 10/6/2017
 - Status: Not started
 - Owner: Andrew Baker (sample)
 - Reporter: Andrew Baker (sample)
 - Priority: Medium
 - Category: To do
- Sidebar:** General, Dashboards, Employees, Contacts, Accounts, Activities (highlighted), Feed.
- Right Panel:** Creatio logo, user icons (Profile, Settings, Help), and a red notification bubble with the number 3.

Execute the request by clicking the [Execute] button. As a result, the Fiddler session window will display a response from the *EntityDataService.svc* service (Fig. 14). Double-click the reply string (1) to open the [Inspectors] tab with the response properties. The body of the HTTP response is empty (2).

Fig. 14. Properties of HTTP response from the EntityDataService.svc



As a result, the title of the record will be modified (Fig. 15).

Fig. 15. Results of the activity edit request

The screenshot shows the Creatio Activities module. At the top, there are buttons for 'NEW TASK' and 'ACTIONS ▾'. On the right, there are icons for 'VIEW ▾', 'Employee', 'Filter', and 'Tag'. Below the header, there are three sections of tasks:

- Visit bpm'online Academy** (Category: To do)

Owner: Supervisor	Due: 10/1/2016 3:45 PM	Status: Not started
-------------------	------------------------	---------------------
- process the incoming website form request (Updated)** (Category: To do)

Owner: Andrew Baker (sample)	Due: 10/5/2017 9:50 AM	Status: Not started
------------------------------	------------------------	---------------------
- Visit bpm'online Knowledge Hub (bpmonline.com/community/base/10301)** (Category: To do)

Owner: Supervisor	Due: 5/31/2017 11:45 PM	Status: Not started
-------------------	-------------------------	---------------------

Working with requests in Postman

Beginner Easy **Medium** Advanced

Introduction

Postman is a toolset for API testing. It is a development environment that lets you create, test, manage, and publish API

documentation.

Working with requests in Postman consists of the following general stages:

- Adding a request
- Setting up the request
- Executing the request
- Saving the request

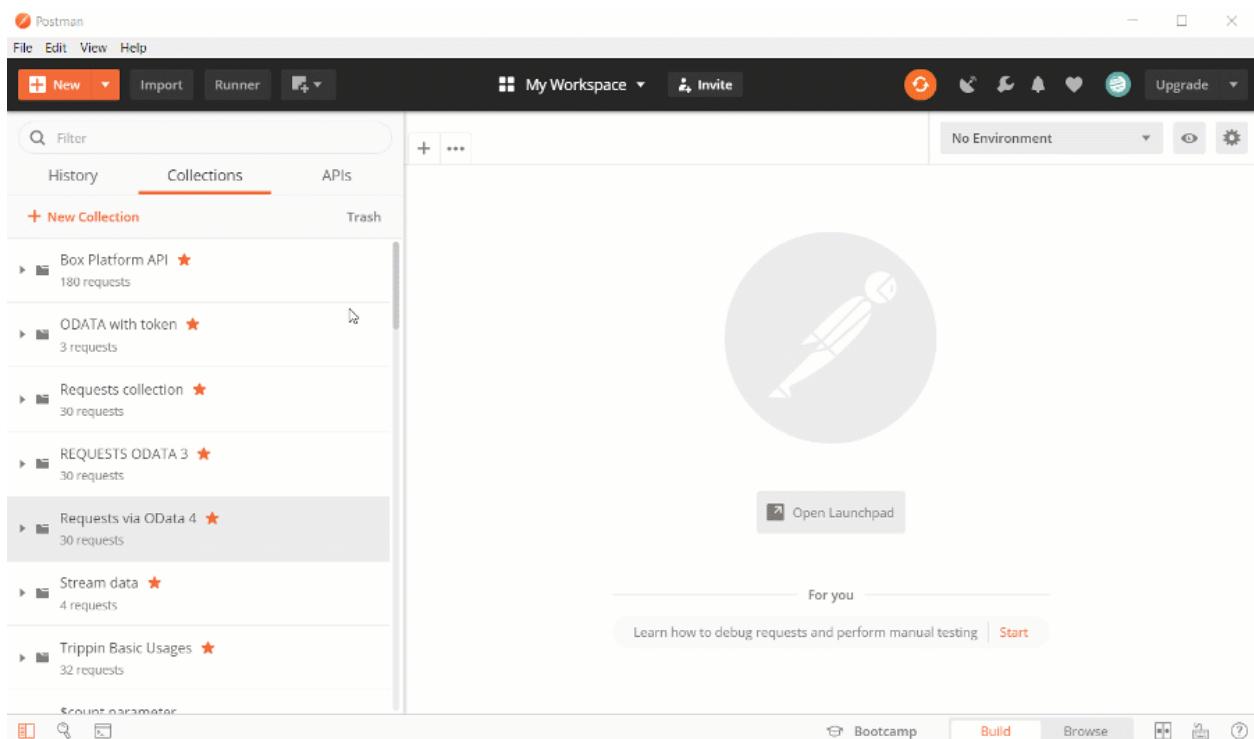
We recommend using Postman for testing queries when developing integrations with Creatio via **OData 3** or **OData 4**. More information about working with Postman is available in the [Postman documentation](#).

Working with requests

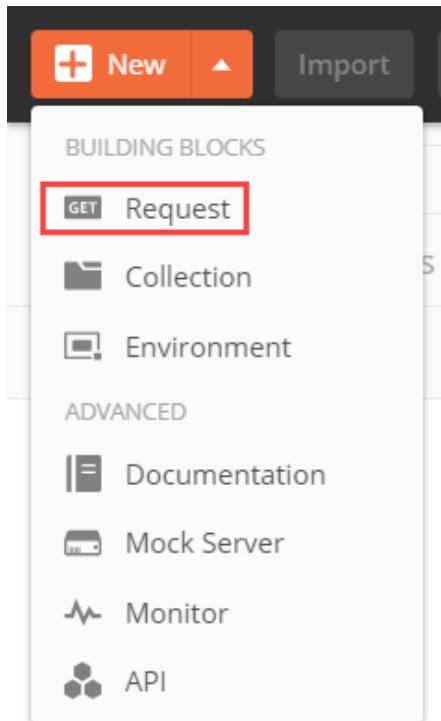
Adding a request

There are two ways of adding a request in Postman:

- Open the [Create new] tab, then in the [Building blocks] click [New] → [Request] (Fig. 1).
Fig. 1. Adding a request



- In the dropdown menu of the [New] button, click [Request] (Fig. 2).
Fig. 2. Clicking [New] → [Request]

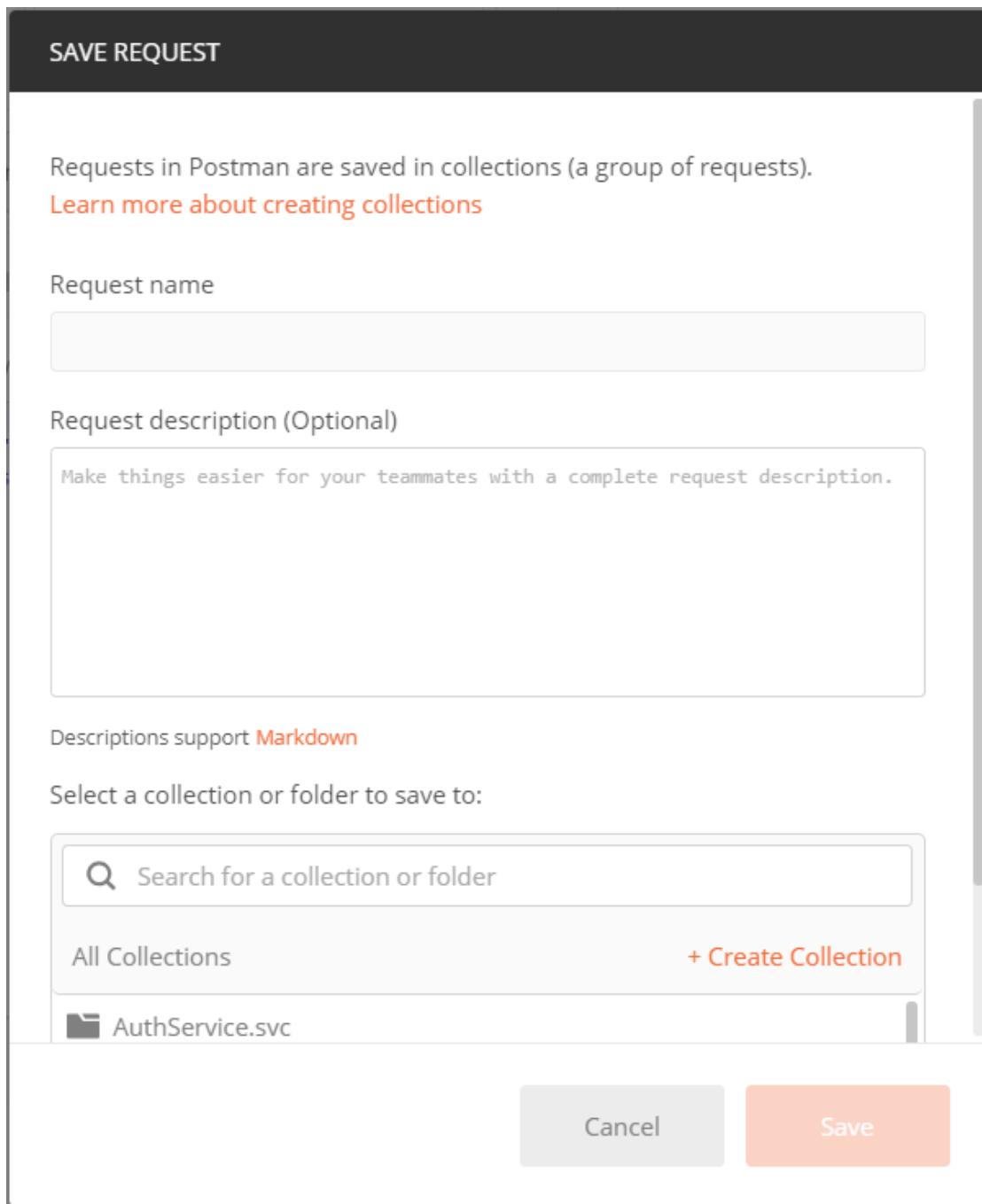


Populate the fields in the new request window (Fig. 3), see table 1:

Table 1. New request fields

Field name	Description
[Request name]	The name of the new request.
[Request description (Optional)]	Additional information about the request (optional).
[Search for a collection or folder]	Search for an earlier created collection of requests, or create a new one.

Fig. 3. New request window



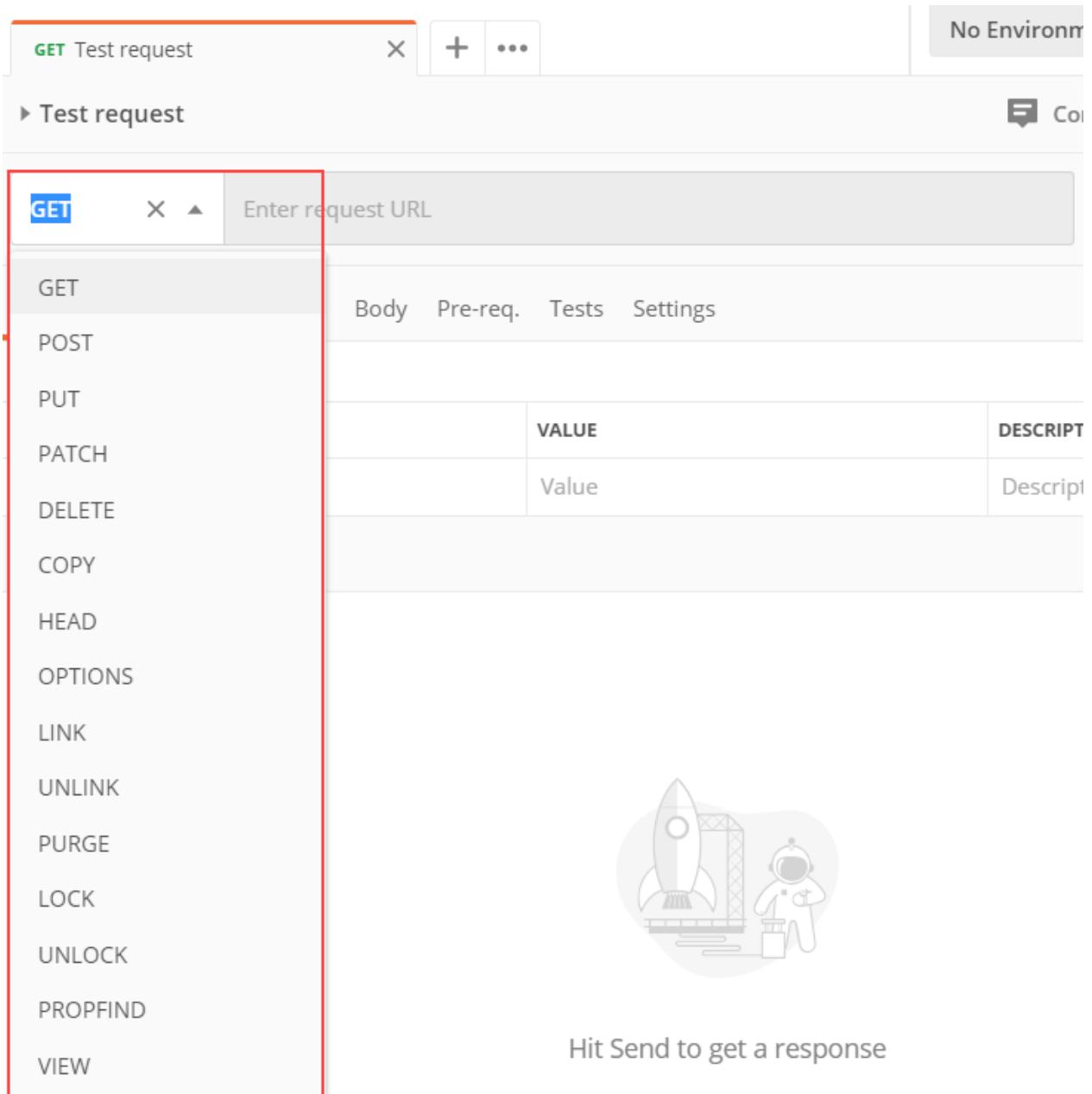
Populate the fields and click [Save]. The button becomes active only after the [Search for a collection or folder] field is populated.

Setting up the request

To set up the request:

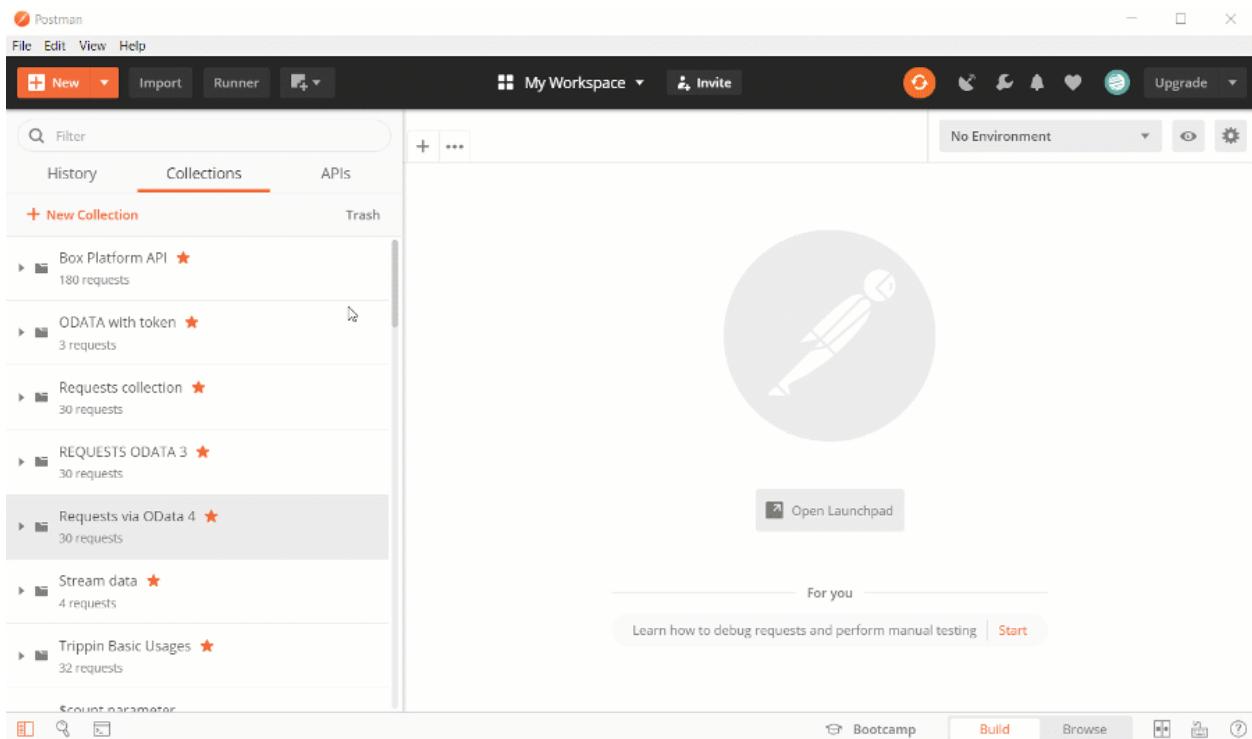
1. Select the request method (Fig. 4).

Fig. 4. Selecting the request method



The screenshot shows the Creatio Test request interface. At the top, there's a header with 'GET Test request' and buttons for 'X', '+', and '...'. To the right, it says 'No Environment' and has a 'Comment' button. Below the header, the title 'Test request' is followed by a 'Comment' icon. The main area has a red border around the left sidebar. The sidebar lists various HTTP methods: GET, POST, PUT, PATCH, DELETE, COPY, HEAD, OPTIONS, LINK, UNLINK, PURGE, LOCK, UNLOCK, PROPFIND, and VIEW. The 'GET' method is selected and highlighted in blue. To the right of the sidebar, there's a table with columns 'Body', 'Pre-req.', 'Tests', and 'Settings'. Below the table, there's a placeholder 'Enter request URL'. On the far right, there's a small circular icon with a rocket and an astronaut, and the text 'Hit Send to get a response'.

2. Enter the request string.
 3. Set the data format of the request. Go to the [Body] tab, select the “raw” option and JSON type (Fig. 5).
- Fig. 5. Select the data format of the request



4. Populate the request body for POST and PATCH methods.
5. Go to the [Headers] tab and set the following headers:

```
Accept: application/json  
Content-Type: application/json; charset=utf-8  
ForceUseSession: true
```

Executing the request

To execute a request, click [Send] (Fig. 6).

Fig. 6. Executing the request

A screenshot of the Postman request configuration screen. At the top, there is a dropdown for "Method" (set to "GET") and a "Send" button which is highlighted with a red box. Below the method dropdown is a field to "Enter request URL". Underneath the URL field are tabs for "Params", "Auth", "Headers (7)", "Body", "Pre-req.", "Tests", and "Settings". The "Headers" tab is currently selected. In the "Headers" section, there is a table with one row: "Key" (Value: "Value") and "Description" (Value: "Description"). Below the header table is a section titled "Query Params" with a table showing one row: "Key" (Value: "Value") and "Description" (Value: "Description"). At the bottom of the screen is a large "Response" panel.

Saving the request

To save a request, click [Save] (Fig. 7).

Fig. 7. Saving the request

The screenshot shows the Postman interface. At the top, there's a header with 'GET' selected, a 'Send' button, and a 'Save' button with a red border. Below the header, tabs include 'Params' (which is active), 'Auth', 'Headers (7)', 'Body', 'Pre-req.', 'Tests', 'Settings', 'Cookies', and 'Code'. Under the 'Params' tab, there's a table titled 'Query Params' with columns 'KEY', 'VALUE', and 'DESCRIPTION'. A single row is present with 'Key' and 'Value' columns both containing 'Description'. At the bottom left, there's a 'Response' section.

We recommend using Postman for testing queries when developing integrations with Creatio via **OData 3** or **OData 4**.

Working with request collections in Postman

Beginner Easy **Medium** Advanced

Introduction

Postman is a toolset for API testing. It is a development environment that lets you create, test, manage, and publish API documentation.

A collection of requests lets you execute several requests consecutively. Any collection of requests to Creatio must include a POST request to **AuthService.svc (Creatio's authentication service)** and a user request for working with data. Using collections facilitates faster testing of large sets of requests.

Working with collections of requests in Postman consists of the following general stages:

- Adding a collection of requests
- Adding requests to the collection
- Setting up variables for the request collection
- Executing the collection of requests

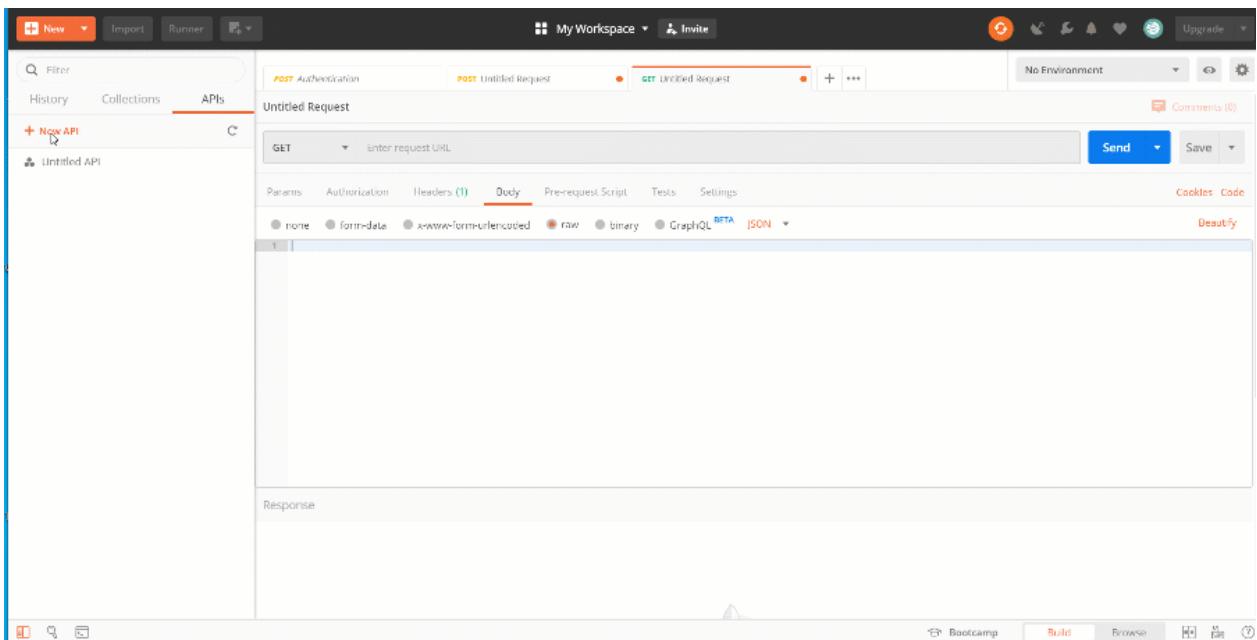
We recommend using Postman for testing queries when developing integrations with Creatio via **OData 3** or **OData 4**. More information about working with Postman is available in the [Postman documentation](#).

Working with request collections

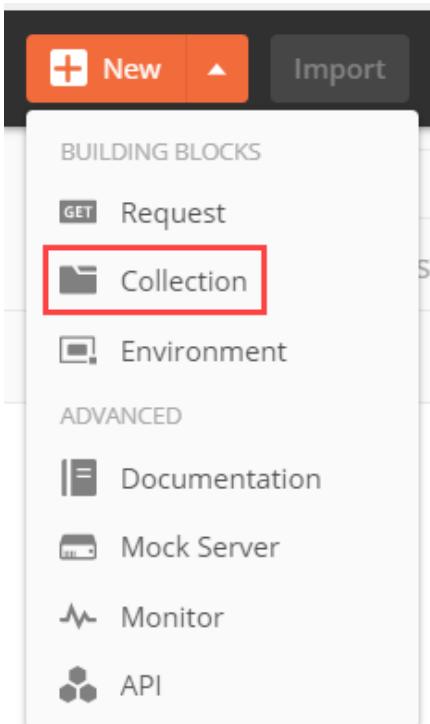
Adding a collection of requests

There are two ways of adding a collection of requests in Postman:

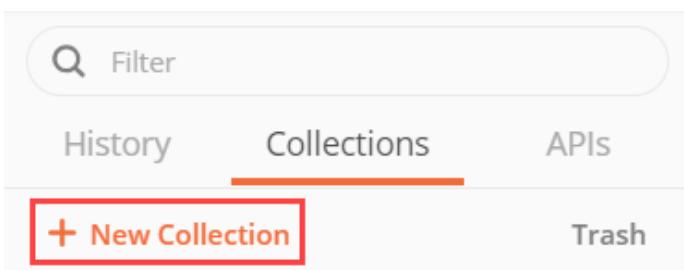
- Open the [Create new] tab, then in the [Building blocks] click [New] → [Collection] (Fig. 1).
Fig. 1. Adding a collection



- In the dropdown menu of the [New] button, click [Collection] (Fig. 2).
- Fig. 2. Clicking [New] → [Collection]



- On the [Collections] tab, click [+ New Collection] (Fig. 3).
- Fig. 3. Adding a new collection

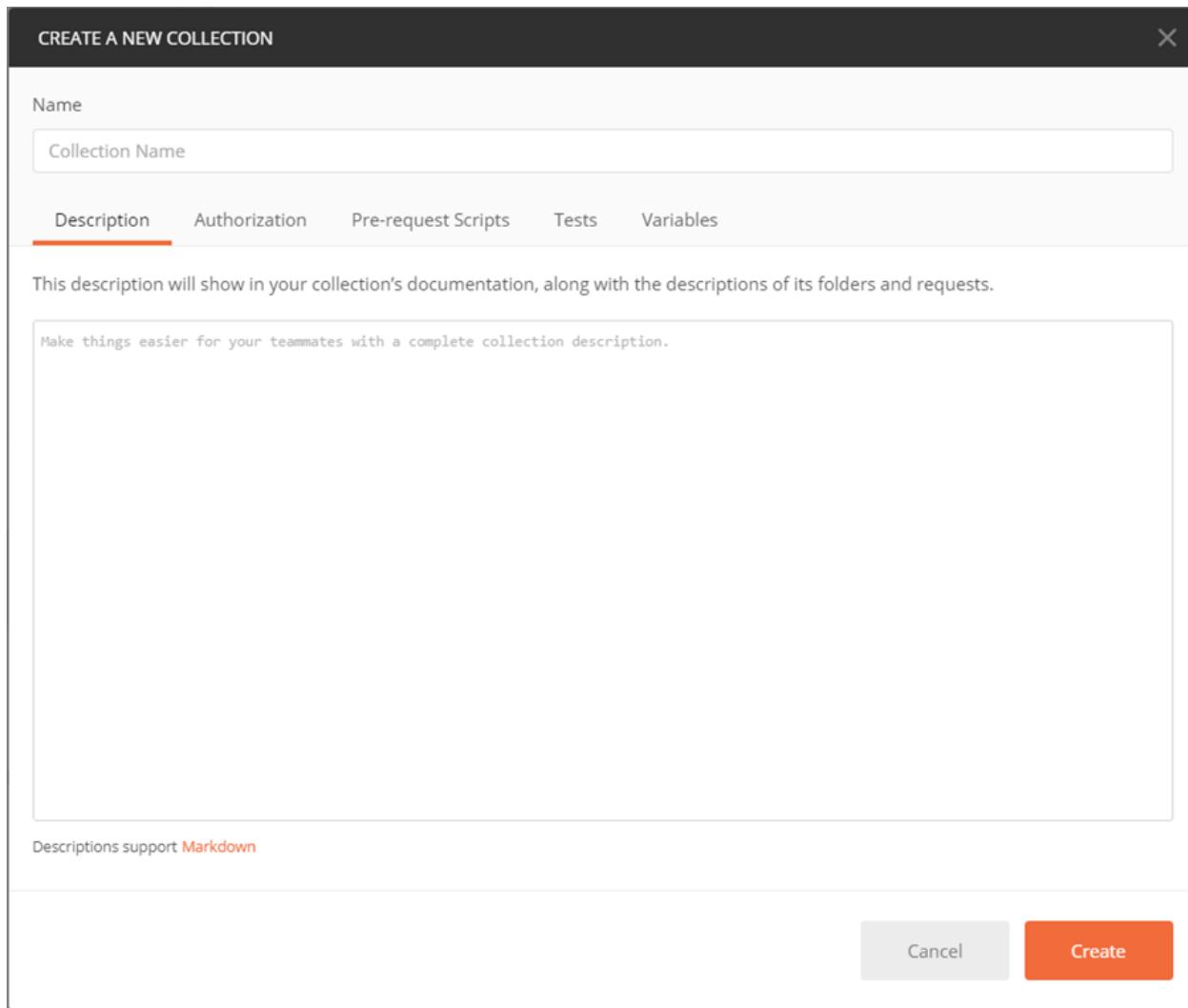


Populate the fields in the new collection window (Fig. 4), see table 1:

Table 1. Fields of the new request collection window

Field name	Description
[Name]	The name of the collection.
[Description]	The description of the collection.

Fig. 4. New collection window



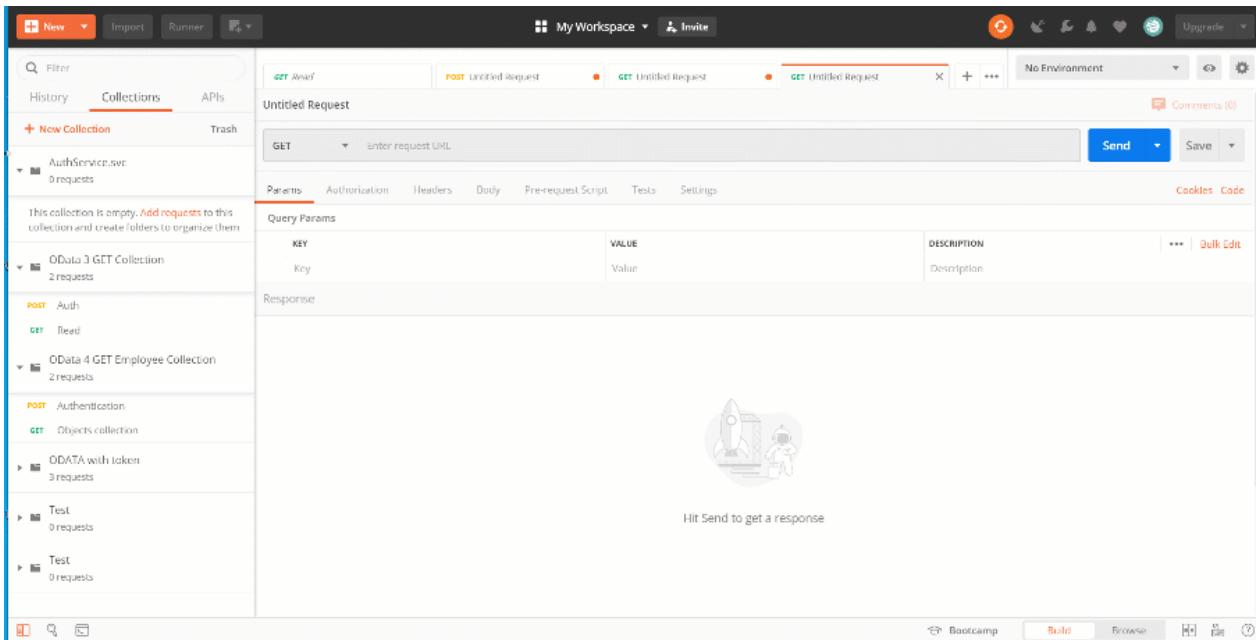
Click [Create].

Adding requests to the collection

There are two ways of adding a request to a collection in Postman:

- Drag and drop an existing request to the collection.
- Right-click the name of an earlier created collection, then click [Add Request] (Fig. 5).

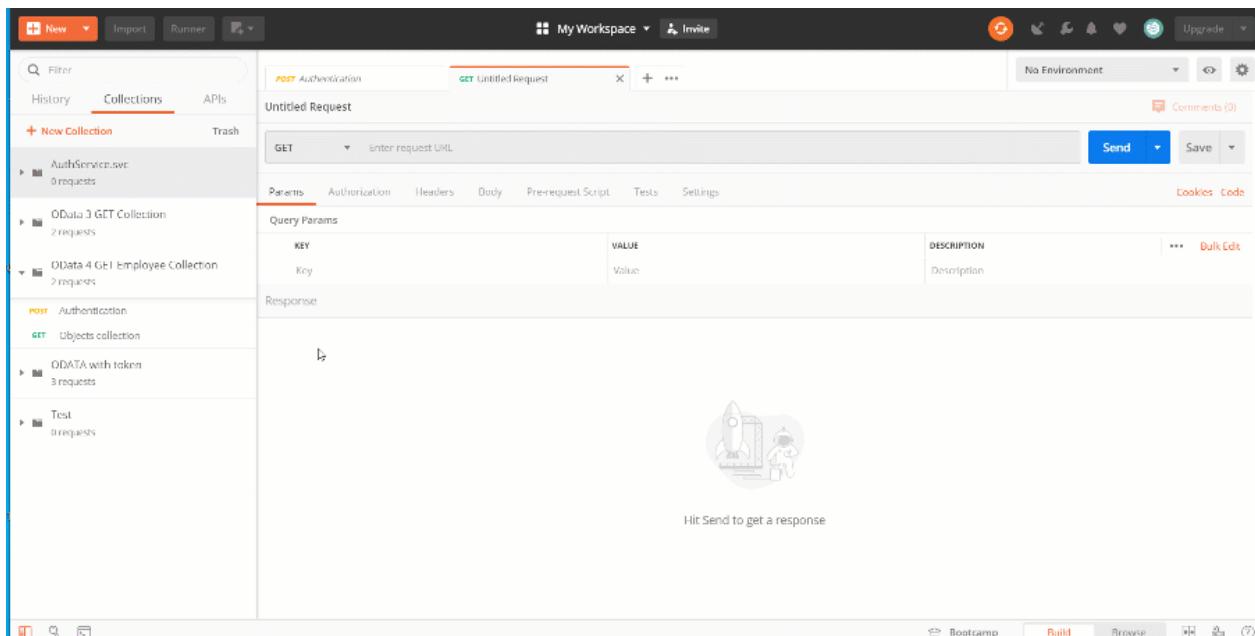
Fig. 5. Adding requests to the collection



Setting up variables for the request collection

Collections enable setting up common variables and parameters for all requests that it contains. To set the collection variables:

1. Right-click an earlier created collection.
 2. Click [Edit], then go to the [Variables] tab (Fig. 6).
- Fig. 6. Setting up common variables



3. Create the following variables for the collection (see table 2): see table 2:
Table 2. Collection variables

Variable name Description

BaseURI	Creatio application URL
UserName	Creatio application user name
UserPassword	Creatio application user password
BPMCSRF	CSRF protection token
CollectionName	Object collection (database table) name.

Variable values in the [Initial value] and [Current value] must be duplicated.

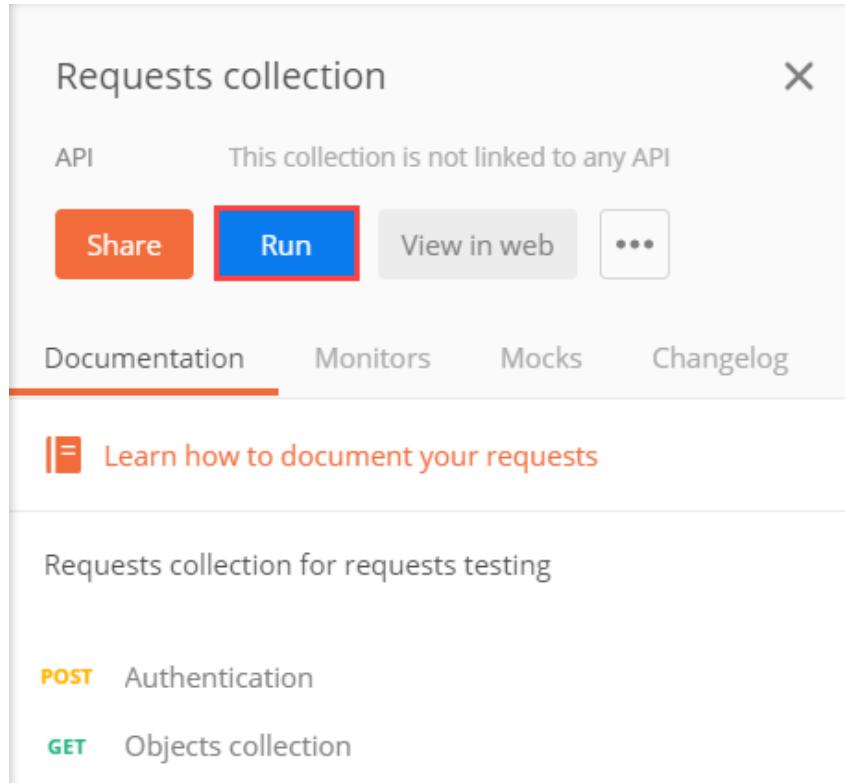
Executing the collection of requests

To execute a collection of requests:

1. Click  next to the collection name.

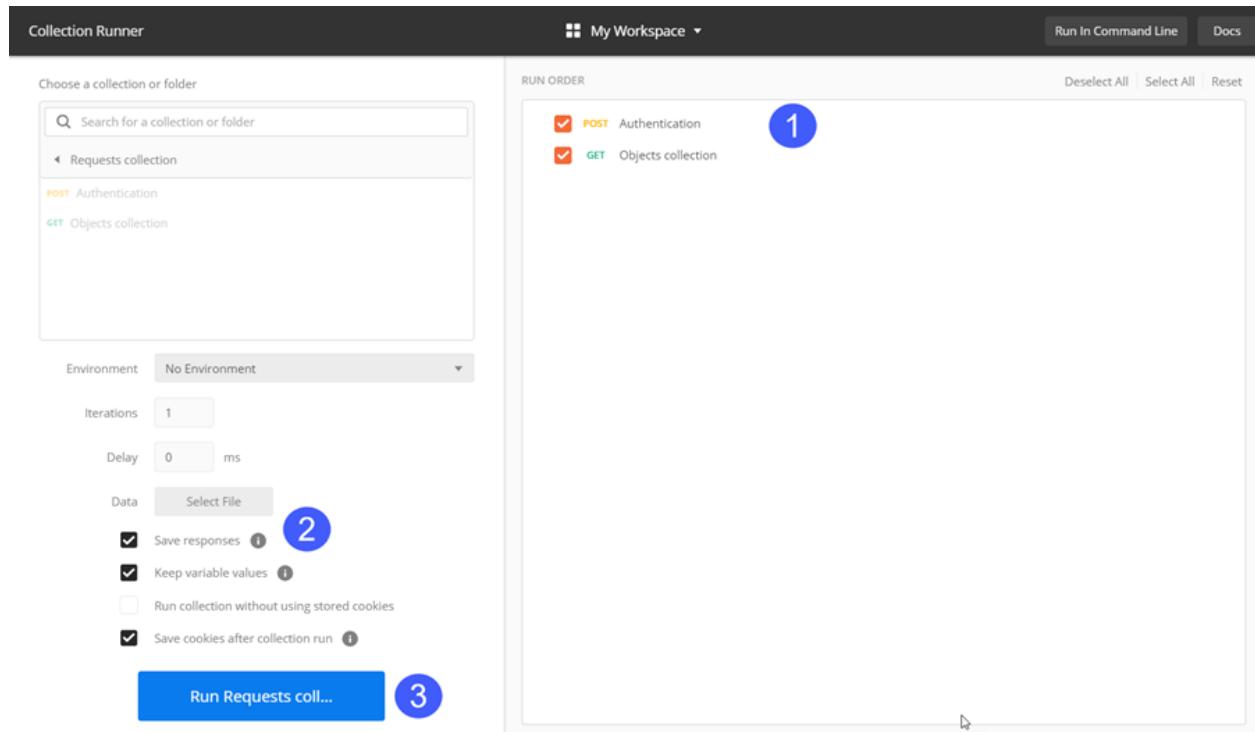
2. Click [Run].

Fig. 7. Running the collection of requests



3. In the [Run order] block, select requests to run and set their order (Fig. 8, 1).

Fig. 8. Running a collection



4. Select the [Save responses] checkbox (Fig. 8, 2).
5. Click [Run] (Fig. 8, 3).

Getting data of the request structure elements and the response:

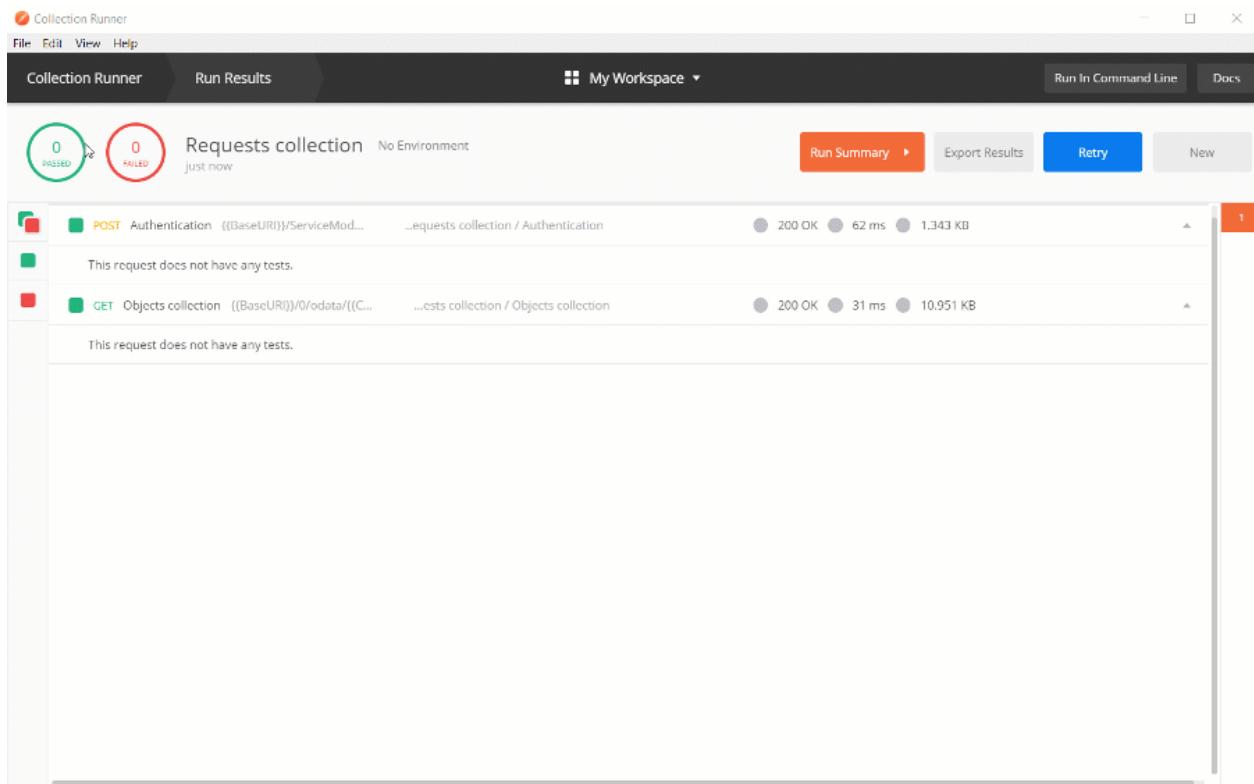
- Click the request name (Fig. 9).

Fig. 9. Request structure elements

Request Type	Request URL	Request Headers	Response Headers	Response Body	Status	Time	Size
POST	Authentication	(11)	(12)		200 OK	171 ms	1.343 KB
GET	Objects collection				200 OK	35 ms	10.951 KB

- Click [View] → [Show Postman Console] (Fig. 10) or press “Alt” + “Ctrl” + “C” on the keyboard.

Fig. 10. Opening the console



Select the needed item.

OData

Contents

- **OData integration examples**
- **Working with Creatio objects over the OData protocol WCF-client**
- **Working with Stream data ('Working with Creatio objects over the OData protocol using Http request' in the on-line documentation)**
- **Working with batch-requests**

OData integration examples

Examples of the different CRUD-operations by OData 3 and OData 4 protocols is available on the separate web-resource <https://documenter.getpostman.com/view/10204500/SztHX5Qb?version=latest>

Working with Creatio objects over the OData protocol WCF-client

Beginner

Easy

Medium

Advanced

General

The access to Creatio entities over the *OData* protocol is provided by the *EntityDataService.svc* web-service:

Address of the OData service

```
http[s]://<server_name>/<Creatio_application_name> +  
"/0/ServiceModel/EntityDataService.svc/"
```

Example of the OData service address

<http://myserver.com/creatioWebApp/0/ServiceModel/EntityDataService.svc>

Generating proxy classes of the EntityDataService.svc service

General

The key point in the organization of the WCF client operation is receiving metadata of the service and creating client proxy classes. A client application will use these mediation classes to exchange data with the web-service.

The following must be performed to implement the .NET client application that could work with the *OData* service of Creatio:

1. Create a .NET project where the integration with Creatio will be implemented.
2. Generate client proxy classes of the *EntityDataService.svc* service.
3. Create an instance of the execute environment for the *EntityDataService.svc* service.
4. Implement the client business logic of integration using the methods of the created proxy class instance.

Proxy classes may be generated on the client by several methods considered below.

Creating proxy classes using the DataServiceModel Metadata Utility Tool (DataSvcutil.exe) utility

DataSvcUtil.exe is a command string program provided by *WCF Data Services services*. The program uses the *OData* channel and forms client classes of the data service required for access to the data service from the .NET Framework client application. This program forms data classes using the following sources of metadata:

- WSDL – a document of metadata of the service describing the data model provided by the data service.
- Data model file (CSDL) determined using the language for determining the conceptual schema (CSDL) described in the specification [\[MC-CSDL\]: format of the file determining the conceptual schema](#).
- EDMX file created with the help of programs for work with the EDM model being a part of *Entity Framework*. Additional data are given in the specification [\[MC-EDMX\]: EDM model for the data service package format](#).

The *DataSvcUtil.exe* program is installed in the .NET Framework catalogue.

This is usually the folder *C:\Windows\Microsoft.NET\Framework\v4.0*. For 64-bit system versions this is the folder *C:\Windows\Microsoft.NET\Framework64\v4.0*.

Format of calling the DataSvcutil.exe utility

```
datasvcutil /out:file [/in:file | /uri:serviceuri] [/dataservicecollection]
[/language:devlang] [/nologo] [/version:ver] [/help]
```

The *DataSvcutil.exe* utility is covered in the [corresponding section of the MSDN](#).

1.

Visual Studio may generate proxy classes of the service from the service metadata file saved on the disk. For this purpose, fulfill part. 1 of the instructions. Then, enter the full path to the metadata file with the prefix "file://" in the *Address* dialogue window.

Example: file:///C:/metadata.xml"

After this, fulfill part. 3–5 of the instructions.

Examples of working with Creatio entities in the WCF client

When proxy classes of the service are generated, a link to the *Microsoft.Data.Services.Client.dll* assembly is added to the project. This assembly supports the *OData v.3* protocol. If, by any reason, the earlier version of the protocol must be used in the client application, a link to the corresponding assembly must be added manually.

This client library allows making requests to the *EntityDataService.svc* data service using standard software templates .NET Framework, and include the use of the *LINQ* request language.

To ensure successful compilation of the examples below, the following must be added to the project code:

Using directives

```
using System;
using System.Data.Services.Client;
using System.Net;
using Terrasoft.Sdk.Examples.CreatioServiceReference;
using System.Linq;
```

Declaring the variable of the OData service address

```
private static Uri serverUri = new
Uri("http://<server_name>/<application_name>/0/ServiceModel/EntityDataService.svc/");
```

Receiving the objects collection of the service

To receive the objects collection of the service, the *DataServiceQuery* universal class is used. This class represents a request to the service, which returns the collection of a certain type of entities.

To implement the request to the *EntityDataService.svc* data service, an instance of the environment context object for the Creatio must be created.

One ought to bear in mind that all external requests to Creatio web-services must be authenticated. Detailed methods of authentication may be found in the article **External requests authentication to Creatio services (on-line documentation)**.

Forms authentication implemented on the basis of the example in the clause above will be in further examples.

To implement the forms authentication, the *LoginClass* class with *authServiceUri* fields (string of a request to the *Login* method of the *AuthService.svc* authenticated service) and *AuthCookie* (Cookie authentications of Creatio) were created. The *TryLogin(string userName, string userPassword)* method implementing the user authentication and saving the server response in the *AuthCookie* field may also be used.

In addition, the *OnSendingRequestCookie* (*object sender, SendingEventArgs e*) method is created. The method will be called in response to an event of the *SendingRequest* context instance (creating a new *HttpWebRequest* instance).

The *OnSendingRequestCookie* method authenticates the user and Cookies received in response are added to the data receipt request.

```
static void OnSendingRequestCookie(object sender, SendingEventArgs e)
{
    // Calling the method of the LoginClass class, which authenticates the user
    // method transmitted in parameters.
    LoginClass.TryLogin("CreatioUserName", "CreatioUserPassword");
    var req = e.Request as HttpWebRequest;
    // Adding pre-received authentication cookie to the data receipt request.
    req.CookieContainer = LoginClass.AuthCookie;
    e.Request = req;
}
```

A request to the service may be implemented by one of the following methods:

- Implementing a *LINQ* request to the *DataServiceQuery* named object received from the service context.
- Implicit listing of the *DataServiceQuery* object received from the service context.
- Explicit call of the *Execute* method of the *DataServiceQuery* or *BeginExecute* object for asynchronous execution.

Below are the examples of access to *EntityDataService.svc* objects by one of the above methods.

1) Example of receiving the contacts collection via a LINQ request

This example demonstrates how the *LINQ* request returning all contact entities of the *EntityDataService.svc* service must be determined and implemented.

```
public static void GetOdataCollectioByLinqWcfExample()
{
```

```
// Creating the context of the Creatio application.
var context = new Creatio(serverUri);
// Determining the method which adds authentication cookie when creating a new
request.
context.SendingRequest += new EventHandler<SendingRequestEventArgs>
(OnSendingRequestCookie);
try
{
    // Building a LINQ request to receive the contacts collection.
    var allContacts = from contacts in context.ContactCollection
                      select contacts;
    foreach (Contact contact in allContacts)
    {
        // Implementing actions with contacts.
    }
}
catch (Exception ex)
{
    // Error processing.
}
```

2) Example of receiving the contacts collection via an implicit request to the OData service via the context object

This example demonstrates how the context must be used to implement the implicit request returning all contact entities of the EntityDataService.svc service.

```
public static void GetOdataCollectionByImplicitRequestExample()
{
    // Creating a context object of the Creatio application.
    var context = new Creatio(serverUri);
    // Determining the method which adds authentication cookie when creating a new
request.
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>
(OnSendingRequestCookie);
    try
    {
        // Determining an implicit request to the service to receive the contacts
collection.
        DataServiceQuery<Contact> allContacts = context.ContactCollection;
        foreach (Contact contact in allContacts)
        {
            // Implementing actions with contacts.
        }
    }
    catch (Exception ex)
    {
        // Error processing.
    }
}
```

3) Example of receiving the contacts collection via an explicit request to the OData service via the context object

This example demonstrates how the *DataServiceContext* context must be used to implement a request to the *EntityDataService.svc* service returning all contact entities.

```
public static void GetOdataCollectionByExplicitRequestExample()
{
    // Determining a Uri request to the service which returns the contacts
```

```
collection.  
    Uri contactUri = new Uri(serverUri, "ContactCollection");  
    // Creating a context object of the Creatio application.  
    var context = new Creatio(serverUri);  
    // Determining the method which adds authentication cookie when creating a new  
request.  
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>  
(OnSendingRequestCookie);  
    try  
    {  
        // Implementing an explicit request to the service by calling the Execute<>()  
method.  
        foreach (Contact contact in context.Execute<Contact>(contactUri))  
        {  
            // Implementing actions with contacts.  
        }  
    }  
    catch (Exception ex)  
    {  
        // Error processing.  
    }  
}
```

Receiving an object with set features

```
public static void GetOdataObjectByWcfExample()  
{  
    // Creating the context of the Creatio application.  
    var context = new Creatio(serverUri);  
    // Determining the method which adds authentication cookie when creating a new  
request.  
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>  
(OnSendingRequestCookie);  
    var contact = context.ContactCollection.Where(c =>  
c.Name.Contains("User")).First();  
    // Implementing actions over the contact.  
}
```

Creating a new object

```
public static void CreateCreatioEntityByOdataWcfExample()  
{  
    // Creating a new contact, initiating properties.  
    var contact = new Contact()  
    {  
        Id = Guid.NewGuid(),  
        Name = "New Test User"  
    };  
    // Creating and initiating properties of a new account, to which the created  
contact refers.  
    var account = new Account()  
    {  
        Id = Guid.NewGuid(),  
        Name = "Some Company"  
    };  
    contact.Account = account;  
    // Creating the context of the Creatio application.  
    var context = new Creatio(serverUri);  
    // Determining the method which adds authentication cookie when creating a new  
request.  
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>  
(OnSendingRequestCookie);
```

```
// Adding the created contact to the contacts collection of the service data model.  
context.AddToAccountCollection(account);  
// Adding the created account to the accounts collection of the service data model.  
context.AddToContactCollection(contact);  
// Setting the relationship between the created contact and account in the service data model.  
context.SetLink(contact, "Account", account);  
// Saving the modification of data in Creatio by one request.  
DataServiceResponse responses = context.SaveChanges(SaveChangesOptions.Batch);  
// Processing the server responses.  
}
```

Modifying an existing object

```
public static void UpdateCreatioEntityByOdatetWcfExample()  
{  
    // Creating the context of the Creatio application.  
    var context = new Creatio(serverUri);  
    // Determining the method which adds authentication cookie when creating a new request.  
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>(OnSendingRequestCookie);  
    // The contact on which basis the data will be modified is selected from the contacts collection.  
    var updateContact = context.ContactCollection.Where(c =>  
c.Name.Contains("Test")).First();  
    // Modifying the selected contact properties.  
    updateContact.Notes = "New updated description for this contact.";  
    updateContact.Phone = "123456789";  
    // Saving the modifications in the service data model.  
    context.UpdateObject(updateContact);  
    // Saving the modification of data in Creatio by one request.  
    var responses = context.SaveChanges(SaveChangesOptions.Batch);  
}
```

Deleting an object

```
public static void DeleteCreatioEntityByOdataWcfExample()  
{  
    // Creating the context of the Creatio application.  
    var context = new Creatio(serverUri);  
  
    context.SendingRequest += new EventHandler<SendingRequestEventArgs>(OnSendingRequestCookie);  
    // The object which will be deleted is selected from the contacts collection.  
    var deleteContact = context.ContactCollection.Where(c =>  
c.Name.Contains("Test")).First();  
    // Deleting the selected object from the service data model.  
    context.DeleteObject(deleteContact);  
    // Saving the modification of data in Creatio by one request.  
    var responses = context.SaveChanges(SaveChangesOptions.Batch);  
    // Processing the server responses.  
}
```

Working with Stream data

Beginner

Easy

Medium

Advanced

Introduction

Creatio 7.16.0 and up supports OData 4 integrations that use data of the Stream type.

The Stream data includes the following elements:

- Images
- Files
- BLOB

Working with data of the Stream type enables:

- Reading data
- Modifying data
- Deleting data

OData 4 use cases and examples of different types of requests are available in the “**Creatio integration via the OData 4 protocol**” article.

Working with Stream data

You need to **authenticate** your requests to Creatio.

Reading data

A request to read the value in a field by an object instance Id

```
// Get instance of an object with Id 44C083FC-2060-4B33-823D-F3D749396217 of the SysImage collection.  
GET http://mycreatio.com/0/odata/SysImage(44C083FC-2060-4B33-823D-F3D749396217)/Data  
  
ForceUseSession: true  
BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO
```

Response to the request to read the value in a field by an object instance Id

```
Status: 200 OK
```



Modifying data

A request to modify the value in a field of a collection object instance

```
// Modify instance of an object with Id 44C083FC-2060-4B33-823D-F3D749396217 of the SysImage collection.  
PUT http://mycreatio.com/0/odata/SysImage(44C083FC-2060-4B33-823D-F3D749396217)/Data
```

Accept: application/json; odata=verbose
Content-Type: application/json; odata=verbose
ForceUseSession: true
BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO

The request body is of the BLOB type and contains the following image.



A response to the request to modify the value in a field of a collection object instance

Status: 200 OK

Deleting data

A request to delete the value in a field by an object instance Id

```
// Delete object instance with Id 44C083FC-2060-4B33-823D-F3D749396217 of the SysImage collection.  
DELETE http://mycreatio.com/0/odata/SysImage(44C083FC-2060-4B33-823D-F3D749396217)/Data
```

ForceUseSession: true
BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO

A response to the request to delete the value in a field by an object instance Id

Status: 204 No Content

Working with batch-requests

Beginner

Easy

Medium

Advanced

Introduction

Creatio 7.16.1 and up supports OData 4 integrations that use batch-requests.

Batch-requests enable combining multiple HTTP requests in one by specifying each request as a separate object in the request body. The server returns one HTTP response that contains responses to each of the passed requests.

Using batch-requests enables you to improve the add-on performance by combining multiple requests in one batch request to the server and one response.

To execute batch-requests, use the POST HTTP method and the [\\$batch](#) parameter.

Creatio supports the following caption types for *Content-Type*:

- *application/json*
- *multipart/mixed*

Using the *application/json* value for *Content-Type* sets a single type of the content returned by the server to all batch-requests. The *multipart/mixed* value of *Content-Type* enables specifying a separate *Content-Type* caption for each request in the batch-request body.

If one of the requests completes with a 4xx-5xx group response code, execution of the following requests interrupts. To continue the execution of the following requests, add the [Prefer: continue-on-error](#) caption to the main HTTP request.

Maximum number of requests in a batch-request - 100.

OData 4 use cases, examples of different types of requests, as well as the server response codes are available in the “[Creatio integration via the OData 4 protocol](#)” article.

Working with batch-requests

You need to **authenticate** your requests to Creatio.

Batch-request (*Content-Type: application/json*)

Batch-request

```
POST http://mycreatio.com/0/odata/$batch
```

```
Content-Type: application/json; odata=verbose
```

```
Accept: application/json
```

```
BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO
```

```
ForceUseSession: true
```

Batch-request body

```
{
  "requests": [
    {
      // Add an object instance of the City collection.
      "method": "POST",
```

```
        "url": "City",
        "id": "t3",
        "body": {
            // Add the Burbank value in the Name field.
            "Name": "Burbank"
        },
        "headers": {
            "Content-Type": "application/json;odata=verbose",
            "Accept": "application/json;odata=verbose",
            "Prefer": "continue-on-error"
        }
    },
    {
        // Add an object instance of the City collection.
        "method": "POST",
        "atomicityGroup": "g1",
        "url": "City",
        "id": "t3",
        "body": {
            // Add the 62f9bc01-57cf-4cc7-90bf-8672acc922e3 value in
            // the Id field.
            "Id": "62f9bc01-57cf-4cc7-90bf-8672acc922e3",
            // Add the Spokane value in the Name field.
            "Name": "Spokane"
        },
        "headers": {
            "Content-Type": "application/json;odata=verbose",
            "Accept": "application/json;odata=verbose",
            "Prefer": "continue-on-error"
        }
    },
    {
        // Change the object instance of the City collection with the
        // 62f9bc01-57cf-4cc7-90bf-8672acc922e3 Id.
        "method": "PATCH",
        "atomicityGroup": "g1",
        "url": "City/62f9bc01-57cf-4cc7-90bf-8672acc922e3",
        "id": "t2",
        "body": {
            // Change the Name field value for Texas.
            "Name": "Texas"
        },
        "headers": {
            "Content-Type": "application/json;odata=verbose",
            "Accept": "application/json;odata=verbose",
            "Prefer": "continue-on-error"
        }
    }
]
```

Response to the batch-request

Status: 200 OK

```
{  
    "responses": [  
        {  
            "id": "t3",  
            "status": 201,  
            "headers": {  
                "location":  
                    "https://mycreatio.com/0/odata/City(b6a05348-55b1-4314-a228-  
436ba305d2f3)",  
                "content-type": "application/json;  
odata.metadata=minimal",  
                "odata-version": "4.0"  
            },  
            "body": {  
                "@odata.context":  
                    "https://mycreatio.com/0/odata/$metadata#City/$entity",  
                "Id": "b6a05348-55b1-4314-a228-436ba305d2f3",  
                "CreatedOn": "2020-05-18T17:50:39.2838673Z",  
                "CreatedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",  
                "ModifiedOn": "2020-05-18T17:50:39.2838673Z",  
                "ModifiedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",  
                "Name": "Burbank",  
                "Description": "",  
                "CountryId": "00000000-0000-0000-0000-000000000000",  
                "RegionId": "00000000-0000-0000-0000-000000000000",  
                "TimeZoneId": "00000000-0000-0000-0000-000000000000",  
                "ProcessListeners": 0  
            }  
        },  
        {  
            "id": "t3",  
            "atomicityGroup": "c59e36b2-2aa9-44fa-86d3-e7d68eecbfa0",  
            "status": 201,  
            "headers": {  
                "location":  
                    "https://mycreatio.com/0/odata/City(62f9bc01-57cf-4cc7-90bf-  
8672acc922e3)",  
                "content-type": "application/json;  
odata.metadata=minimal",  
                "odata-version": "4.0"  
            },  
            "body": {  
                "@odata.context":  
                    "https://mycreatio.com/0/odata/$metadata#City/$entity",  
                "Id": "62f9bc01-57cf-4cc7-90bf-8672acc922e3",  
                "CreatedOn": "2020-05-18T17:50:39.361994Z",  
                "CreatedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",  
                "ModifiedOn": "2020-05-18T17:50:39.361994Z",  
            }  
        }  
    ]  
}
```

```
        "ModifiedById": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",
        "Name": "Spokane",
        "Description": "",
        "CountryId": "00000000-0000-0000-0000-000000000000",
        "RegionId": "00000000-0000-0000-0000-000000000000",
        "TimeZoneId": "00000000-0000-0000-0000-000000000000",
        "ProcessListeners": 0
    }
},
{
    "id": "t2",
    "atomicityGroup": "c59e36b2-2aa9-44fa-86d3-e7d68eecbfa0",
    "status": 204,
    "headers": {
        "odata-version": "4.0"
    }
}
]
```

Batch-request (Content-Type: application/json and Prefer: continue-on-error)

Batch-request

```
POST http://mycreatio.com/0/odata/$batch
```

```
Content-Type: application/json; odata=verbose
Accept: application/json
BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO
ForceUseSession: true
Prefer: continue-on-error
```

Batch-request body

```
{
    "requests": [
        {
            // Add an object instance of the City collection.
            "method": "POST",
            "url": "City",
            "id": "t3",
            "body": {
                // Add the Burbank value in the Name field.
                "Name": "Burbank"
            },
            "headers": {
                "Content-Type": "application/json;odata=verbose",
                "Accept": "application/json;odata=verbose",
                "Prefer": "continue-on-error"
            }
        },
        {
            // Add an object instance of the City collection.
            "method": "POST",
            "url": "City",
            "id": "t4",
            "body": {
                // Add the Seattle value in the Name field.
                "Name": "Seattle"
            },
            "headers": {
                "Content-Type": "application/json;odata=verbose",
                "Accept": "application/json;odata=verbose",
                "Prefer": "continue-on-error"
            }
        }
    ]
}
```

```
// Change the object instance of the City collection with the
62f9bc01-57cf-4cc7-90bf-8672acc922e2 Id.
    "method": "PATCH",
    "atomicityGroup": "g1",
    "url": "City/62f9bc01-57cf-4cc7-90bf-8672acc922e2",
    "id": "t2",
    "body": {
        // Change the Name field value for Indiana.
        "Name": "Indiana"
    },
    "headers": {
        "Content-Type": "application/json;odata=verbose",
        "Accept": "application/json;odata=verbose",
        "Prefer": "continue-on-error"
    }
},
{
    // Add an object instance of the City collection.
    "method": "POST",
    "atomicityGroup": "g1",
    "url": "City",
    "id": "t3",
    "body": {
        // Add the 62f9bc01-57cf-4cc7-90bf-8672acc922a1 value in
the Id field.
        "Id": "62f9bc01-57cf-4cc7-90bf-8672acc922a1",
        // Add the Iowa value in the Name field.
        "Name": "Iowa"
    },
    "headers": {
        "Content-Type": "application/json;odata=verbose",
        "Accept": "application/json;odata=verbose",
        "Prefer": "continue-on-error"
    }
}
]
```

Response to the batch-request

Status: 200 OK

```
{
    "responses": [
        {
            "id": "t3",
            "status": 201,
            "headers": {
                "location":
"https://mycreatio.com/0/odata/City(2f5e68f8-38bd-43c1-8e15-
a2f13b0aa56a)",
                "content-type": "application/json;
            }
        }
    ]
}
```

```
odata.metadata=minimal",
    "odata-version": "4.0"
},
"body": {
    "@odata.context":
"https://mycreatio.com/0/odata/$metadata#City/$entity",
    "Id": "2f5e68f8-38bd-43c1-8e15-a2f13b0aa56a",
    "CreatedOn": "2020-05-18T18:06:50.7329808Z",
    "CreatedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",
    "ModifiedOn": "2020-05-18T18:06:50.7329808Z",
    "ModifiedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",
    "Name": "Burbank",
    "Description": "",
    "CountryId": "00000000-0000-0000-000000000000",
    "RegionId": "00000000-0000-0000-000000000000",
    "TimeZoneId": "00000000-0000-0000-0000-000000000000",
    "ProcessListeners": 0
}
},
{
    "id": "t2",
    "atomicityGroup": "0c1c4019-b9fb-4fb3-8642-2d0660c4551a",
    "status": 204,
    "headers": {
        "odata-version": "4.0"
    }
},
{
    "id": "t3",
    "atomicityGroup": "0c1c4019-b9fb-4fb3-8642-2d0660c4551a",
    "status": 201,
    "headers": {
        "location":
"https://mycreatio.com/0/odata/City(62f9bc01-57cf-4cc7-90bf-
8672acc922a1)",
        "content-type": "application/json;
odata.metadata=minimal",
        "odata-version": "4.0"
    },
    "body": {
        "@odata.context":
"https://mycreatio.com/0/odata/$metadata#City/$entity",
        "Id": "62f9bc01-57cf-4cc7-90bf-8672acc922a1",
        "CreatedOn": "2020-05-18T18:06:50.7954775Z",
        "CreatedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",
        "ModifiedOn": "2020-05-18T18:06:50.7954775Z",
        "ModifiedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",
        "Name": "Iowa",
        "Description": "",
        "CountryId": "00000000-0000-0000-000000000000",
        "RegionId": "00000000-0000-0000-0000-000000000000",
```

```
        "TimeZoneId": "00000000-0000-0000-0000-000000000000",
        "ProcessListeners": 0
    }
}
]
```

Batch-request (Content-Type: multipart/mixed)

Batch-request

```
POST http://mycreatio.com/0/odata/$batch

Content-Type: multipart/mixed;boundary=batch_a685-9724-d873
BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO
ForceUseSession: true
```

Batch-request body

```
--batch_a685-9724-d873
Content-Type: multipart/mixed; boundary=changeset_06da-d998-8e7e

--changeset_06da-d998-8e7e
Content-Type: application/http
Content-Transfer-Encoding: binary

// Add an object instance of the City collection.
POST City HTTP/1.1
Content-ID: 1
Accept: application/atomsrv+xml;q=0.8,
application/json;odata=verbose;q=0.5, */*;q=0.1
Content-Type: application/json;odata=verbose

// Add the Gilbert value in the Name field.
{"Name": "Gilbert"}

--changeset_06da-d998-8e7e
Content-Type: application/http
Content-Transfer-Encoding: binary

// Change the object instance of the City collection with the 62f9bc01-
57cf-4cc7-90bf-8672acc922e2 Id.
PATCH City/62f9bc01-57cf-4cc7-90bf-8672acc922e2 HTTP/1.1
Content-ID: 2
Accept: application/atomsrv+xml;q=0.8,
application/json;odata=verbose;q=0.5, */*;q=0.1
Content-Type: application/json;odata=verbose

// Change the Name field value for Lincoln.
{"Name": "Lincoln"}

--changeset_06da-d998-8e7e
```

```
Content-Type: application/http  
Content-Transfer-Encoding: binary
```

```
// Delete the object instance of the City collection with the 62f9bc01-  
57cf-4cc7-90bf-8672acc922e2 Id.  
DELETE City/62f9bc01-57cf-4cc7-90bf-8672acc922e2 HTTP/1.1  
Content-ID: 3  
Accept: application/atomsvc+xml; q=0.8,  
application/json; odata=verbose; q=0.5, */*; q=0.1  
Content-Type: application/json; odata=verbose
```

Response to the batch-request

```
Status: 200 OK
```

```
--batchresponse_e17aace9-5cbb-49bd-b7ad-f1be8cc8c9d8  
Content-Type: multipart/mixed; boundary=changesetresponse_a08c1df6-  
4b82-4a9b-be61-7ef4cc7b23ba
```

```
--changesetresponse_a08c1df6-4b82-4a9b-be61-7ef4cc7b23ba  
Content-Type: application/http  
Content-Transfer-Encoding: binary  
Content-ID: 1
```

```
HTTP/1.1 201 Created  
Location: https://mycreatio.com/0/odata/City(fbd0565f-fa8a-4214-ae89-  
c976c5f3acb4)  
Content-Type: application/json; odata.metadata=minimal  
OData-Version: 4.0
```

```
{  
    "@odata.context":  
    "https://mycreatio.com/0/odata/$metadata#City/$entity",  
    "Id": "fbd0565f-fa8a-4214-ae89-c976c5f3acb4",  
    "CreatedOn": "2020-05-18T18:41:57.091Z",  
    "CreatedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",  
    "ModifiedOn": "2020-05-18T18:41:57.091Z",  
    "ModifiedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",  
    "Name": "Gilbert",  
    "Description": "",  
    "CountryId": "00000000-0000-0000-0000-000000000000",  
    "RegionId": "00000000-0000-0000-0000-000000000000",  
    "TimeZoneId": "00000000-0000-0000-0000-000000000000",  
    "ProcessListeners": 0
}
```

```
--changesetresponse_a08c1df6-4b82-4a9b-be61-7ef4cc7b23ba  
Content-Type: application/http  
Content-Transfer-Encoding: binary  
Content-ID: 2
```

```
HTTP/1.1 204 No Content  
OData-Version: 4.0
```

```
--changesetresponse_a08c1df6-4b82-4a9b-be61-7ef4cc7b23ba
Content-Type: application/http
Content-Transfer-Encoding: binary
Content-ID: 3
```

```
HTTP/1.1 204 No Content
```

```
--changesetresponse_a08c1df6-4b82-4a9b-be61-7ef4cc7b23ba--
--batchresponse_e17aace9-5ccb-49bd-b7ad-f1be8cc8c9d8--
```

Batch-request (Content-Type: multipart/mixed and different sets of requests)

Batch-request

```
POST http://mycreatio.com/0/odata/$batch

Content-Type: multipart/mixed; boundary=batch_a685-9724-d873
Accept: application/json
BPMCSRF: OpK/NuJJ1w/SQxmPvwNvfO
ForceUseSession: true
```

Batch-request body

```
--batch_a685-9724-d873
Content-Type: multipart/mixed; boundary=changeset_06da-d998-8e7e

--changeset_06da-d998-8e7e
Content-Type: application/http
Content-Transfer-Encoding: binary

// Add an object instance of the City collection.
POST City HTTP/1.1
Content-ID: 1
Accept: application/atomsvc+xml;q=0.8,
application/json;odata=verbose;q=0.5, */*;q=0.1
Content-Type: application/json;odata=verbose

// Add the d6bc67b1-9943-4e47-9aaf-91bf83e9c285 value in the Id field.
// Add the Nebraska value in the Name field.
{"Id": "d6bc67b1-9943-4e47-9aaf-91bf83e9c285", "Name": "Nebraska"}

--batch_a685-9724-d873
Content-Type: multipart/mixed; boundary=changeset_06da-d998-8e71

--changeset_06da-d998-8e71
Content-Type: application/http
Content-Transfer-Encoding: binary

// Add an object instance of the City collection.
```

```
POST City HTTP/1.1
Content-ID: 2
Accept: application/atomsvc+xml;q=0.8,
application/json;odata=verbose;q=0.5, */*;q=0.1
Content-Type: application/json;odata=verbose

// Add the d6bc67b1-9943-4e47-9aaf-91bf83e9c286 value in the Id field.
// Add the Durham value in the Name field.
{"Id": "d6bc67b1-9943-4e47-9aaf-91bf83e9c286", "Name": "Durham"}
```

Response to the batch-request

Status: 200 OK

```
{
  "responses": [
    {
      "id": "1",
      "atomicityGroup": "e9621f72-42bd-47c1-b271-1027e4b68e3b",
      "status": 201,
      "headers": {
        "location":
"https://mycreatio.com/0/odata/City(d6bc67b1-9943-4e47-9aaf-91bf83e9c285)",
        "content-type": "application/json;
odata.metadata=minimal",
        "odata-version": "4.0"
      },
      "body": {
        "@odata.context":
"https://mycreatio.com/0/odata/$metadata#City/$entity",
        "Id": "d6bc67b1-9943-4e47-9aaf-91bf83e9c285",
        "CreatedOn": "2020-05-18T18:49:16.3766324Z",
        "CreatedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",
        "ModifiedOn": "2020-05-18T18:49:16.3766324Z",
        "ModifiedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",
        "Name": "Nebraska",
        "Description": "",
        "CountryId": "00000000-0000-0000-0000-000000000000",
        "RegionId": "00000000-0000-0000-0000-000000000000",
        "TimeZoneId": "00000000-0000-0000-0000-000000000000",
        "ProcessListeners": 0
      }
    },
    {
      "id": "2",
      "atomicityGroup": "960e2272-d8cb-4b4d-827c-0181485dd71d",
      "status": 201,
      "headers": {
        "location":
"https://mycreatio.com/0/odata/City(d6bc67b1-9943-4e47-9aaf-91bf83e9c286)",
```

```
        "content-type": "application/json",
odata.metadata=minimal",
        "odata-version": "4.0"
    },
    "body": {
        "@odata.context":
"https://mycreatio.com/0/odata/$metadata#City/$entity",
        "Id": "d6bc67b1-9943-4e47-9aaf-91bf83e9c286",
        "CreatedOn": "2020-05-18T18:49:16.4078852Z",
        "CreatedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",
        "ModifiedOn": "2020-05-18T18:49:16.4078852Z",
        "ModifiedBy": "dad159f3-6c2d-446a-98d2-0f4d26662bbe",
        "Name": "Durham",
        "Description": "",
        "CountryId": "00000000-0000-0000-0000-000000000000",
        "RegionId": "00000000-0000-0000-0000-000000000000",
        "TimeZoneId": "00000000-0000-0000-0000-000000000000",
        "ProcessListeners": 0
    }
}
]
```

DataService

Contents

- [DataService. Adding records](#)
- [DataService. Reading records](#)
- [DataService. Data filtering](#)
- [DataService. Using macros](#)
- [DataService. Updating records](#)
- [DataService. Deleting records](#)
- [DataService. Batch queries](#)

DataService. Adding records

Beginner

Easy

Medium

Advanced

General information

The Creatio DataService web service is a [RESTfull service](#). RESTful is a quite simple information management interface that doesn't use any additional internal layers, i.e., the data doesn't need to be converted to any third-party format, such as XML. In a simple RESTful service, each record is uniquely identified by a global identifier such as URL. Each URL, in turn, has a strictly specified format. However, this service is not always convenient for transferring large amounts of data.

With the use of the DataService, the data can be automatically configured in various data formats such as XML, JSON, HTML, CSV, and JSV. The data structure is determined by [data contracts](#). A complete list of data contracts used by the DataService, can be found in the "**DataService**" article.

InsertQuery data contract

The *InsertQuery* data contract is used to add records to sections. The data is transferred to the DataService via HTTP by using the POST request with the following URL:

```
// URL format of the POST query to add data to DataService.
http(s)://[Creatio application address]/[Configuration number]/dataservice/[Data
format]/reply/InsertQuery
// URL example for the POST query to add data to DataService.
http(s)://example.creatio.com/0/dataservice/json/reply/InsertQuery
```

The InsertQuery data contract has a hierarchical structure with multiple nesting levels. In the Creatio application server part, the InsertQuery data contract is represented by the *InsertQuery* class of the *Terrasoft.Nui.ServiceModel.DataContract* namespace of the *Terrasoft.Nui.ServiceModel.dll* class library. However, for simplicity, the hierarchical structure of the *InsertQuery* data contract is conveniently presented as a JSON format object:

```
{
    "RootSchemaName": "[Root object schema name]",
    "OperationType": [Record operation type],
    "ColumnValues": {
        "Items": [
            {
                "Added columnName": {
                    "ExpressionType": [Expression type],
                    "Parameter": {
                        "DataValueType": [Data type],
                        "Value": "[Column value]"
                    }
                }
            }...
        ]
    }
}
```

The basic properties of the *InsertQuery* class and their possible values are presented in table 1.

Table 1. *InsertQuery* class properties.

Property	Description										
RootSchemaName	A string containing the name of the root object schema of the added record.										
OperationType	Operation type is set by the <i>QueryOperationType</i> namespace <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace enumeration value. For the <i>InsertQuery</i> the <i>QueryOperationType.Insert</i> value is set. <i>QueryOperationType</i> enumeration values: <table> <tbody> <tr> <td><i>Select</i></td><td>0</td></tr> <tr> <td><i>Insert</i></td><td>1</td></tr> <tr> <td><i>Update</i></td><td>2</td></tr> <tr> <td><i>Delete</i></td><td>3</td></tr> <tr> <td><i>Batch</i></td><td>4</td></tr> </tbody> </table>	<i>Select</i>	0	<i>Insert</i>	1	<i>Update</i>	2	<i>Delete</i>	3	<i>Batch</i>	4
<i>Select</i>	0										
<i>Insert</i>	1										
<i>Update</i>	2										
<i>Delete</i>	3										
<i>Batch</i>	4										
ColumnValues	Contains a collection of column values of the added record. Its <i>ColumnValues</i> type is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace.										

The *ColumnValues* class has a single *Items* property that is defined as a collection of the *Dictionary<string, ColumnExpression>* keys and values. The key is a string with the added column title, and the value is the object with the *ColumnExpression* type defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace. The basic properties of the *ColumnExpression* class used when adding records, are given in table 2.

Table 2. *ColumnExpression* class main properties

Property	Description
ExpressionType	The expression type that defines the value that will be contained in the added column. Set by the <i>EntitySchemaQueryExpressionType</i> enumeration of the

Terrasoft.Core.Entities namespace defined in the *Terrasoft.Core* class library. For the *InsertQuery* the *EntitySchemaQueryExpressionType.Parameter* value is set.

EntitySchemaQueryExpressionType enumeration type:

<i>SchemaColumn</i>	0
<i>Function</i>	1
<i>Parameter</i>	2
<i>SubQuery</i>	3
<i>ArithmeticOperation</i>	4

Parameter	Defines the value that will be contained in the added column. Its <i>Parameter</i> type is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace.
-----------	---

The *Parameter* class has multiple properties, two of which are used to add records (table 3).

Table 3 Parameter class main properties

Property	Description
<i>DataValueType</i>	The data value type that defines the value that will be contained in the added column. Set by the <i>DataValueType</i> enumeration value of the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace. <i>DataValueType</i> enumeration type:
<i>Value</i>	The object that contains the added column value.

DataService. Reading records

Beginner

Easy

Medium

Advanced

General provisions

The DataService web service of Creatio is a RESTful ([Representational State Transfer, REST](#)) service. RESTful data management interface does not require converting data to an external format, such as XML. In a simple RESTful service, each information unit is determined by a global Identifier such as URL. Each URL, in its turn, has a strictly specified format. This is not an optimal way to transfer large arrays of data.

With the use of the DataService, the data can be automatically configured in various data formats such as XML, JSON, HTML, CSV, and JSV. The data structure is determined by [data contracts](#). A complete list of data contracts used by the DataService, can be found in the "**DataService**" article.

SelectQuery data contract

The SelectQuery data contract is used for reading section records. The query data is transferred to DataService via HTTP, with the help of POST by the following URL:

```
// URL format of the POST query to read data from DataService.  
http(s)://[Creatio application address]/[Configuration number]/dataservice/[Data  
format]/reply>SelectQuery  
// URL example of the POST query to read data from DataService.  
http(s)://example.creatio.com/0/dataservice/json/reply>SelectQuery
```

The SelectQuery data contract has a complex hierarchical structure with a number of nesting levels. In the Creatio server core, it is represented by a SelectQuery class of theTerrasoft.Nui.ServiceModel.DataContract namespace of the Terrasoft.Nui.ServiceModel.dll library of classes. The hierarchical data structure of the SelectQuery data contract can be conveniently viewed in JSON format:

```
{  
    "RootSchemaName": "[Object root schema name]",  
    "OperationType": [Type of record operation],  
    "Columns": {  
        "Items": {  
            "Name": {  
                "OrderDirection": [Sorting order],  
                "OrderPosition": [Column position],  
                "Caption": "[Title]",  
                "Expression": {  
                    "ExpressionType": [Expression type],  
                    "ColumnPath": "[Path to column]",  
                    "Parameter": [Parameter],  
                    "FunctionType": [Function type],  
                    "MacrosType": [Macro type],  
                    "FunctionArgument": [Function argument],  
                    "DatePartType": [Type of date part],  
                    "AggregationType": [Aggregation type],  
                    "AggregationEvalType": [Aggregation scope],  
                    "SubFilters": [Buit-in filters]  
                }  
            }  
        }  
    },  
    "AllColumns": [Indicates that all columns are selected],  
    "ServerESQCacheParameters": {  
        "CacheLevel": [Caching level],  
        "CacheGroup": [Caching group],  
        "CacheItemName": [Record key in repository]  
    },  
    "IsPageable": [Indicates page-by-page],  
    "IsDistinct": [Indicates uniqueness],  
    "RowCount": [Number of selected records],  
    "ConditionalValues": [Conditions for building a pageable query],  
    "IsHierarchical": [Indicates hierarchical data selection],  
    "HierarchicalMaxDepth": [Maximum nesting level of the hierarchical query],  
},
```

```

    "HierarchicalColumnName": [Column name used to create hierarchical query],
    "HierarchicalColumnValue": [Initial value of hierarchical column],
    "Filters": [Filters]
  }
}

```

Primary properties of the SelectQuery class and their possible values are available in table 1.

Table 1. SelectQuery class properties

Property	Type	Notes
RootSchemaName	<i>string</i>	String that contains root schema name of the added record object.
OperationType	<i>QueryOperationType</i>	Type of write operation. Specified as a QueryOperationType enumeration value of the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space. The <i>QueryOperationType.Select</i> value is set for <i>SelectQuery</i> . Values of the <i>QueryOperationType</i> enumeration:
	<i>Select</i>	0
	<i>Insert</i>	1
	<i>Update</i>	2
	<i>Delete</i>	3
	<i>Batch</i>	4
Columns	<i>SelectQueryColumns</i>	Contains a collection of the record columns being read. It has the <i>SelectQueryColumns</i> type defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space. It must be configured if the <i>AllColumns</i> checkbox is set to <i>false</i> and a set of specific root schema columns, which does not include the <i>[Id]</i> column, is required.
AllColumns	<i>bool</i>	Indicates if all columns are selected. If the value is set to true, all columns of the root schema will be selected by the query.
ServerESQCacheParameters	<i>ServerESQCacheParameters</i>	Parameters of <i>EntitySchemaQuery</i> caching on server. The <i>ServerESQCacheParameters</i> type is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space.
IsPageable	<i>bool</i>	Indicates whether the data is selected page-by-page.
IsDistinct	<i>bool</i>	Indicates whether duplicates must be eliminated in the resulting data set.
RowCount	<i>int</i>	Number of selected strings. By default, the value is -1, i.e. all strings are selected.
ConditionalValues	<i>ColumnValues</i>	Conditions of creating a page-by-page query. The <i>ColumnValues</i> type is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space.
IsHierarchical	<i>bool</i>	Indicates whether the data is selected hierarchically.
HierarchicalMaxDepth	<i>int</i>	Maximum nesting level of a hierarchical query.
hierarchicalColumnName	<i>string</i>	Name of the column used for creating a hierarchical query.
hierarchicalColumnValue	<i>string</i>	Initial value of hierarchical column from which the hierarchy will be built.
Filters	<i>Filters</i>	Collection of query filters. The <i>Filters</i> type is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space.
ColumnValues	<i>ColumnValues</i>	Contains collection of column values for the added record.

The *ColumnValues* type is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space.

The *SelectQueryColumns* class has a single *Items* property, defined as a collection of keys and values *Dictionary<string, SelectQueryColumn>*. The key is the string with the name of the added column. The value is an instance of the *SelectQueryColumn* class, defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. The properties of the *SelectQueryColumn* are available in the table 2.

Table 2. *SelectQueryColumn* class properties

Property	Type	Notes
OrderDirection	<i>OrderDirection</i>	Sorting order. Specified with a value from the <i>OrderDirection</i> enumeration of the <i>Terrasoft.Common</i> name space defined in the <i>Terrasoft.Common</i> class library.
OrderPosition	<i>int</i>	Sets position number in the collection of the query columns, by which the sorting is done.
Caption	<i>string</i>	Column title.
Expression	<i>ColumnExpression</i>	Property that defines expression of the type of selected column.

The *ColumnExpression* class defines expression that sets the type of the schema column. The class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space of the *Terrasoft.Nui.ServiceModel* library. The properties of an instance of this class are filled in depending on the *ExpressionType* property, which sets the expression type. The full list of the *ColumnExpression* class properties is available in table 3.

Table 3. Primary properties of the *ColumnExpression* class

Property	Type	Notes															
ExpressionType	<i>EntitySchemaQuery ExpressionType</i>	Type of expression that determines the value that the added column will contain. Specified with a value from the <i>EntitySchemaQueryExpressionType</i> enumeration of the <i>Terrasoft.Core.Entities</i> name space defined in the <i>Terrasoft.Core</i> class library. The <i>EntitySchemaQueryExpressionType.Parameter</i> value is set for <i>InsertQuery</i> . Values of the <i>EntitySchemaQueryExpressionType</i> enumeration: <table> <tr> <td><i>SchemaColumn</i></td> <td>0</td> <td>Schema column</td> </tr> <tr> <td><i>Function</i></td> <td>1</td> <td>Function</td> </tr> <tr> <td><i>Parameter</i></td> <td>2</td> <td>Parameter</td> </tr> <tr> <td><i>SubQuery</i></td> <td>3</td> <td>Subquery</td> </tr> <tr> <td><i>ArithmeticOperation</i></td> <td>4</td> <td>Arithmetic operation</td> </tr> </table>	<i>SchemaColumn</i>	0	Schema column	<i>Function</i>	1	Function	<i>Parameter</i>	2	Parameter	<i>SubQuery</i>	3	Subquery	<i>ArithmeticOperation</i>	4	Arithmetic operation
<i>SchemaColumn</i>	0	Schema column															
<i>Function</i>	1	Function															
<i>Parameter</i>	2	Parameter															
<i>SubQuery</i>	3	Subquery															
<i>ArithmeticOperation</i>	4	Arithmetic operation															
ColumnPath	<i>string</i>	Path to the column in relation to the root schema. Rules for building paths are available in the " The use of EntitySchemaQuery for creation of queries in database (on-line documentation) " article.															
Parameter	<i>Parameter</i>	Determines the value that the added column will contain. The <i>Parameter</i> type is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space.															
FunctionType	<i>FunctionType</i>	Function type. Specified with a value from the <i>FunctionType</i> enumeration, which is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space. Values of the <i>FunctionType</i> enumeration: <table> <tr> <td><i>None</i></td> <td>0</td> <td>Not defined</td> </tr> </table>	<i>None</i>	0	Not defined												
<i>None</i>	0	Not defined															

		<i>Macros</i>	1	Macro
		<i>Aggregation</i>	2	Aggregate function
		<i>DatePart</i>	3	Part of date value
		<i>Length</i>	4	Length
MacrosType	<i>EntitySchemaQuery MacrosType</i>	Macro type. Specified with a value of the <i>EntitySchemaQueryMacrosType</i> enumeration, which is defined in the <i>Terrasoft.Core.Entities</i> name space.		
FunctionArgument	<i>BaseExpression</i>	Function argument. Accepts a value if the function is defined with a parameter. The <i>BaseExpression</i> class is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space, is an ancestor for the <i>ColumnExpression</i> class and has the same set of properties.		
DatePartType	<i>DatePart</i>	Part of date value Specified with a value from the <i>DatePart</i> enumeration, which is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space.		
		Values of the <i>DatePart</i> enumeration:		
		<i>None</i>	0	Not defined
		<i>Day</i>	1	Day
		<i>Week</i>	2	Week
		<i>Month</i>	3	Month
		<i>Year</i>	4	Year
		<i>Weekday</i>	5	Week day
		<i>Hour</i>	6	Hour
		<i>HourMinute</i>	7	Minute
AggregationType	<i>AggregationType</i>	Aggregate function type. Specified with a value from the <i>AggregationType</i> enumeration defined in the <i>Terrasoft.Common</i> name space defined in the <i>Terrasoft.Common</i> class library.		
AggregationEvalType	<i>AggregationEvalType</i>	Aggregate function scope. Specified with a value from the <i>AggregationEvalType</i> enumeration defined in the <i>Terrasoft.Common</i> name space defined in the <i>Terrasoft.Common</i> class library.		
SubFilters	<i>Filters</i>	Collection of subquery filters. The <i>Filters</i> type is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space.		

The *Parameter* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. Its properties are available in table 4.

Table 4. Parameter class properties

Property	Type	Notes
DataValueType	DataValueType	Type of data for the value that the added column will contain. Specified as a <i>DataValueType</i> enumeration value of the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space.
		Values of the <i>DataValueType</i> enumeration:
	<i>Guid</i>	0
	<i>Text</i>	1

<i>Integer</i>	4
<i>Float</i>	5
<i>Money</i>	6
<i>DateTime</i>	7
<i>Date</i>	8
<i>Time</i>	9
<i>Lookup</i>	10
<i>Enum</i>	11
<i>Boolean</i>	12
<i>Blob</i>	13
<i>Image</i>	14
<i>ImageLookup</i>	16
<i>Color</i>	18
<i>Mapping</i>	26

Value	<i>object</i>	The object that contains the value of the added column.
ArrayValue	<i>string[]</i>	Array of the added column values. Used when serializing arrays and BLOBs.
ShouldSkipConversion	<i>bool</i>	Indicates the need to skip the process of providing the type for the <i>Value</i> property.

The *ServerESQCacheParameters* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. Its properties are available in table 5.

Table 5. ServerESQCacheParameters class properties

Property	Type	Notes
CacheLevel	<i>int</i>	Data allocation level in the EntitySchemaQuery cache.
CacheGroup	<i>string</i>	Caching group.
CacheItemName	<i>string</i>	Repository record key.

The *Filters* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. For details on the properties of this class and its use, please see the "**DataService. Data filtering**" article.

DataService. Data filtering

Beginner Easy Medium Advanced

General information

During the execution of DataService operations, it is often necessary to filter data. For example, when reading section records, you need to fetch only those records that meet certain criteria. Creatio provides the *Filters* class to form these criteria .

The Filters class

The *Filters* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace of the *Terrasoft.Nui.ServiceModel.dll* class library. For simplicity, the hierarchical structure of the *Filters* data filter is conveniently presented as a JSON format object:

```

"Filters": {
    "RootSchemaName": ["Root schema name"],
    "FilterType": [Filter type],
    "ComparisonType": [Comparison type],
    "LogicalOperation": [Logical operation],
    "IsNull": [Completeness checkbox],
    "IsEnabled": [Activation checkbox],
    " IsNot": [Negation operator checkbox],
    "SubFilters": [Subquery filters],
    "Items": [Filter group collection],
    "LeftExpression": [Expression to be checked],
    "RightExpression": [Filtration expression],
    "RightExpressions": [Filtration expressions array],
    "RightLessExpression": [Initial filtration range expression],
    "RightGreaterExpression": [Final filtration range expression],
    "TrimDateTimeParameterToDate": [Cutting time for date/time parameters checkbox],
    "Key": ["Filter key in the filter collection"],
    "IsAggregative": [Aggregating filter checkbox],
    "LeftExpressionCaption": ["Expression title to be checked"],
    "ReferenceSchemaName": ["Reference schema name"]
}

```

The basic properties of the *Filters* class and their possible values are presented in table 1.

Table 1. *Filters class properties*.

Property	Type	Description																					
RootSchemaName	<i>string</i>	A string containing the name of the root object schema of the added record.																					
FilterType	<i>FilterType</i>	Filter type. Set by the <i>FilterType</i> enumeration value of the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace. <i>FilterType</i> enumeration values: <table> <tr> <td><i>None</i></td><td>0</td><td>Filter type not defined.</td></tr> <tr> <td><i>CompareFilter</i></td><td>1</td><td>Comparison filter. Used to compare expression results.</td></tr> <tr> <td><i>IsNullFilter</i></td><td>2</td><td>The filter that defines whether an expression is empty.</td></tr> <tr> <td><i>Between</i></td><td>3</td><td>The filter that defines whether an expression is one of the expressions.</td></tr> <tr> <td><i>InFilter</i></td><td>4</td><td>The filter that defines whether an expression equals one of the expressions.</td></tr> <tr> <td><i>Exists</i></td><td>5</td><td>Existence filter.</td></tr> <tr> <td><i>FilterGroup</i></td><td>6</td><td>Filter group. Filter groups can be nested in one another, i.e., the collection itself can be an element of another collection.</td></tr> </table>	<i>None</i>	0	Filter type not defined.	<i>CompareFilter</i>	1	Comparison filter. Used to compare expression results.	<i>IsNullFilter</i>	2	The filter that defines whether an expression is empty.	<i>Between</i>	3	The filter that defines whether an expression is one of the expressions.	<i>InFilter</i>	4	The filter that defines whether an expression equals one of the expressions.	<i>Exists</i>	5	Existence filter.	<i>FilterGroup</i>	6	Filter group. Filter groups can be nested in one another, i.e., the collection itself can be an element of another collection.
<i>None</i>	0	Filter type not defined.																					
<i>CompareFilter</i>	1	Comparison filter. Used to compare expression results.																					
<i>IsNullFilter</i>	2	The filter that defines whether an expression is empty.																					
<i>Between</i>	3	The filter that defines whether an expression is one of the expressions.																					
<i>InFilter</i>	4	The filter that defines whether an expression equals one of the expressions.																					
<i>Exists</i>	5	Existence filter.																					
<i>FilterGroup</i>	6	Filter group. Filter groups can be nested in one another, i.e., the collection itself can be an element of another collection.																					
ComparisonType	<i>FilterComparisonType</i>	Comparison operation type. Set by the <i>FilterComparisonType</i> enumeration value of the <i>Terrasoft.Core.Entities</i> namespace.																					
LogicalOperation	<i>LogicalOperationStrict</i>	Logical operation. This type does not allow the None value specified in the <i>LogicalOperationStrict</i> enumeration of the <i>Terrasoft.Common</i> namespace.																					

IsNull	<i>bool</i>	Expression completion checkbox.
.IsEnabled	<i>bool</i>	Checkbox that defines whether the filter is active and will be taken into account when building a request.
IsNot	<i>bool</i>	Specifies whether to use the negation logical operator.
SubFilters	<i>Filters</i>	Subrequest filters. Cannot contain filters with other subrequests.
Items	<i>Dictionary<string, Filter></i>	Collection containing a filter group.
LeftExpression	<i>BaseExpression</i>	The expression in the left part of the comparison, i.e. the expression to be tested. The <i>BaseExpression</i> class is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace.
RightExpression	<i>BaseExpression</i>	The filter expression that will be compared to the expression contained in the <i>LeftExpression</i> property.
RightExpressions	<i>BaseExpression[]</i>	The expression array that will be compared to the expression contained in the <i>LeftExpression</i> property.
RightLessExpression	<i>BaseExpression</i>	Initial filtration range expression.
RightGreaterExpression	<i>BaseExpression</i>	Final filtration range expression.
TrimDateTime	<i>bool</i>	Checkbox indicating whether to cut time from the date-time parameters.
ParameterToDate		
Key	<i>string</i>	Filter key in the collection of Items filters.
IsAggregative	<i>bool</i>	Aggregating filter checkbox.
LeftExpressionCaption	<i>string</i>	Left comparison part title.
ReferenceSchemaName	<i>string</i>	The object schema name referenced by the left part of the filter if the column type is lookup.

The *BaseExpression* class is the base expression class. It is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace of the *Terrasoft.Nui.ServiceModel* library. The properties of this class instance are populated depending on the *ExpressionType* property that specifies the expression type. A complete list of the *BaseExpression* class properties is given in table 2.

Table 2. *BaseExpression* class main properties

Property	Type	Description	
ExpressionType	<i>EntitySchemaQuery</i> <i>ExpressionType</i>	The expression type that defines the value that will be contained in the added column. Set by the <i>EntitySchemaQueryExpressionType</i> enumeration of the <i>Terrasoft.Core.Entities</i> namespace defined in the <i>Terrasoft.Core</i> class library. For the <i>InsertQuery</i> the <i>EntitySchemaQueryExpressionType.Parameter</i> value is set.	The <i>EntitySchemaQueryExpressionType</i> enumeration values:
		<i>SchemaColumn</i>	0 Schema column.
		<i>Function</i>	1 Function
		<i>Parameter</i>	2 Parameter
		<i>SubQuery</i>	3 Subquery
		<i>ArithmeticOperation</i>	4 Arithmetic operation
ColumnPath	<i>string</i>	The path to a column relative to the root schema. The rules for building the paths can be found in the "The use of EntitySchemaQuery for creation of queries in database"	

[\(on-line documentation\)](#)" article.

Parameter	<i>Parameter</i>	Defines the value that will be contained in the added column. Its <i>Parameter</i> type is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace.																								
FunctionType	<i>FunctionType</i>	Function type. Set by the value from the <i>FuctionType</i> enumeration defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace. <i>FunctionType</i> enumeration values:																								
		<table> <tr> <td><i>None</i></td><td>0</td><td>Not defined</td></tr> <tr> <td><i>Macros</i></td><td>1</td><td>Macro</td></tr> <tr> <td><i>Aggregation</i></td><td>2</td><td>Aggregating function</td></tr> <tr> <td><i>DatePart</i></td><td>3</td><td>Date part</td></tr> <tr> <td><i>Length</i></td><td>4</td><td>Length</td></tr> </table>	<i>None</i>	0	Not defined	<i>Macros</i>	1	Macro	<i>Aggregation</i>	2	Aggregating function	<i>DatePart</i>	3	Date part	<i>Length</i>	4	Length									
<i>None</i>	0	Not defined																								
<i>Macros</i>	1	Macro																								
<i>Aggregation</i>	2	Aggregating function																								
<i>DatePart</i>	3	Date part																								
<i>Length</i>	4	Length																								
MacrosType	<i>EntitySchemaQuery MacrosType</i>	Macro type. Set by the value from the <i>EntitySchemaQueryMacrosType</i> enumeration defined in the <i>Terrasoft.Core.Entities</i> namespace.																								
FunctionArgument	<i>BaseExpression</i>	Function argument. Takes the value if the function is defined with a parameter. The <i>BaseExpression</i> class is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace and is the ancestor of the <i>ColumnExpresion</i> class and has the same set of properties.																								
DatePartType	<i>DatePart</i>	Date part. Set by the value from the <i>DatePart</i> enumeration defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace. <i>DatePart</i> enumeration values:																								
		<table> <tr> <td><i>None</i></td><td>0</td><td>Not defined</td></tr> <tr> <td><i>Day</i></td><td>1</td><td>Day</td></tr> <tr> <td><i>Week</i></td><td>2</td><td>Week</td></tr> <tr> <td><i>Month</i></td><td>3</td><td>Month</td></tr> <tr> <td><i>Year</i></td><td>4</td><td>Year</td></tr> <tr> <td><i>Weekday</i></td><td>5</td><td>Day of the week</td></tr> <tr> <td><i>Hour</i></td><td>6</td><td>Hour</td></tr> <tr> <td><i>HourMinute</i></td><td>7</td><td>Minute</td></tr> </table>	<i>None</i>	0	Not defined	<i>Day</i>	1	Day	<i>Week</i>	2	Week	<i>Month</i>	3	Month	<i>Year</i>	4	Year	<i>Weekday</i>	5	Day of the week	<i>Hour</i>	6	Hour	<i>HourMinute</i>	7	Minute
<i>None</i>	0	Not defined																								
<i>Day</i>	1	Day																								
<i>Week</i>	2	Week																								
<i>Month</i>	3	Month																								
<i>Year</i>	4	Year																								
<i>Weekday</i>	5	Day of the week																								
<i>Hour</i>	6	Hour																								
<i>HourMinute</i>	7	Minute																								
AggregationType	<i>AggregationType</i>	Aggregating function type. Sets the value of <i>AggregationType</i> enumeration defined in the namespace <i>Terrasoft.Common</i> defined in the class library <i>Terrasoft.Common</i>																								
AggregationEvalType	<i>AggregationEvalType</i>	Aggregating function Set by the value from the <i>AggregationEvalType</i> enumeration defined in the <i>Terrasoft.Core.DB</i> namespace defined in the <i>Terrasoft.Core</i> class library.																								
SubFilters	<i>Filters</i>	Subquery filter collection. Its <i>Filter</i> type is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace.																								

Learn more about filters in the "[The EntitySchemaQuery class. Filters handling](#)" article. Next, there is an example of using filters in requests to the DataService service from a third-party application.

DataService. Using macros

Beginner

Easy

Medium

Advanced

General provisions

During execution of DataService operations data often needs to be filtered for a certain period of time. Macros simplify such tasks and help to avoid creating unnecessary custom methods. The macros are implemented in a form of special classes that are designed for calculating typical values in query expressions, such as calculating the start and end date of the current quarter. Macros can be used only if the query expression type is a function. For more information about macro expression types, please see the **DataService. Data filtering** article.

Types of macros

When creating queries to DataService, both parameterized (ie requiring an argument) and non-parameterized macros can be used. Macro types that must be used in the macro expressions are defined in the *EntitySchemaQueryMacrosType* enumeration in the *Terrasoft.Core.Entities* name space. Enumeration values of macro types and their descriptions are available in table 1.

Table 1. Values of the *EntitySchemaQueryMacrosType* enumeration and their descriptions

Macro	Value	Description
CurrentHalfYear	16	Current half-year (January-June or July-December).
CurrentHour	21	Current hour.
CurrentMonth	10	Current month.
CurrentQuarter	13	Current quarter.
CurrentUser	1	Current user.
CurrentUserContact	2	Contact record of the current user.
CurrentWeek	7	Current week.
CurrentYear	19	Current year.
DayOfMonth	28	Day of month. Requires parameterization.
DayOfWeek	29	Week day. Requires parameterization.
Hour	30	Hour. Requires parameterization.
HourMinute	31	Time. Requires parameterization.
Month	32	Month. Requires parameterization.
NextHalfYear	17	Next half-year (January-June or July-December).
NextHour	22	Next hour.
NextMonth	11	Next month.
NextNDays	24	Next N days. Requires parameterization.
NextNHours	26	Next N hours. Requires parameterization.
NextQuarter	14	Next quarter.
NextWeek	8	Next week.
NextYear	23	Next year.
None	0	Type of macro not defined.
PreviousHalfYear	15	Previous half-year (January-June or July-December).
PreviousHour	20	Previous hour.
PreviousMonth	9	Previous month.

PreviousNDays	25	Previous N days. Requires parameterization.
PreviousNHours	27	Previous N hours. Requires parameterization.
PreviousQuarter	12	Previous quarter.
PreviousWeek	6	Previous week.
PreviousYear	18	Previous year.
Today	4	Today.
Tomorrow	5	Tomorrow.
Year	33	Year. Requires parameterization.
Yesterday	3	Yesterday.

DataService. Updating records

Beginner Easy Medium **Advanced**

General provisions

The DataService web service of Creatio is a RESTful ([Representational State Transfer, REST](#)) service. The RESTful data management interface does not require converting data to an external format, such as XML. In a simple RESTful service, each information unit is determined by a global Identifier such as URL. Each URL, in its turn, has a strictly specified format. This is not an optimal way to transfer large arrays of data.

With the use of the DataService, the data can be automatically configured in various data formats such as XML, JSON, HTML, CSV, and JSV. The data structure is determined by [data contracts](#). A complete list of data contracts used by the DataService, can be found in the "**DataService**" article.

UpdateQuery data contract

The UpdateQuery data contract is used for updating section records. The query data is transferred to DataService via HTTP, with the help of POST by the following URL:

```
// URL format of the POST query to DataService to update data.  
http(s)://[Creatio application address]/[Configuration number]/dataservice/[Data  
format]/reply/UpdateQuery  
// URL example of the POST query to DataService to update data.  
http(s)://example.creatio.com/0/dataservice/json/reply/UpdateQuery
```

The UpdateQuery data contract has a hierarchical structure with a number of nesting levels. In the Creatio server core, it is represented by a UpdateQuery class of the `Terrasoft.Nui.ServiceModel.DataContract` namespace of the `Terrasoft.Nui.ServiceModel.dll` library of classes. For the hierarchical data structure of the UpdateQuery data contract can be conveniently viewed in JSON format:

```
{  
    "RootSchemaName": "[Root schema]",  
    "OperationType": [Type of operation with record],  
    "IsForceUpdate": [Force update],  
    "ColumnValues": {  
        "Items": {  
            "Name of the added column": {  
                "ExpressionType": [Expression type],  
                "Parameter": {  
                    "DataValueType": [Data type],  
                    "Value": "[Column value]"  
                }  
            }...  
        },  
        "Filters": [Request filters]  
    }  
}
```

}

Primary properties of the `UpdateQuery` class and their possible values are available in table 1.

Table 1. `UpdateQuery` class properties

Property	Type	Notes										
<code>RootSchemaName</code>	<code>string</code>	String that contains root schema name of added record object.										
<code>OperationType</code>	<code>QueryOperationType</code>	Type of write operation. Specified as a <code>QueryOperationType</code> enumeration value of the <code>Terrasoft.Nui.ServiceModel.DataContract</code> name space. The <code>QueryOperationType.Insert</code> value is set for <code>InsertQuery</code> . Values of the <code>QueryOperationType</code> enumeration:										
		<table> <tr> <td><i>Select</i></td><td>0</td></tr> <tr> <td><i>Insert</i></td><td>1</td></tr> <tr> <td><i>Update</i></td><td>2</td></tr> <tr> <td><i>Delete</i></td><td>3</td></tr> <tr> <td><i>Batch</i></td><td>4</td></tr> </table>	<i>Select</i>	0	<i>Insert</i>	1	<i>Update</i>	2	<i>Delete</i>	3	<i>Batch</i>	4
<i>Select</i>	0											
<i>Insert</i>	1											
<i>Update</i>	2											
<i>Delete</i>	3											
<i>Batch</i>	4											
<code>IsForceUpdate</code>	<code>bool</code>	Indicates force update. If the value is <code>true</code> , the entity will be saved on the server even if column values have been modified. Default value: <code>false</code> .										
<code>ColumnValues</code>	<code>ColumnValues</code>	Contains collection of column values for the added record. The <code>ColumnValues</code> type is defined in the <code>Terrasoft.Nui.ServiceModel.DataContract</code> name space.										
<code>Filters</code>	<code>Filters</code>	Collection of query filters. The <code>Filters</code> type is defined in the <code>Terrasoft.Nui.ServiceModel.DataContract</code> name space.										

The `ColumnValues` class has a single `Items` property, defined as a collection of keys and values `Dictionary<string, ColumnExpression>`. The key is the string with the name of the added column. The value is an object of the `ColumnExpression` type, defined in the `Terrasoft.Nui.ServiceModel.DataContract` name space. General properties of the `ColumnExpression` class used when adding records are available in table 2.

Table 2. Primary properties of the `ColumnExpression` class

Property	Description										
<code>EntityType</code>	Type of expression that determines the value that the added column will contain. Specified with a value from the <code>EntitySchemaQueryExpressionType</code> enumeration of the <code>Terrasoft.Core.Entities</code> name space defined in the <code>Terrasoft.Core</code> class library. The <code>EntitySchemaQueryExpressionType.Parameter</code> value is set for <code>InsertQuery</code> . Values of the <code>EntitySchemaQueryExpressionType</code> enumeration:										
	<table> <tr> <td><i>SchemaColumn</i></td><td>0</td></tr> <tr> <td><i>Function</i></td><td>1</td></tr> <tr> <td><i>Parameter</i></td><td>2</td></tr> <tr> <td><i>SubQuery</i></td><td>3</td></tr> <tr> <td><i>ArithmeticOperation</i></td><td>4</td></tr> </table>	<i>SchemaColumn</i>	0	<i>Function</i>	1	<i>Parameter</i>	2	<i>SubQuery</i>	3	<i>ArithmeticOperation</i>	4
<i>SchemaColumn</i>	0										
<i>Function</i>	1										
<i>Parameter</i>	2										
<i>SubQuery</i>	3										
<i>ArithmeticOperation</i>	4										
<code>Parameter</code>	Determines the value that the added column will contain. The <code>Parameter</code> type is defined in the <code>Terrasoft.Nui.ServiceModel.DataContract</code> name space.										

The Parameter class has a number of properties, only two of which are used for adding records (table 3).

Table 3. Primary properties of the Parameter class

Property	Description																																
DataValueType	Type of data for the value that the added column will contain. Specified as a <i>DataValueType</i> enumeration value of the <i>Terrasoft.Nui.ServiceModel.DataContract</i> name space. Values of the <i>DataValueType</i> enumeration: <table><tbody><tr><td><i>Guid</i></td><td>0</td></tr><tr><td><i>Text</i></td><td>1</td></tr><tr><td><i>Integer</i></td><td>4</td></tr><tr><td><i>Float</i></td><td>5</td></tr><tr><td><i>Money</i></td><td>6</td></tr><tr><td><i>DateTime</i></td><td>7</td></tr><tr><td><i>Date</i></td><td>8</td></tr><tr><td><i>Time</i></td><td>9</td></tr><tr><td><i>Lookup</i></td><td>10</td></tr><tr><td><i>Enum</i></td><td>11</td></tr><tr><td><i>Boolean</i></td><td>12</td></tr><tr><td><i>Blob</i></td><td>13</td></tr><tr><td><i>Image</i></td><td>14</td></tr><tr><td><i>ImageLookup</i></td><td>16</td></tr><tr><td><i>Color</i></td><td>18</td></tr><tr><td><i>Mapping</i></td><td>26</td></tr></tbody></table>	<i>Guid</i>	0	<i>Text</i>	1	<i>Integer</i>	4	<i>Float</i>	5	<i>Money</i>	6	<i>DateTime</i>	7	<i>Date</i>	8	<i>Time</i>	9	<i>Lookup</i>	10	<i>Enum</i>	11	<i>Boolean</i>	12	<i>Blob</i>	13	<i>Image</i>	14	<i>ImageLookup</i>	16	<i>Color</i>	18	<i>Mapping</i>	26
<i>Guid</i>	0																																
<i>Text</i>	1																																
<i>Integer</i>	4																																
<i>Float</i>	5																																
<i>Money</i>	6																																
<i>DateTime</i>	7																																
<i>Date</i>	8																																
<i>Time</i>	9																																
<i>Lookup</i>	10																																
<i>Enum</i>	11																																
<i>Boolean</i>	12																																
<i>Blob</i>	13																																
<i>Image</i>	14																																
<i>ImageLookup</i>	16																																
<i>Color</i>	18																																
<i>Mapping</i>	26																																
Value	The object that contains the value of the added column. Has the <i>Object</i> type.																																

The *Filters* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* name space. For details on the properties of this class and its use, please see the "**DataService. Data filtering**" article.

An instance of the *UpdateQuery* class must contain a link to a correctly initialized instance of the *Filters* class in the *Filters* property. Otherwise, new column values from the *ColumnValues* property will be set for ALL section records.

DataService. Deleting records

Beginner

Easy

Medium

Advanced

General information

The Creatio DataService web service is a [RESTfull service](#). RESTful is a quite simple information management interface that doesn't use any additional internal layers, i.e., the data doesn't need to be converted to any third-party format, such as XML. In a simple RESTful service, each record is uniquely identified by a global identifier such as a URL. Each URL has a strictly specified format. However, this service is not always convenient for transferring large amounts of data.

With the use of the DataService, the data can be automatically configured in various data formats such as XML, JSON, HTML, CSV, and JSV. The data structure is determined by [data contracts](#). A complete list of data contracts used by the DataService, can be found in the "**DataService**" article.

DeleteQuery data contract

The *DeleteQuery* contract is used to delete sections. The data is transferred to the DataService via HTTP by using the POST request with the following URL:

```
// URL format of the POST query to DataService to delete data.  
http(s)://[Creatio application address]/[Configuration number]/dataservice/[Data  
format]/reply/DeleteQuery  
// URL example of the POST query to DataService to delete data.  
http(s)://example.creatio.com/0/dataservice/json/reply/DeleteQuery
```

The *DeleteQuery* data contract has a hierarchical structure with multiple nesting levels. In the Creatio application server part, the *DeleteQuery* data contract is represented by the *InsertQuery* class of the *Terrasoft.Nui.ServiceModel.DataContract* namespace of the *Terrasoft.Nui.ServiceModel.dll* class library. However, for simplicity, the hierarchical structure of the *DeleteQuery* data contract is conveniently presented as a JSON format object:

```
{  
    "RootSchemaName": "[Root schema]",  
    "OperationType": [Record operation type],  
    "ColumnValues": [Column values. Not used.],  
    "Filters": [Query filters]  
}
```

The basic properties of the *DeleteQuery* class and their possible values are presented in table 1.

Table 1. *DeleteQuery class properties*.

Property	Type	Description										
RootSchemaName	string	A string containing the name of the root object schema of the added record.										
OperationType	<i>QueryOperationType</i>	Operation type is set by the <i>QueryOperationType</i> namespace <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace enumeration value. For the <i>InsertQuery</i> the <i>QueryOperationType.Insert</i> value is set. <i>QueryOperationType</i> enumeration values: <table><tbody><tr><td><i>Select</i></td><td>0</td></tr><tr><td><i>Insert</i></td><td>1</td></tr><tr><td><i>Update</i></td><td>2</td></tr><tr><td><i>Delete</i></td><td>3</td></tr><tr><td><i>Batch</i></td><td>4</td></tr></tbody></table>	<i>Select</i>	0	<i>Insert</i>	1	<i>Update</i>	2	<i>Delete</i>	3	<i>Batch</i>	4
<i>Select</i>	0											
<i>Insert</i>	1											
<i>Update</i>	2											
<i>Delete</i>	3											
<i>Batch</i>	4											
ColumnValues	<i>ColumnValues</i>	Contains a collection of column values of the added record. Inherited from the <i>BaseQuery</i> parent class. Not used in this type of queries.										
Filters	<i>Filters</i>	Query filter collection. Its <i>Filter</i> type is defined in the <i>Terrasoft.Nui.ServiceModel.DataContract</i> namespace.										

The *Filters* class is defined in the *Terrasoft.Nui.ServiceModel.DataContract* namespace. The properties of this class are described in the "**DataService. Data filtering**" article.

IMPORTANT

The *DeleteQuery* query class instance must contain a link to the correctly initialized *Filters* class instance in the *Filters* property. Otherwise ALL section records will be deleted.

DataService. Batch queries

Beginner

Easy

Medium

Advanced

General provisions

The DataService web service of Creatio is a RESTful ([Representational State Transfer, REST](#)) service. The RESTful data management interface does not require converting data to an external format, such as XML. In a simple RESTful service, each information unit is determined by a global Identifier such as URL. Each URL, in its turn, has a strictly specified format. This is not an optimal way to transfer large arrays of data.

With the use of the DataService, the data can be automatically configured in various data formats such as XML, JSON, HTML, CSV, and JSV. The data structure is determined by [data contracts](#). A complete list of data contracts used by the DataService, can be found in the "**DataService**" article.

Batch queries

Batch queries are used to minimize requests to DataService, which improves application performance. Packet query is a collection that contains a custom set of DataService requests. The query data is transferred to DataService via HTTP, with the help of POST by the following URL:

```
// URL format of the batch POST query to DataService.  
http(s)://[Creatio application address]/[Configuration number]/dataservice/[Data  
format]/reply/BatchQuery  
// URL example of the batch POST query to DataService.  
http(s)://example.creatio.com/0/dataservice/json/reply/BatchQuery
```

The data that comprises a batch query can be passed in different formats. One of the more convenient formats is JSON: The structure of a batch query in JSON format is as follows:

```
{  
    "items": [  
        {  
            "__type": "[Full qualified name of the query type]",  
            //One-time query contents.  
            ...  
        },  
        // Other one-time queries.  
        ...  
    ]  
}
```

To generate the contents of one-time queries that comprise a batch query, use the following data constants: *InsertQuery*, *SelectQuery*, *UpdateQuery* and *DeleteQuery*.

Creatio development cases

Contents

- **Section business logic**
- **Page configuration**
- **Work with details**
- **Business processes**
- **Typical customizations**
- **Analytics**
- **Sales Creatio customization**
- **Financial Services Creatio customization**
- **Marketing Creatio customization**
- **Service Creatio customization**
- **Prediction**

Section business logic

Contents

- **Creating a new section**
- **Adding an action to the list**
- **How to add a button to a section**
- **How to highlight a record in the list in color**
- **Adding quick filter block to a section**
- **Deleting a section**

Creating a new section

Beginner

Easy

Medium

Advanced

Introduction

One of typical development tasks in Creatio is that of adding a new section. Use [section wizard](#) to implement this. The section wizard enables you to set up base properties of sections, pages, business rules and DCM cases.

The result of the performed settings will be object and section page schemas, added to the current custom package (table 1.2).

Table 1. Object schemas created by the section wizrad

Naming rules	Purpose	Parent
[Section object name]	Primary section object	Base object (<i> BaseEntity</i>)
[Section object name] Folder	Object group.	Base folder (<i> BaseFolder</i>)
	Utility object for the correct groupage of section records. Forms the overall section folder tree.	
[Section object name] InFolder	Object in the group.	Base element in the group (<i> BaseItemInFolder</i>)
	Utility object for the correct operation of section record groupage. Defines links between the section records and folders they belong to.	

[Section object name] File	Object for the [Attachments] detail.	File (<i>File</i>)
[Section object name] Tag	Section tag.	Base tag (<i>BaseTag</i>)
[Section object name] InTag	Tag in the section object.	Base tag in the base object (<i>BaseEntityInTag</i>)

Table 2. Client schemas created by the section wizard

Naming rules	Purpose	Parent
[Section object name] Section	Section schema	Base section schema (<i>BaseSectionV2</i>)
[Section object name] Page	Section edit page schema	<i>Section edit page base schema</i> (<i>BaseModulePageV2</i>)

Section wizard and the [Custom] package

Section wizard does not only create different schemas but also links data to the current package. However, it is almost impossible to transfer the linked data to another custom package if your current package is the [Custom]. The [Custom] package is not used for committing to the version control system and transferring the changes to other environments. That is why the [Custom] package is not recommended to use as the current custom package. Learn more about the [Custom] package in the “[Package \[Custom\]](#)” article.

To change the current package, use the [Current package] system setting (*CurrentPackageId*). We recommend you to check this system setting value before you run the section wizard.

Sequence of actions for adding a section

1. Create a section using the section wizard and add the necessary workplace.
2. Add the necessary columns to the section object schema and display them on the record list, on the edit page and in details.

Case description

Add a [Car showroom] workplace to the application. Add a [Trucks] custom section to the created workplace. The “trucks” object schema must contain the following obligatory fields:

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

Case implementation algorithm

1. Create a section using the section wizard and add it to the necessary workplace.

Creating a workplace is covered in the “[Workplace setup](#)” article. Indicate the [Name] – “Car showroom” for the new workplace and add a group of users who will have access to the created workplace.

Section wizard operation is covered in the “[Section wizard](#)” article. Use the first section wizard step to create a new section. The [Name] column with the “string” type and columns inherited from the base object will be added to the primary section object. For the initial section setup, populate the following values at the first wizard step:

- [Code] – “UsrTruck”;
- [Title] – “Trucks”.

The section object name populated in the [Code] field of the section wizard should not contain prefixes “Base”, “Sys” and “Vw”. Neither should it contain suffixes “InFolder”, “Lcz”, “Lookup” and “Settings”. Otherwise, you will not be able to set up import from Excel for this object..

If you work with Creatio default settings, you will receive a notification that the value should start with the “Usr” prefix when you populate the [Code] field. The prefix value is indicated in the [Prefix for object name] system setting (*SchemaNamePrefix*). You can customize the prefix value if needed. We do not recommend to use an empty string as

a prefix because of possible name matches with other base configuration elements.

As a result, you will create all schemas that are necessary for section operation in the custom package (fig.1).

Fig. 1. – The [Trucks] section schemas in the custom package

Schemas: All		External Assemblies: All		SQL Scripts: All		Data: All		Package Dependencies					
Add	Edit	Delete											
▲ Name ¹	Package	▲ Title ²							Database Update Required				
UsrTruck	sdkCreateNewSection	Truck							<input checked="" type="checkbox"/>				
UsrTruck1Page	sdkCreateNewSection	Card schema: "Truck"							<input type="checkbox"/>				
UsrTruck1Section	sdkCreateNewSection	Section schema: "Trucks"							<input type="checkbox"/>				
UsrTruckFile	sdkCreateNewSection	Truck attachment							<input checked="" type="checkbox"/>				
UsrTruckFolder	sdkCreateNewSection	Truck folder							<input checked="" type="checkbox"/>				
UsrTruckInFolder	sdkCreateNewSection	Truck in Folder							<input checked="" type="checkbox"/>				
UsrTruckInTag	sdkCreateNewSection	Truck section record tag							<input checked="" type="checkbox"/>				
UsrTruckTag	sdkCreateNewSection	Truck section tag							<input checked="" type="checkbox"/>				

2. Add the necessary columns to the section object schema and display them.

There are two ways to add new columns to the object schema:

1. Create a new column via the section wizard and add it to the edit page immediately. The column will be automatically added to the section primary object schema. Setting up the section record edit page fields is covered in the "[How to set up page fields](#)" article.
2. Add a column to the section primary object schema via object designer in the [Configuration] section. Add columns to the page via the section wizard. You can learn more about the [Configuration] section capabilities in the "[Built-in IDE. The \[Configuration\] section](#)" article.

Since the section wizard is involved in any case, using the first way is more convenient.

The [Name] column is created and added to the section pages automatically by the section wizard.

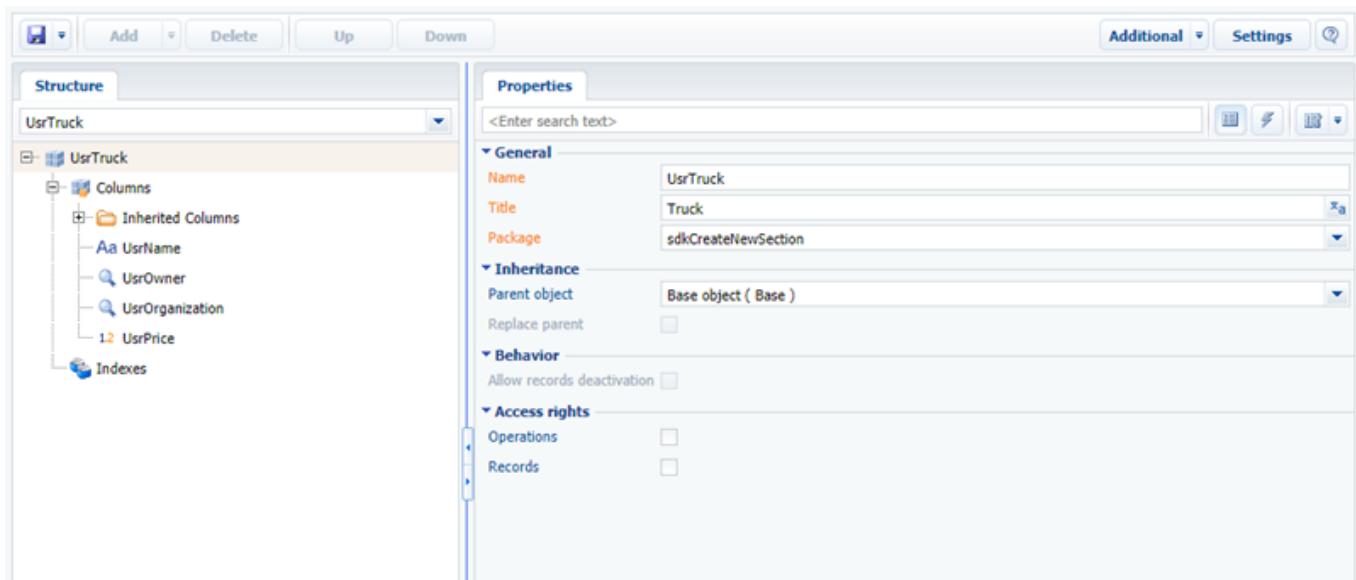
Populate the following properties for the rest of columns:

Column	Type	Title	Name in DB
Owner	The [Contact] lookup	Owner	UsrOwner
Organization	The [Account] lookup	Organization	UsrOrganization
Price	Decimal	Price, USD	UsrPrice

Select the [Is required] checkbox for all columns.

New columns will be added to the *UsrTruck* object schema after you save the changes in the section wizard (fig.2) and the corresponding configuration objects will be added to the *diff* modification array of the *UsrTruckPage* edit page schema.

Fig. 2. – The [Trucks] section primary object schema.



Display the columns on the section record list. Select the [Select fields to display] command in the list's [View] menu to open the column setup page. Section column setup is covered in the "[Setting up columns](#)" article.

After you add new records, the section will look as follows (fig.3):

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

The screenshot shows the 'Trucks' section in the 'Car showroom' workplace. The list view displays two records:

- Mercedes-benz AXOR 1840LS 4x2**: Owner - Murphy Valerie, Organization - Future Vision. Price, USD: 50 000.
- DAF XF 95**: Owner - Scott Riley, Organization - Elite Systems. Price, USD: 20 000.

Adding an action to the list

Contents

- **Introduction**
- **How to add a section action: handling the selection of a single record**
- **How to add a section action: handling the selection of several records**
- **Handling the selection of several records. Examples**

Adding an action to the list

Beginner

Easy

Medium

Advanced

Overview

Creatio has the possibility to set up a list of actions from the [Actions] menu for selected records of a section.

The list of section actions is an instance of the Terrasoft.ViewModelCollection class. Each item of the actions list is a view model.

An action is set up in the configuration object where the properties of the actions view model may be set both explicitly and through the use of the base binding mechanism.

The base content of the [Actions] menu for a section page is implemented in the base class of the BaseSectionV2 section.

The list of section actions returns the getSectionActions protected virtual method from the BaseSectionV2 schema.

A separate action is added to the collection by calling the addItem method.

The getButtonItem callback method is passed to it as a parameter. The method creates an instance of the actions view model by the configuration object passed to it as a parameter.

Example 1. — Base implementation of an action addition

```
/**  
 * This returns the actions collection of the section in the list display mode  
 * @protected  
 * @virtual  
 * @return {Terrasoft.ViewModelCollection} Actions collection of the section  
 */  
getSectionActions: function() {  
    // List of actions - Terrasoft.ViewModelCollection instance  
    var actionMenuItems = this.Ext.create("Terrasoft.ViewModelCollection");  
    // Adding an action to the collection. The method instantiating the action  
    // model instance by the passed configuration object is passed as callback.  
    actionMenuItems.addItem(  
        this.getButtonItem({  
            // Configuration object of setting an action.  
        })  
    );  
    return actionMenuItems;  
}
```

Below are the properties of the configuration object of the section action to be passed as a parameter to the getButtonItem method:

Type	a type of the [Actions] menu item	A horizontal line for separating the menu blocks may be added to the action menu using this property. For this purpose, the Terrasoft.MenuSeparator string must be specified as the property value. If no property value is specified, the menu item will be added by default.
Caption	the title of the [Actions] menu item	To set titles, the use of localizable schema strings is recommended.
Click	the action handler method is bound in this property by the method name	
Enabled	a logic property controlling the menu item availability	
Visible	a logical property controlling the menu item visibility	

Procedure for adding a custom action

1. Override the getSectionActions method.
2. Add an action to the actions collection using the addItem method.
3. Pass a configuration object with the added action settings to the getButtonItem callback method.

When base sections are replaced in the `getSectionActions` method of the replacing module, the parent implementation of this method must be called first to initialize actions of the parent section.

Examples of implementing an action addition

- **How to add a section action: handling the selection of a single record**
- **How to add a section action: handling the selection of several records**
- **Handling the selection of several records. Examples**

See also:

- **Adding an action to the edit page**

How to add a section action: handling the selection of a single record

Beginner

Easy

Medium

Advanced

Case description

Implement an action, which displays the order creation date in the message window for the [Orders] section list. The action is only available for orders with the [In progress] status.

The [Orders] section is available in Sales Creatio products.

You can address the selected record via the `ActiveRow` section view model attribute, which gets the primary column value of the selected record. This value can further be used for getting the values, downloaded into the selected object field list, for instance, from a regular list data collection, which is stored in the `GridData` list view model property.

Source code

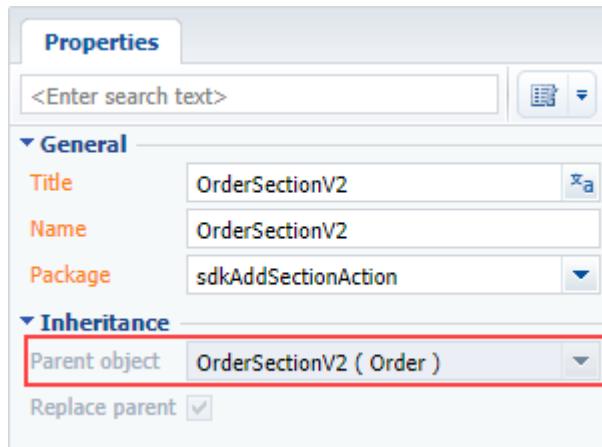
Use this [link](#) to download the case implementation package.

Case implementation algorithm

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

Create a replacing client module and specify the `OrderSectionV2` schema as parent object (Fig. 1). The procedure for creating a replacing page is described in the “**Creating a custom client module schema**” article.

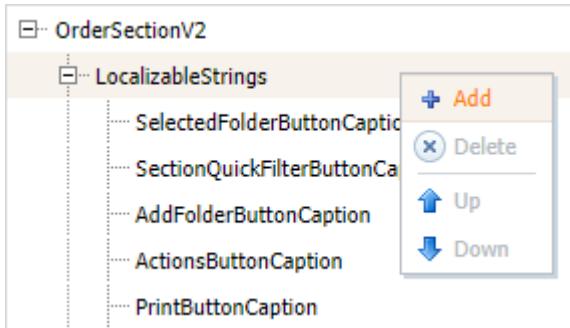
Fig. 1. Properties of the [Orders] section replacing page



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

Create a new localized string (Fig.2).

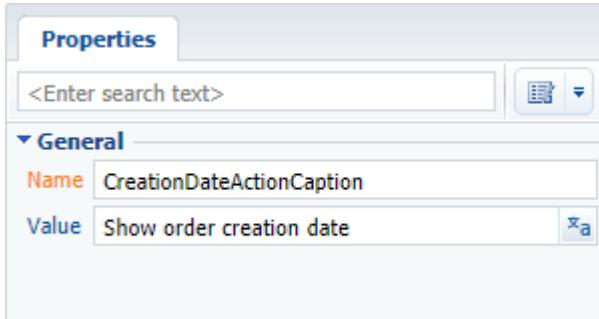
Fig.2 – Adding the localized string to the schema



Populate the following values for the created string (Fig.3):

- [Name] – "CreationDateActionCaption"
- [Value] – "Show order creation date"

Fig. 3. Custom localized string properties



3. Add method implementation to the section view model method collection

- *isRunning()* – verifies if the selected list order has the [In progress] status.
- *isCustomActionEnabled()* – determines if the added menu option is enabled.
- *showOrderInfo()* – the action handler method that displays the selected order estimated completion date in the message window.
- *getSectionActions()* – an overridden parent schema method that gets the section action collection.

The replacing schema source code is as follows:

```
define("OrderSectionV2", ["OrderConfigurationConstants"],
  function(OrderConfigurationConstants) {
    return {
      // Section object schema name.
      entitySchemaName: "Order",
      // Section view model methods.
      methods: {
        // Verifies the order status.
        // activeRowId – the primary column value of the selected list record.
        isRunning: function(activeRowId) {
          // Getting the section list view data collection.
          var gridData = this.get("GridData");
          // Getting the selected order model according to the indicated
          primary column value.
          var selectedOrder = gridData.get(activeRowId);
          // Getting the model property – the selected order status.
          var selectedOrderStatus = selectedOrder.get("Status");
          // The method gets true if the order status is [In Progress].
          Otherwise it gets false.
          return selectedOrderStatus.value ===
          OrderConfigurationConstants.Order.OrderStatus.Running;
        }
      }
    }
  }
);
```

```
        },
        // Determines if the menu option is enabled.
        isCustomActionEnabled: function() {
            // Attempt of getting the active (list selected) record identifier.
            var activeRowId = this.get("ActiveRow");
            // If the identifier is determined and the order status is
            // [In Progress], it gets true, otherwise - it gets false.
            return activeRowId ? this.isRunning(activeRowId) : false;
        },
        // Action handler method. Displays the order creation date in the message
        window.
        showOrderInfo: function() {
            var activeRowId = this.get("ActiveRow");
            var gridData = this.get("GridData");
            // Getting the order creation date. The column must be added to the
            list.
            var dueDate = gridData.get(activeRowId).get("Date");
            // Message window display.
            this.showInformationDialog(dueDate);
        },
        // Overriding the base virtual method that gets the section action
        collection.
        getSectionActions: function() {
            // Calling of the method parent implementation for getting the
            // initiated action collection of the section.
            var actionMenuItems = this.callParent(arguments);
            // Adding a separator line.
            actionMenuItems.addItem(this.getButtonItem({
                Type: "Terrasoft.MenuSeparator",
                Caption: ""
            }));
            // Adding a menu option to the section action list.
            actionMenuItems.addItem(this.getButtonItem({
                // Linking the menu option title to the schema localized string.
                "Caption": {bindTo:
                    "Resources.Strings.CreationDateActionCaption"},
                // Action handler method linking.
                "Click": {bindTo: "showOrderInfo"},
                // Linking of the menu option enabling property to the value that
                gets the isCustomActionEnabled method.
                "Enabled": {bindTo: "isCustomActionEnabled"}
            }));
            // Getting the appended section action collection.
            return actionMenuItems;
        }
    );
});
});
```

After you save the schema and update the application page with clearing the browser cache, a new action appears in the [Orders] section. It will be active when you select an order with the [In progress] status (Fig.4).

Fig. 4. Case result

Order ID	Date	Status	Account Name	Owner	Total Amount	Currency	
ORD-17	5/8/2017 AM	4. Complete d	Factorial Services	Symon Clarke	10,300.00	\$	
ORD-43	4/29/2017 PM	3. In progress	Apex Solutions	Vicky Beaudry	11,290.67	\$	
ORD-16	4/29/2017 7:00 AM	4. Complete d	Streamline Development	Symon Clarke	8,400.00	\$	
ORD-15	4/25/2017 7:00 AM	3. In progress	Apex Solutions	Henry Wayne	Mary King	11,900.00	\$
ORD-14	4/19/2017 7:00 AM	4. Complete d	Streamline Development	Alice Phillips	Mary King	11,725.00	\$

See also

- **Adding an action to the list**
- **How to add a section action: handling the selection of several records**
- **Handling the selection of several records. Examples**

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

[Beginner](#)

[Easy](#)

[Medium](#)

[Advanced](#)

Introduction

The section single record mode is used by default. To select multiple active list records use the [Select multiple records] option in the [Actions] button menu. The list visual view will change – you will see record selection elements appear. To cancel the multiple record mode, click [Cancel multiple selection] in the [Actions] button menu.

Case description

Implement an action, which displays account names of several selected list orders in the message window for the [Orders] section list.

The [Orders] section is available in Sales Creatio products.

The primary column values of the selected records are stored in the *SelectedRows* property of the section view model. These values can further be used for getting the values, downloaded into the selected object field list, for instance, from a regular list data collection, which is stored in the *GridData* list view model property.

Source code

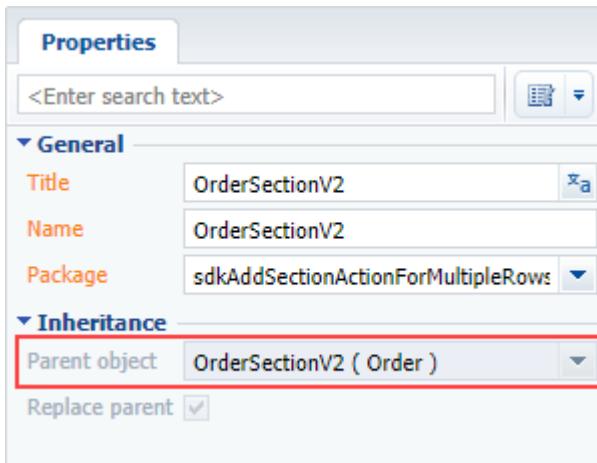
Use this [link](#) to download the case implementation package.

Case implementation algorithm

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

Create a replacing client module and specify the *OrderSectionV2* schema as parent object (Fig. 1). The procedure for creating a replacing page is described in the “**Creating a custom client module schema**” article.

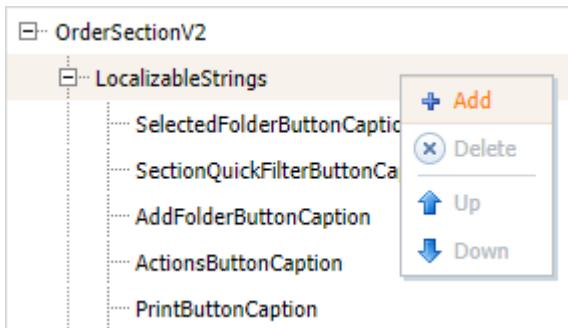
Fig. 1. Properties of the [Orders] section replacing page



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

Create a new localized string (Fig.2).

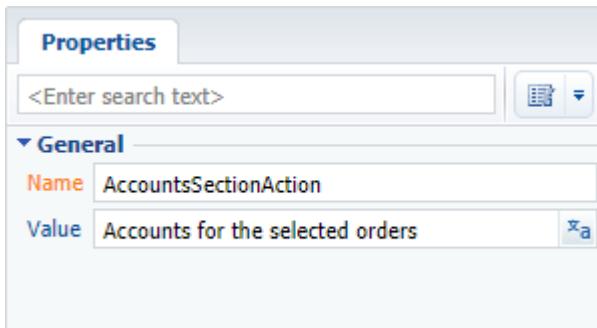
Fig. 2 – Adding the localized string to the schema



Populate the following values for the created string (Fig.3):

- [Name] – “AccountsSectionAction”
- [Value] – “Accounts for the selected orders”

Fig. 3. Custom localized string properties



3. Add method implementation to the section view model method collection

- `isCustomActionEnabled()` – determines if the added menu option is enabled.
- `showOrderInfo()` – the action handler method that displays the selected order account list in the message window.
- `getSectionActions()` – an overridden parent schema method that gets the section action collection.

The replacing schema source code is as follows:

```
define("OrderSectionV2", ["OrderConfigurationConstants"],
```

```
function(OrderConfigurationConstants) {
    return {
        // Section schema name.
        entitySchemaName: "Order",
        // Section view model methods.
        methods: {
            // Determines if the menu option is enabled.
            isCustomActionEnabled: function() {
                // Attempt of getting the selected record identifier array.
                var selectedRows = this.get("SelectedRows");
                // If the array contains any elements (at least one list record is
selected),
                // it gets true, otherwise - it gets false.
                return selectedRows ? (selectedRows.length > 0) : false;
            },
            // Action handler method. Displays the account list in the message
window.
            showOrdersInfo: function() {
                // Getting the selected record identifier array.
                var selectedRows = this.get("SelectedRows");
                // Getting the list record data collection.
                var gridData = this.get("GridData");
                // Variable for storage of the selected order object model.
                var selectedOrder = null;
                // Variable for storage of the selected order account name.
                var selectedOrderAccount = "";
                // Variable for the message window text.
                var infoText = "";
                // Handling of the selected section record identifier array.
                selectedRows.forEach(function(selectedRowId) {
                    // Getting the selected order object model.
                    selectedOrder = gridData.get(selectedRowId);
                    // Getting the selected order account name. The column must be
added to the list.
                    selectedOrderAccount = selectedOrder.get("Account").displayValue;
                    // Adding the account name to the message window text.
                    infoText += "\n" + selectedOrderAccount;
                });
                // Message window display.
                this.showInformationDialog(infoText);
            },
            // Overriding the base virtual method that gets the section action
collection.
            getSectionActions: function() {
                // Calling of the method parent implementation for getting the
                // initiated action collection of the section.
                var actionMenuItems = this.callParent(arguments);
                // Adding a separator line.
                actionMenuItems.addItem(this.getButtonMenuItem({
                    Type: "Terrasoft.MenuSeparator",
                    Caption: ""
                }));
                // Adding a menu option to the section action list.
                actionMenuItems.addItem(this.getButtonMenuItem({
                    // Linking the menu option title to the schema localized string.
                    "Caption": {bindTo: "Resources.Strings.AccountsSectionAction"},
                    // Action handler method linking.
                    "Click": {bindTo: "showOrdersInfo"},
                    // Linking of the menu option enabling property to the value that
gets
                    // the isCustomActionEnabled method.
                    "Enabled": {bindTo: "isCustomActionEnabled"},
```

```
// Multiselection mode enabling.  
"IsEnabledForSelectedAll": true  
});  
// Getting the appended section action collection.  
return actionMenuItems;  
}  
};  
};  
});
```

After you save the schema and update the application page with clearing the browser cache, a new action appears in the [Orders] section. It will be active when you select orders in the multiple record selection mode (Fig.4).

Fig. 4. Case result

Actions (4)		Recent Orders			
		Order ID	Date	Owner	Total
<input type="checkbox"/>	ORD-19	4.	Completed	Alpha Business	John Best 22,950.00 \$
<input checked="" type="checkbox"/>	ORD-18	4.	Completed	Durable Industries	Virginia L. Quinn 340.00 \$
<input checked="" type="checkbox"/>	ORD-17	4.	Completed	Novelty	Symon Clarke 10,300.00 \$
<input checked="" type="checkbox"/>	ORD-43	1.	Draft	Nuclon	Vicky Beaudry Symon Clarke 11,290.67 \$

See also

- **Adding an action to the list**
 - **How to add a section action: handling the selection of a single record**
 - **Handling the selection of several records. Examples**

Handling the selection of several records. Examples

Beginner **Easy** **Medium** **Advanced**

Introduction

Before you start implementing cases it is recommended to study the "**How to add a section action: handling the selection of several records**" article.

Example

Case description

Implement action for the [Activities] section list that will set the [Completed] status for several selected list activities.

Source code

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

Case implementation algorithm

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

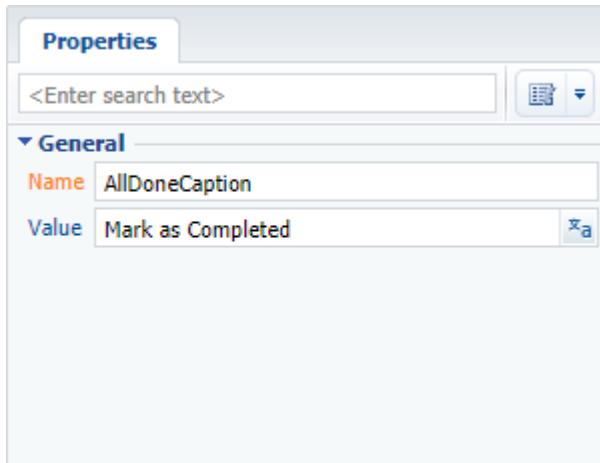
The procedure for creating a replacing page is described in the “[Creating a custom client module schema](#)” article.

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

Populate the following values for the created string (Fig.1):

- [Name] – "AllDoneCaption";
- [Value] – "Mark as \"Completed\"".

Fig. 1. Properties of the custom localizable string



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

- *isCustomActionEnabled()* – the method that determines if the added menu option is enabled.
- *setAllDone()* – the action handler method that sets the [Completed] status for several selected list activities.
- *getSectionActions()* – an overridden parent schema method that gets the section action collection.

The replacing schema source code is as follows:

```
define("ActivitySectionV2", ["ConfigurationConstants"],
  function(ConfigurationConstants) {
    return {
      // Section schema name.
      entitySchemaName: "Activity",
      // Section view model methods.
      methods: {
        // Defines if the menu option is enabled.
        isCustomActionEnabled: function() {
          // Attempt to receive the selected record identifier array.
          var selectedRows = this.get("SelectedRows");
          // If the array contains some elements (at least one of the
          records is selected from the list),
          // it returns true, otherwise - false.
          return selectedRows ? (selectedRows.length > 0) : false;
        },
        // Action handler method. Sets the [Completed] status for the
        selected records.
      }
    }
  }
);
```

```
setAllDone: function() {
    // Receiving the selected record identifier array.
    var selectedRows = this.get("SelectedRows");
    // The procession starts if at least one record is selected.
    if (selectedRows.length > 0) {
        // Creation of the batch query class instance.
        var batchQuery = this.Ext.create("Terrasoft.BatchQuery");
        // Update of each selected record.
        selectedRows.forEach(function(selectedRowId) {
            // Creation of the UpdateQuery class instance with the
Activity root schema.
            var update = this.Ext.create("Terrasoft.UpdateQuery", {
                rootSchemaName: "Activity"
            });
            // Applying filter to determine the record for update.
            update.enablePrimaryColumnFilter(selectedRowId);
            // The "Success" value is set to the Status column via
            // the ConfigurationConstants.Activity.Status.Done.
            update.setParameterValue("Status",
ConfigurationConstants.Activity.Status.Done,
this.Terrasoft.DataValueType.GUID);
                // Adding a record update query to the batch query.
                batchQuery.add(update);
            }, this);
            // Batch query to the server.
            batchQuery.execute(function() {
                // Record list update.
                this.reloadGridData();
            }, this);
        }
    },
    // Overriding the base virtual method, returning the section action
collection.
getSectionActions: function() {
    // Calling of the parent method implementation,
    // returning the initialized section action collection.
    var actionMenuItems = this.callParent(arguments);
    // Adding separator line.
    actionMenuItems.addItem(this.getButtonItem({
        Type: "Terrasoft.MenuSeparator",
        Caption: ""
    }));
    // Adding a menu option to the section action list.
    actionMenuItems.addItem(this.getButtonItem({
        // Binding the menu option title to the localized schema
string.
        "Caption": { bindTo: "Resources.Strings.AllDoneCaption" },
        // Binding of the action handler method.
        "Click": { bindTo: "setAllDone" },
        // Binding the menu option enable property to the value that
returns the isCustomActionEnabled method.
        "Enabled": { bindTo: "isCustomActionEnabled" },
        // Multiselection mode enabling.
        "IsEnabledForSelectedAll": true
    });
    // Returning of the added section action collection.
    return actionMenuItems;
}
};

});
```

After saving the schema and updating the app page with clearing the cache you will be able to apply the [Completed] status to several selected activities in the [Activities] section by using the new [Mark as Completed] action.

Fig. 2. Case result demonstration

The screenshot shows the 'Activities' section of the Creatio application. A context menu is open over a list of activities, with the 'Mark as Completed' option highlighted and enclosed in a red box. The activities listed are:

- 1. Project of customer acquisition**: End date 1/3/2018 6:30 PM, Account Alpha Business, Status Not started, Category Paper work.
- 2. Verify payments**: End date 1/5/2018 2:00 PM, Status Completed, Category Meeting, Result Information received.
- 3. Prepare specifications**: End date 1/16/2018 3:00 PM, Account Apex Solutions, Status Not started, Category Meeting.
- 4. Streamline Development**: End date 1/15/2018 5:00 PM, Account Streamline Development, Status Not started, Category Call.
- 5. Clearsoft**: End date 1/8/2018 5:30 PM, Account Clearsoft, Status Not started, Category Paper work.

Example 2

Case description

Implement action for the [Activities] section list that will call the record owner selection window and set the selected value for several selected list activities.

Source code

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

Case implementation algorithm

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

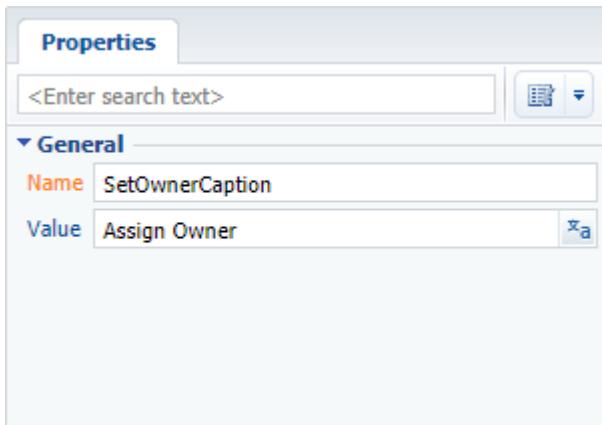
The procedure for creating a replacing page is described in the “[Creating a custom client module schema](#)” article.

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

Populate the following values for the created string (Fig.3):

- [Name] – "SetOwnerCaption";
- [Value] – "Assign Owner".

Fig. 3. Properties of the custom localizable string



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

- *isCustomActionEnabled()* – the method that determines if the added menu option is enabled.
- *setOwner()* – the action handler method that calls opening of the [Contacts] lookup.
- *lookupCallback()* – the callback-method that sets the lookup selected contact as the owner for the selected list records.
- *getSectionActions()* – an overridden parent schema method that gets the section action collection.

The replacing schema source code is as follows:

```
define("ActivitySectionV2", ["ConfigurationConstants"],
  function(ConfigurationConstants) {
    return {
      // Section schema name.
      entitySchemaName: "Activity",
      // Section view model methods.
      methods: {
        // Defines if the menu option is enabled.
        isCustomActionEnabled: function() {
          // Attempt to receive the selected record identifier array
          var selectedRows = this.get("SelectedRows");
          // If the array contains some elements (at least one of the
records is selected from the list),
          // it returns true, otherwise - false.
          return selectedRows ? (selectedRows.length > 0) : false;
        },
        // Action handler method. Opens the [Contacts] lookup.
        setOwner: function() {
          // Defining the lookup configuration.
          var config = {
            // The [Contact] Schema.
            entitySchemaName: "Contact",
            // Multiple selection is disabled.
            multiSelect: false,
            // The displayed column - [Name].
            columns: ["Name"]
          };
          // Opening of the lookup with certain configuration and call-back
function that is triggered
          // after you click [Select].
          this.openLookup(config, this.lookupCallback, this);
        },
        // Sets the lookup selected contact as the owner
        // for the selected list records.
        lookupCallback: function(args) {
```

```
// The selected lookup record identifier.
var activeRowId;
// Receiving of the lookup selected records.
var lookupSelectedRows = args.selectedRows.getItems();
if (lookupSelectedRows && lookupSelectedRows.length > 0) {
    // Receiving of the first lookup selected record Id.
    activeRowId = lookupSelectedRows[0].Id;
}
// Receiving of the selected record identifier array.
var selectedRows = this.get("SelectedRows");
// The procession starts if at least one record is selected from
the list and the owner is selected
// in the lookup.
if ((selectedRows.length > 0) && activeRowId) {
    // Creation of the batch query class instance.
    var batchQuery = this.Ext.create("Terrasoft.BatchQuery");
    // Update of each selected record.
    selectedRows.forEach(function(selectedRowId) {
        // Creation of the UpdateQuery class instance with the
Activity root schema.
        var update = this.Ext.create("Terrasoft.UpdateQuery", {
            rootSchemaName: "Activity"
        });
        // Applying filter to determine the record for update.
        update.enablePrimaryColumnFilter(selectedRowId);
        // The [Owner] column is populated with the value that
equals to
        // the lookup selected contact id.
        update.setParameterValue("Owner", activeRowId,
this.Terrasoft.DataValueType.GUID);
        // Adding a record update query to the batch query.
        batchQuery.add(update);
    }, this);
    // Batch query to the server.
    batchQuery.execute(function() {
        // Record list update.
        this.reloadGridData();
    }, this);
}
},
// Overriding the base virtual method, returning the section action
collection.
getSectionActions: function() {
    // Calling of the parent method implementation,
    // returning the initialized section action collection.
    var actionMenuItems = this.callParent(arguments);
    // Adding separator line.
    actionMenuItems.addItem(this.getButtonMenuItem({
        Type: "Terrasoft.MenuSeparator",
        Caption: ""
    }));
    // Adding a menu option to the section action list.
    actionMenuItems.addItem(this.getButtonMenuItem({
        // Binding the menu option title to the localized schema
string.
        "Caption": { bindTo: "Resources.Strings.SetOwnerCaption" },
        // Binding of the action handler method.
        "Click": { bindTo: "setOwner" },
        // Binding the menu option enable property to the value that
returns the isCustomActionEnabled method.
        "Enabled": { bindTo: "isCustomActionEnabled" },
        // Multiselection mode enabling.
    })
}
```

```
        "IsEnabledForSelectedAll": true
    }));
    // Returning of the added section action collection.
    return actionMenuItems;
}
};

});
```

After saving the schema and updating the app page with clearing the cache you will be able to change the owner of several selected activities in the [Activities] section by using the new [Assign Owner] action.

Fig. 4. Case result demonstration

Activities

ACTIONS (3) ▾

Synchronize activities < Due date > X Employee Filter Tag

Activity	End Date	Account	Status	Category
Project	1/1/2018 8:00 PM	Milestone Consulting	Completed	Call
	1/2/2018 2:00 PM		Completed	Call
Prepare presentation	1/3/2018 4:00 PM	Parsons & Co	Completed	Paper work
<input checked="" type="checkbox"/> Presentation	1/4/2018 1:30 PM	Durable Industries	Completed	Meeting
<input checked="" type="checkbox"/> Prepare quotation	1/4/2018 5:00 PM	Infocom	Completed	Paper work

What can I do for you? >

Creatio

VIEW ▾

Actions:

- Synchronize activities
- Cancel multiple selection
- Deselect all
- Select all
- Export list to file
- Delete
- Assign Owner

Notifications:

- 4

See also

- **Adding an action to the list**
 - **How to add a section action: handling the selection of a single record**
 - **How to add a section action: handling the selection of several records**

How to add a button to a section

Beginner **Easy** **Medium** **Advanced**

Introduction

During the process of section customization, you may need to create a custom action and add the appropriate button to the section. For this, use the container of action buttons (`ActionButtonsContainer`) with the new record button and the button with drop-down action list. More information can be found in the [“Section actions”](#) article.

To add a custom button to the view model, you need to change the following properties:

- *diff* array of configuration objects. Add a configuration object for setting up location of the component on the edit page.
 - *methods* collection. Add an implementation of the handler method, that will be called on button click, and other auxiliary methods necessary for the work of the control. These can be methods that will regulate the

visibility or availability of the control, depending on the conditions.

More information about button visualization can be found in the “**Adding a button to the edit page**” article.

Case description

Add a button to the [Accounts] section. The button should open the edit page of the primary contact of the account selected in the list.

Access to the selected record is performed through the *ActiveRow* attribute of the section view model, that returns the value of the primary column of selected record. This value can be used to get values loaded into the list of the fields of selected object, for example, from the list data collection that is stored in the *GridData* property of the list view model.

Source code

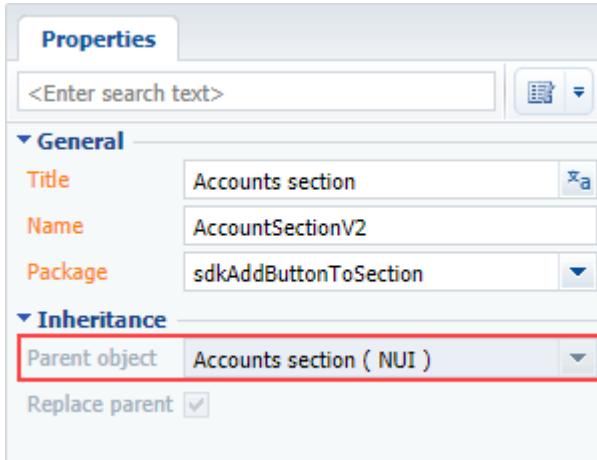
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Create a replacing edit page of the [Accounts] section

Create a replacing client module and specify the [Accounts section] schema as parent object (Fig. 1). The procedure for creating a replacing page is covered in the “**Creating a custom client module schema**” article.

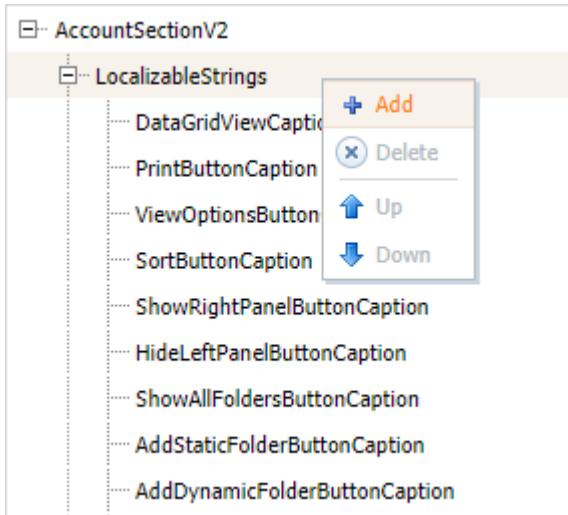
Fig. 1. Properties of the [Accounts] section replacing schema



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

Create a new localizable string (Fig. 2).

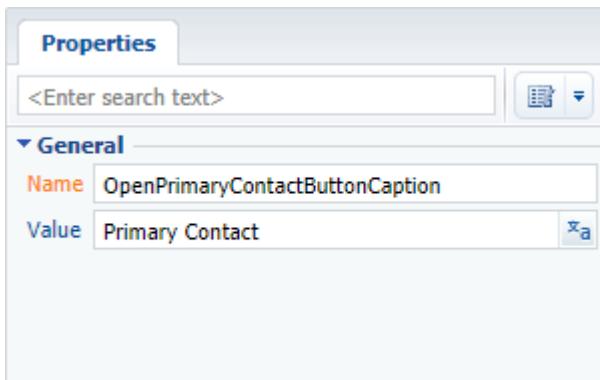
Fig.2. Adding localizable string to the schema



For the created string specify (Fig. 3):

- [Name] – "OpenPrimaryContactButtonCaption"
- [Value] – "Primary Contact".

Fig. 3. Properties of the custom localizable string



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

- *isAccountPrimaryContactSet()* – checks if the [Primary contact] field is filled.
- *onOpenPrimaryContactClick()* – button click handler method. Opens the edit page of the primary contact.

4. Add a configuration object with the settings determining the button position in the *diff* array

Add an object with the settings determining the button position on the page in the *diff* array.

The replacing schema source code is as follows:

```
define("AccountSectionV2", [], function() {
    return {
        // Name of the section object schema.
        entitySchemaName: "Account",
        // Method of the section view model.
        methods: {
            // Button click handler method.
            onOpenPrimaryContactClick: function() {
                // Getting the id of the selected record.
                var activeRow = this.get("ActiveRow");
                if (!activeRow) {
                    return;
                }
            }
        }
    }
});
```

```
        }
        // Defining the id of the primary contact.
        var primaryId =
this.get("GridData").get(activeRow).get("PrimaryContact").value;
        if (!primaryId) {
            return;
        }
        // Creation of the address string.
        var requestUrl = "CardModuleV2/ContactPageV2/edit/" + primaryId;
        // Publication a message about updating the navigation history of
pages and
        // opening to the primary contact edit page.
        this.sandbox.publish("PushHistoryState", {
            hash: requestUrl
        });
    },
    // Checks if the [Primary Contact] field of the selected item is filled.
    isAccountPrimaryContactSet: function() {
        var activeRow = this.get("ActiveRow");
        if (!activeRow) {
            return false;
        }
        var pc = this.get("GridData").get(activeRow).get("PrimaryContact");
        return (pc || pc !== "") ? true : false;
    }
},
//Display button in the section.
diff: /**SCHEMA_DIFF*/[
    // Metadata for adding a custom button to a section.
    {
        // The operation of adding a component to the page is in progress..
        "operation": "insert",
        // The meta name of the parent container to which the button is
added.
        "parentName": "ActionButtonsContainer",
        // The button is added to the parent component's collection.
        "propertyName": "items",
        // The meta-name of the button to be added.
        "name": "MainContactSectionButton",
        // Properties passed to the component's constructor.
        "values": {
            // The type of the component to add is the button.
            itemType: Terrasoft.ViewItemType.BUTTON,
            // Bind the button header to the localizable string of the
schema.
            caption: { bindTo:
"Resources.Strings.OpenPrimaryContactButtonCaption" },
            // Bind the button click handler method.
            click: { bindTo: "onOpenPrimaryContactClick" },
            // Binding the button availability property.
            enabled: { bindTo: "isAccountPrimaryContactSet" },
            // Setting the location of the button.
            "layout": {
                "column": 1,
                "row": 6,
                "colSpan": 1
            }
        }
    }
]/**SCHEMA_DIFF*/
};

});
```

After saving the schema, clearing the browser cache and updating the application page, the [Primary Contact] button will be displayed in the [Accounts] section. The button will be enabled after selecting the account with the specified primary contact (Fig. 4).

Fig. 4. Case result

Account Name	Primary contact	Web	Address	Primary phone	Type
XT Group	Jason Robinson	www.xtg.au	148 Bunda St, Canberra ACT 2601	+61 2 6247 5656	Customer
Global Venture	Zane Rogers	www.globalventure.com.ny	412 Pine Street, New York, NY	+1 212 721 1810	Customer
Feature IT	Tony Campbell	www.feature-it.com	85 46th Street	+1 212 735 2537	Partner
Fast Works	Kate Roberts	www.fast-works.com		+1 212 775 9836	Customer

How to highlight a record in the list in color

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

Introduction

Creatio enables you to configure the layout of the record list by highlighting some records when a specific condition is met. This setting helps to highlight the records that require special attention.

The display of the list record is controlled by the *customStyle* property of the record list.

The *customStyle* property is an object, which properties are similar to CSS properties and generate style of displaying a list record. Example:

```
item.customStyle = {
    // The text color is white.
    "color": "white",
    // The background color is orange.
    "background": "orange"
};
```

Follow these steps to configure the display of individual list records:

- Override the *prepareResponseCollectionItem(item)*, base method in the replacing schema of the section. This method modifies a data string before loading it to the list.
- In the *prepareResponseCollectionItem(item)* method, implement the assigning of specific value to the *customStyle* property for the necessary list records.

Case description

For the [Orders] section, implement highlighting the list records with the [In progress] status.

The [Orders] section is available in Sales Creatio products.

Source code

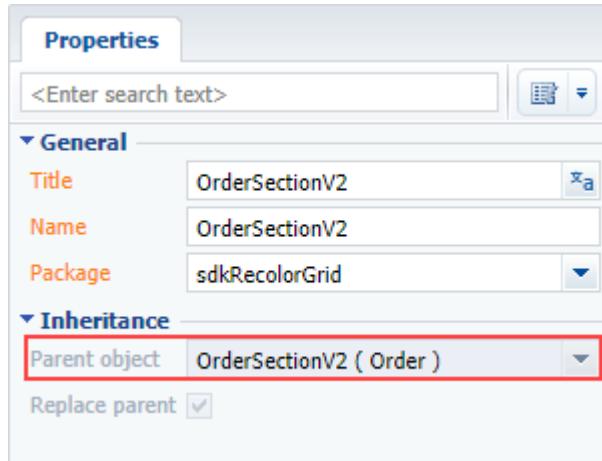
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

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

Create a replacing client module and specify the *OrderSectionV2* schema as parent object (Fig. 1). The procedure for creating a replacing page is covered in the “**Creating a custom client module schema**” article.

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



2. Override the `prepareResponseCollectionItem` method

Add the `prepareResponseCollectionItem()` to the method collection of the created schema. The method overrides the base method, modifies the data string before uploading it to the list, and adds custom styles to the specific list records.

The replacing schema source code is as follows:

```
define("OrderSectionV2", ["OrderConfigurationConstants"],
function(OrderConfigurationConstants) {
    return {
        // The name of the section scheme.
        entitySchemaName: "Order",
        // Methods of the section representation model.
        methods: {
            // Override the base method, which modifies the data string before it is
            // loaded into the list.
            prepareResponseCollectionItem: function(item) {
                // Calling the base method.
                this.callParent(arguments);
                item.customStyle = null;
                // Determining the order status.
                var running = item.get("Status");
                // If the status of the order is "In progress", the record style
                // changes.
                if (running.value ===
                    OrderConfigurationConstants.Order.OrderStatus.Running) {
                    item.customStyle = {
                        // The text color is white.
                        "color": "white",
                        // The background color is green.
                        "background": "#8ecb60"
                    };
                }
            }
        }
    }
});
```

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

After saving the schema, clearing the browser cache and updating the application page, the orders in the [Orders] section with the [In progress] status will be highlighted in green (Fig. 2).

Fig. 2. Case result

The screenshot shows a CRM application interface. On the left, a vertical sidebar lists navigation items: Sales, Dashboards, Feed, Leads, Accounts, Contacts, Activities, Opportunities, and Orders. The Orders item is currently selected and highlighted in orange. At the top center, the word "Orders" is displayed above a toolbar containing icons for creating a new order, filtering, and viewing. Below the toolbar is a search bar with the placeholder "What can I do for you?". To the right of the search bar is the brand name "Creatio". The main content area displays a table of order records. The table has columns for Order ID, Start Date, Due Date, Status, Company, Owner, Amount, and Currency. The table rows are color-coded in green and white. A red notification badge with the number "4" is visible on the Activities icon in the sidebar.

	<Start date>	<Due date>	Owner	Filter	Tag		
1	7						
ORD-32	4/16/2017 3:00 AM	1. Draft	Alpha Business	Jordan Anderson	John Best	18,840.00	\$
ORD-29	4/13/2017 3:00 AM	4. Completed	Alpha Business		Mary King	4,800.00	\$
ORD-35	4/13/2017 3:00 AM	3. In progress	Infocom	Barber Andrew	Mary King	9,415.00	\$
ORD-30	4/13/2017 3:00 AM	3. In progress	Apex Solutions	Henry Wayne	John Best	1,140.00	\$
ORD-34	4/13/2017 3:00 AM	3. In progress	Build Technologies		Mary King	4,400.00	\$
ORD-42	4/12/2017 2:00 PM	3. In progress	Planet Soft	Scot Grammer	Mary King	7,122.19	\$
ORD-11	4/12/2017 2:00 AM	4. Completed	Alpha Business	Alexander Wilson	John Best	10,000.00	\$
ORD-27	4/11/2017 4:00 AM	2. Confirmation	Alpha Business	Jordan Anderson	John Best	9,000.00	\$

Adding quick filter block to a section

Beginner **Easy** **Medium** **Advanced**

Introduction

Filters are designed to search and filter records in sections. In Creatio quick, standard and advanced filters and folders are provided. More information can be found in the "[Filters](#)" article.

To add a block of quick filters to the section, override the `initFixedFiltersConfig()` method in the replacement schema, create the `fixedFilterConfig` configuration object in this method with the following properties:

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

Assign a reference to the created configuration object to the `fixedFiltersConfig` view model attribute:

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

Case description

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

Source code

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

Case implementation algorithm

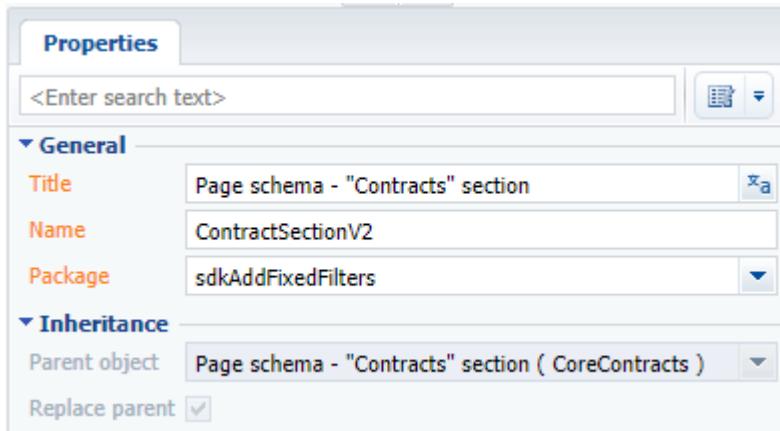
1. Create a replacing schema of the [Contracts] section in the custom package.

Create a replacing custom module and populate its properties with (Fig.1):

- [Parent object] – “Page schema – “Contracts” section”;
- [Name] – *ContractSectionV2*.

The procedure for creating a replacing client schema is covered in the “[Creating a custom client module schema](#)” article.

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

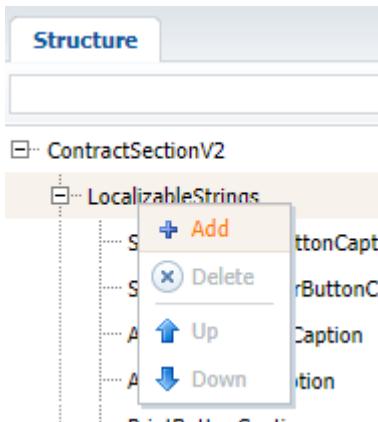


2. Add localizable strings to the schema structure.

Create two new localizable strings (Fig.2) with the following properties:

Name	Value
OwnerFilterCaption	Owner
PeriodFilterCaption	Period

Fig. 2. Adding localizable string to the schema



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

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

The replacing schema source code is as follows:

```
define("ContractSectionV2", ["BaseFiltersGenerateModule"],
function(BaseFiltersGenerateModule) {
    return {
        // Name of the section schema
        entitySchemaName: "Contract",
        // Method collection of the section view model.
        methods: {
```

```

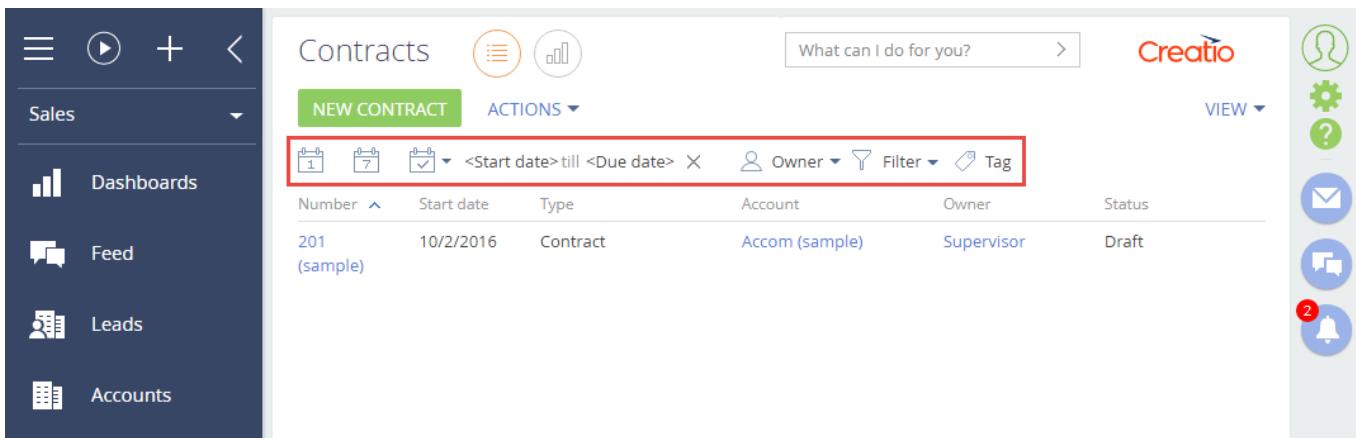
// Initializes the fixed filters.
initFixedFiltersConfig: function() {
    // Creating a Configuration Object.
    var fixedFilterConfig = {
        // The schema of the section object is specified as an object
schema for fixed filters.
        entitySchema: this.entitySchema,
        // Array of filters.
        filters: [
            // Start period filter.
            {
                // The name of the filter.
                name: "PeriodFilter",
                // Filter header.
                caption:
this.get("Resources.Strings.PeriodFilterCaption"),
                // The data type - date.
                dataValueType: this.Terrasoft.DataValueType.DATE,
                // Start date of the filtering period.
                startDate: {
                    // Filter the data from the [Date] column.
                    columnName: "StartDate",
                    // Default value.
                    defValue: this.Terrasoft.startOfWeek(new Date())
                },
                // Date of the filtering period completion.
                dueDate: {
                    columnName: "StartDate",
                    defValue: this.Terrasoft.endOfWeek(new Date())
                }
            },
            // Owner filter.
            {
                // The name of the filter.
                name: "Owner",
                // Filter header.
                caption:
this.get("Resources.Strings.OwnerFilterCaption"),
                // Filter the data from the [Owner] column.
                columnName: "Owner",
                // Current user contact is specified as default value.
                // Value is received from the system setting.
                defValue: this.Terrasoft.SysValue.CURRENT_USER_CONTACT,
                // The data type - lookup.
                dataValueType: this.Terrasoft.DataValueType.LOOKUP,
                // Filter.
                filter: BaseFiltersGenerateModule.OwnerFilter
            }
        ]
    };
    // A link to the configurational object is assigned to the
[FixedFilterConfig] column.
    this.set("FixedFilterConfig", fixedFilterConfig);
}
};

}) ;

```

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

Fig. 3. Case result



Deleting a section

Beginner **Easy** **Medium** **Advanced**

Introduction

If a section was created based on the object that must be deleted, this section must be deleted first.

To delete a custom section in Creatio, you need access to the system configuration tools and its database.

Before you delete a custom section, unlock the corresponding section in the SVN storage.

First, delete the database records. Use the following script to delete a section:

```
DECLARE @UID UNIQUEIDENTIFIER
DECLARE @ModuleEntityUID UNIQUEIDENTIFIER;
DECLARE @ModuleID UNIQUEIDENTIFIER;
DECLARE @Name NVARCHAR(max) = 'ToDelete';
select @UID = UId from SysSchema where Name Like @Name
select @ModuleEntityUID = Id from SysModuleEntity where
SysEntitySchemaUID = @UID
select @ModuleID = Id from SysModule where SysModuleEntityId = @ModuleEntityUID;
delete from SysModuleInWorkplace where SysModuleId = @ModuleID;
delete from SysModule where Id = @ModuleID;
delete from SysModuleEdit where SysModuleEntityId = @ModuleEntityUID;
delete from SysModuleEntity where Id = @ModuleEntityUID;
delete from SysDetail where EntitySchemaUID = @UID;
delete from SysLookup where SysEntitySchemaUID = @UID;
delete from [Lookup] where SysEntitySchemaUID = @UID;
```

Please note that the “ToDelete” value must be replaced with the custom section schema name. After deleting database records, delete section custom schemas in the [Configuration] section (located under [Advanced settings]) in the following order:

1. ToDeleteFile
2. ToDeleteInFolder
3. ToDeleteInTag
4. ToDeleteTag
5. ToDeleteFolder
- 6.ToDelete

Page configuration

Contents

- **Introduction**
- **Setting the edit page fields using business rules**
- **Adding an action to the edit page**
- **Control elements**
- **Adding calculated fields**
- **Adding an action panel**
How to set a default value for a field
- **How to add the field validation**
- **Using filtration for lookup fields. Examples**
- **Adding an action panel**
- **Adding a new channel to the action panel**
- **Displaying contact's time zone**
- **How to display the difference between dates on edit page fields**
- **How to block fields of the edit page**

Page configuration

Beginner

Easy

Medium

Advanced

Introduction

An edit page is a container having a number of fields for entering and changing the columns of section object schema (see “**Section list**”). It opens when you add a new record to the section list, or when you edit the existing record. Every section has one or several edit pages.

Business rules are one of the tools to setup page logic in Creatio.

Business rules represent a standard Creatio mechanism that enables you to set up the page field behavior by configuring the view model columns.

Business rules enable you to:

- hide and display fields
- lock and unlock fields
- make fields required or optional
- filter the lookup field value depending on another field value

The **primary control elements** of a page include:

- an input field
- a button
- an image field
- a color button
- a multicurrency field

Creatio enables you to add and edit standard control elements on the edit page as well as to create custom control elements.

Page configuration options enable setting up the behavior of the existing control elements on the page:

- **add validation**
- **set up calculated fields**
- **apply filtration to lookup fields**
- **setting default values for fields**

Contents

- **Setting the edit page fields using business rules**
- **Adding an action to the edit page**
- **Control elements**
- **Adding an action panel**

- Adding a new channel to the action panel
- Adding calculated fields
- How to set a default value for a field
- How to add the field validation
- Using filtration for lookup fields. Examples
- Adding an action panel
- Adding a new channel to the action panel
- Displaying contact's time zone
- How to display the difference between dates on edit page fields
- How to block fields of the edit page

Setting the edit page fields using business rules

Contents

- Introduction
- The FILTRATION rule use case
- The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition
- The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition
- The BINDPARAMETER rule. How to make a field required based on a specific condition
- Business rules created via wizards

Setting the edit page fields using business rules

Beginner

Easy

Medium

Advanced

Introduction

Business rules are one of the tools to setup page logic in Creatio.

Business rules are a standard Creatio mechanism that enables you to set up the page field behavior by configuring the view model columns.

Business rules enable you to:

- hide and display fields
- lock and unlock fields
- make fields required or optional
- filter the lookup field value depending on another field value

You can add business rules in two ways:

1. Via section wizard or detail wizard.

- Wizard generated business rules are added to the *businessRules* client module property.
- The generated business rules have higher priority at execution.
- BusinessRuleModule enumerations are not used when describing the generated business rules.

See the "[Setting up business rules](#)" article for more information on business rule setup via wizard. The manual setup of the generated business rules is covered in the "**Business rules. The businessRules property**" article.

2. Via configuring the "rules" property of the client module schema.

The functions of business rules are implemented in the *BusinessRuleModule* client module. To use these functions, add the *BusinessRuleModule* module to the list of user schema dependencies of the view model.

Case of declaring user module with using business rules

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

```

        return {
            // View model schema implementation.
        };
    });
}

```

Capability of adding business rules by developer tools ensures compatibility with previous versions.

General requirements to business rule declaring in a client module

- All rules are described in the *rules* property of the page view model.
- The rules are applied to view model columns and not to control elements.
- Business rules are not supported on list pages.
- Rules have names.
- Rule parameters are set in its configuration object.

Examples of business rule declaring

```

// List of view model rules.
rules : {
    // Name of the column where the rule is added.
    "FirstColumnName" : {
        // FirstColumnName column list of rules.
        // Rule name.
        FirstRuleName : {
            // FirstRuleName configuration object.
            ruleType: <BusinessRuleModule.enums.RuleType enumeration value>
            // The rest of rule configuration properties.
        },
        SecondRuleName: {
            // SecondRuleName configuration object.
            ruleType: <BusinessRuleModule.enums.RuleType enumeration value>
            // The rest of rule configuration properties.
        }
    },
    "SecondColumnName" : {
        // SecondColumnName column list of rules.
        ...
    }
}

```

Business rule types

Rule types are defined in the *RuleType* enumeration of the *BusinessRuleModule* module.

Currently two rule types are used – BINDPARAMETER and FILTRATION.

BINDPARAMETER

This rule type is used for linking properties of a column to values of different parameters. For instance, for setting up the visibility of a column or to enable a column depending on the value of another column. The main BINDPARAMETER configuration object properties are described in table 1.

Table 1. – BINDPARAMETER configuring

Property	Value
<i>ruleType</i>	Type of the rule. It is defined by the <i>BusinessRuleModule.enums.RuleType</i> enumeration value. In this case BINDPARAMETER type is used.
<i>property</i>	Control element property. Set by the <i>BusinessRuleModule.enums.Property</i> enumeration value:

Property	Value
	<ul style="list-style-type: none"> • VISIBLE – column visibility • ENABLED – enabling of a column • REQUIRED – column populating is required • READONLY – column for reading only
<i>conditions</i>	<p>Condition array for rule application.</p> <p>Every condition represents a configuration object, whose properties are described in table 2.</p>
<i>logical</i>	Logical operation of combining the conditions from the <i>conditions</i> property. Set by the <i>Terrasoft.LogicalOperatorType</i> enumeration value.

Table 2. – BINDPARAMETER configuring

Property	Value
<i>leftExpression</i>	Expression of the left side of the condition. Represents a configuration object with the following properties: <ul style="list-style-type: none"> • <i>Type</i> – expression type. Set by the <i>BusinessRuleModule.enums.ValueType</i> enumeration value: <ul style="list-style-type: none"> • CONSTANT – constant value • ATTRIBUTE – the view model column value • SYSSETTING – system setting • SYSVALUE – system value The <i>Terrasoft.core.enums.SystemValueType</i> system value list element. • <i>Attribute</i> – model column name • <i>attributePath</i> – meta-path to the lookup schema column • <i>Value</i> – value for comparison
<i>comparisonType</i>	Type of comparison. Set by the <i>Terrasoft.core.enums.ComparisonType</i> enumeration value.
<i>rightExpression</i>	Expression of the right side of the condition. Similar to <i>leftExpression</i> .

FILTRATION

Use the FILTRATION rule to set up filtering of values in view model columns. For example, you can filter a lookup column depending on the current status of a page.

Table 3. – FILTRATION configuring

Property	Value
<i>ruleType</i>	Rule type. Set by the <i>BusinessRuleModule.enums.RuleType</i> .enumeration value. In this case FILTRATION type is used.
<i>autocomplete</i>	Reverse filtering checkbox. Can take the <i>true</i> or <i>false</i> values.
<i>autoClean</i>	The checkbox of automated value cleaning upon changing the column that is used for filtration. Can take the <i>true</i> or <i>false</i> values.
<i>baseAttributePatch</i>	Meta-path to the lookup schema column that will be used for filtration. The feedback principle is applied when building the column path similar to EntitySchemaQuery ,. The path is generated in relation to the schema, referred to by the model column.
<i>comparisonType</i>	Type of comparison operation. Set by the <i>Terrasoft.ComparisonType</i> .enumeration value.

Property	Value
<i>type</i>	The value type for comparison <i>baseAttributePatch</i> . Set by the <i>BusinessRuleModule.enums.ValueType</i> enumeration value.
<i>attribute</i>	The view model column name. This property is described if ATTRIBUTE value type (<i>type</i>) is indicated.
<i>attributePath</i>	Meta-path to the object schema column. The feedback principle is applied when building the column path similar to EntitySchemaQuery . The path is generated in relation to the schema, referred to by the model column.
<i>value</i>	Filtration value. This property is described if ATTRIBUTE value type (<i>type</i>) is indicated.

See also

- **The FILTRATION rule use case**
- **The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to make a field required based on a specific condition**
- **Business rules created via wizards**

The FILTRATION rule use case

Beginner

Easy

Medium

Advanced

Introduction

The *FILTRATION* rule is used to configure filtering of the lookup column of the view model based on the value of another column. For more information on business rules, see the “[Setting the edit page fields using business rules](#)” article.

In Creatio, you can configure business rules using developer tools the as well as the section wizard. For more information please refer to the “[Setting up the business rules](#)” article.

Case description

Add the [Country], [State/Province] and [City] fields to the page. If the [Country] field is populated, the values in the [State/Province] field must include only states and provinces of that country. If the [State/Province] field is populated, the values in the [City] field must include only cities located in that state or province. If the [City] field is populated first, the [Country] and [State/Province] fields must be automatically populated with the corresponding values.

The base contact page schema already has a rule for filtering cities by country. Therefore, if the [Country] field is not added, only cities from the country specified for a contact can be selected.

Source code

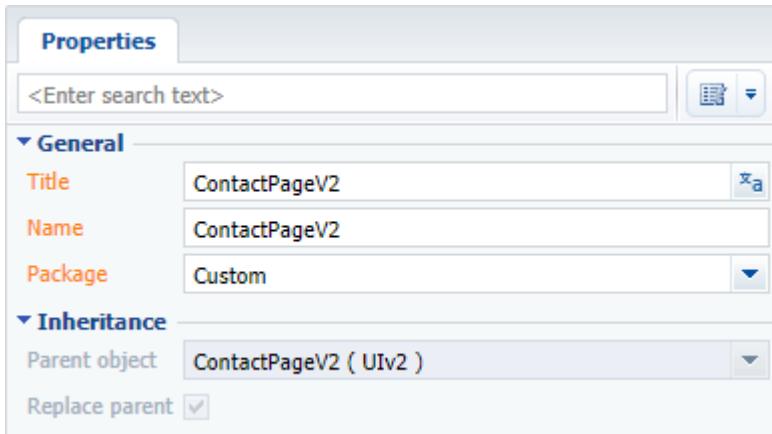
Use this [link](#) to download the case implementation package.

Case implementation algorithm

1. Create a replacing contact page

Create a replacing client module and specify the [Display schema — Contact card] schema as parent object (Fig. 1). The procedure for creating a replacing page is covered in the “[Creating a custom client module schema](#)” article.

Fig. 1. Order edit page replacing schema properties



2. Add the [Country], [State/Province] and [City] fields to the page.

To do this, add three configuration objects with the settings for the corresponding field properties to the *diff* array.

3. Add FILTRATION-type rules to the [City] and [State/Province] columns.

To do this, add two rules of the *BusinessRuleModule.enums.RuleType.FILTRATION* type to the *rules* property for the [City] and [Region] columns. To enable reverse filtering (i.e., to automatically populate the [Country] and [State/Province] fields based on the selected city), set the *autocomplete* property to *true*.

The replacing schema source code is as follows:

```
// Add the module BusinessRuleModul to the dependency list of the module.
define("ContactPageV2", ["BusinessRuleModule"],
  function(BusinessRuleModule) {
    return {
      // Name of the schema of the edit page object.
      entitySchemaName: "Contact",
      // A property that contains a collection of business rules for the schema
      // of the page view model.
      rules: {
        // A set of rules for the [City] column of the view model..
        "City": {
          // The rule for filtering the [City] column by the value of the
          // [Region] column.
          "FiltrationCityByRegion": {
            // FILTRATION rule type.
            "ruleType": BusinessRuleModule.enums.RuleType.FILTRATION,
            // Reverse filtering will be performed.
            "autocomplete": true,
            // The value will be cleared when the value of the [Region]
            // column changes.
            "autoClean": true,
            // The path to the column for filtering in the [City]
            // reference schema,
            // which is referenced by the [City] column of the
            // edit page view model.
            "baseAttributePatch": "Region",
            // The type of the comparison operation in the filter.
            "comparisonType": Terrasoft.ComparisonType.EQUAL,
            // The column (attribute) of the view model will be the
            // comparison value.
            "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
            // The column name of the view model of the edit page,
            // the value of which will be filtered.
          }
        }
      }
    }
  }
)
```

```
        "attribute": "Region"
    }
},
// A set of rules for the [Region] column of the view model.
"Region": {
    "FiltrationRegionByCountry": {
        "ruleType": BusinessRuleModule.enums.RuleType.FILTRATION,
        "autocomplete": true,
        "autoClean": true,
        "baseAttributePatch": "Country",
        "comparisonType": Terrasoft.ComparisonType.EQUAL,
        "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
        "attribute": "Country"
    }
}
},
// Setting up the visualization of the [Country], [State/Province] and
[City] fields on the edit page.
diff: [
    // Metadata for adding the [Country] field.
    {
        "operation": "insert",
        "parentName": "ProfileContainer",
        "propertyName": "items",
        "name": "Country",
        "values": {
            "contentType": Terrasoft.ContentType.LOOKUP,
            "layout": {
                "column": 0,
                "row": 6,
                "colSpan": 24
            }
        }
    },
    // Metadata for adding the [State/Province] field.
    {
        "operation": "insert",
        "parentName": "ProfileContainer",
        "propertyName": "items",
        "name": "Region",
        "values": {
            "contentType": Terrasoft.ContentType.LOOKUP,
            "layout": {
                "column": 0,
                "row": 7,
                "colSpan": 24
            }
        }
    },
    // Metadata for adding the [City] field.
    {
        "operation": "insert",
        "parentName": "ProfileContainer",
        "propertyName": "items",
        "name": "City",
        "values": {
            "contentType": Terrasoft.ContentType.LOOKUP,
            "layout": {
                "column": 0,
                "row": 8,
                "colSpan": 24
            }
        }
    }
]
```

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

4. Save the created replacing page schema

After saving the schema and updating the application page, three new fields will be added to the contact profile (Fig. 2). Their values will be filtered based on the values entered in any of these fields. The filtering also works in the lookup selection window (Fig. 4).

Fig. 2. New fields in the contact profile



Full name*

Anna Baker

Full job title

Specialist

Mobile phone

+1 617 221 5187

Business phone

+1 617 440 2031

Email

a.baker@ac.com

Country

United States



State/province

Massachusetts

City

Boston

Fig. 3. Filtering

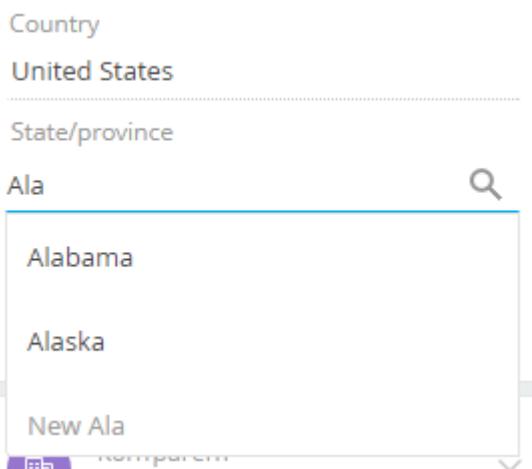


Fig. 4. Filtered values in the lookup selection window

Select: States/provinces X

SELECT CANCEL NEW ACTIONS ▾ VIEW ▾

Name ▼ **SEARCH**

Name ▲

- Alabama
- Alaska
- Arizona
- Arkansas
- California
- Colorado
- Connecticut
- Delaware
- District of Columbia
- Florida
- Georgia
- Hawaii
- Idaho
- Illinois
- Indiana
- Iowa
- Kansas
- Kentucky
- Louisiana
- Maine
- Maryland
- Massachusetts
- Michigan
- Minnesota
- Mississippi
- Missouri
- Montana
- Nebraska
- Nevada
- New Hampshire
- New Jersey
- New Mexico
- New York
- Pennsylvania
- Rhode Island
- South Carolina
- Tennessee
- Texas
- Utah
- Vermont
- Virginia
- Washington
- West Virginia
- Wisconsin
- Wyoming

See also

- **Setting the edit page fields using business rules**
- **The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to make a field required based on a specific condition**

The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition

Beginner

Easy

Medium

Advanced

Introduction

BINDPARAMETER rule is used for resolving the following tasks:

- hide and display fields
- lock and unlock fields
- make fields required or optional

For more information on business rules, see the “[Setting the edit page fields using business rules](#)” article.

In Creatio, you can configure business rules using developer tools as well as the section wizard. For more information please refer to the “[Setting up the business rules](#)”.

Case description

Add a new [Meeting place] field to the activity page. The field will be available only for activities of the [Meeting] type.

You can add fields to the edit page manually or via the section wizard.

For more on adding fields to edit pages see the “[Adding a new field to the edit page](#)” article.

Source code

Use this [link](#) to download the case implementation package.

Case implementation algorithm

1. Create a replacing object and add a new column to it.

Create an [Activity] replacing object and add a new [Meeting place] column of the “string” type to it (Fig. 1). Learn more about creating a replacing object schema in the “[Creating the entity schema](#)” article.

Fig. 1. Adding a custom column to the replacing object

The screenshot shows the 'Properties' tab of the Creatio application. On the left, the 'Structure' panel displays a tree view with 'UsrMeetingPlace' selected under 'Activity'. The 'Properties' panel on the right shows the following configuration:

- General** section:

Title	Meeting place
Name	UsrMeetingPlace
Data type	Text (250 characters)
- String** section: Multi-line text (unchecked), Localizable text (unchecked).
- Lookup** section: Lookup (dropdown), Cascade connection (unchecked), Do not control integrity (unchecked), List (unchecked).
- Behavior** section: Required (No), Default value (No), Make copy (checked), Indexed (unchecked), Update change log (unchecked), Usage mode (General).

2. Create a replacing client module for the activity page

Create a replacing client module and specify the [Activity edit page] schema as parent object (Fig. 2). The procedure for creating a replacing page is covered in the “[Creating a custom client module schema](#)” article.

Fig. 2. Replacing edit page properties

The screenshot shows the 'Properties' tab for a client module. The 'General' section contains the following information:

Title	Activity edit page
Name	ActivityPageV2
Package	Custom

The 'Inheritance' section contains:

Parent object	Activity edit page (UIv2)
Replace parent	<input checked="" type="checkbox"/>

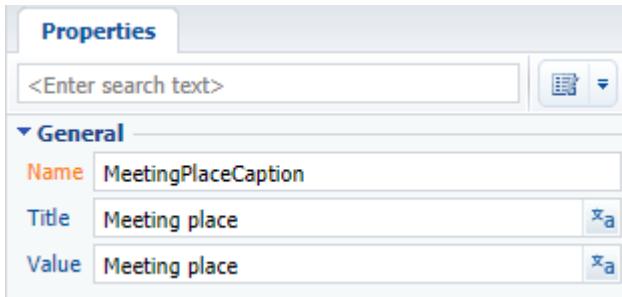
3. Add a new field to the activity edit page.

Add a configuration object with the [Meeting place] field properties on the page to the diff array. The process of

adding fields to pages is covered in the “**Adding a new field to the edit page**” article.

To enable localization of this field, add a localizable string (Fig. 3) and bind it to the field title.

Fig. 3. Localizable string properties



4. Add a rule to the “rules” property of the page view model

For the *UsrMeetingPlace* column, add the rule with the BINDPARAMETER type to *rules* property of the page view model. Set the *BusinessRuleModule.enums.Property.VISIBLE* value for the rule’s *property*. Add the following condition for rule execution to the *conditions* array: the value in the *ActivityCategory* column of the model should be equal to the *ConfigurationConstants.Activity.ActivityCategory.Meeting* configuration constant.

The *ConfigurationConstants.Activity.ActivityCategory.Meeting* configurational constant contain the id of the “Meeting” record of the [Activity category] lookup.

The replacing schema source code is as follows:

```
// Add the module BusinessRuleModule and ConfigurationConstants to the dependency
list of the module.
define("ActivityPageV2", ["BusinessRuleModule", "ConfigurationConstants"],
  function(BusinessRuleModule, ConfigurationConstants) {
    return {
      // Name of the page schema of the edit page.
      entitySchemaName: "Activity",
      // Displaying a new field on the edit page.
      diff: /**SCHEMA_DIFF*/[
        // Metadata for adding a field [Meeting place].
        {
          // The operation of adding a component to a page.
          "operation": "insert",
          // The meta name of the parent container to which the field is
          // added.
          "parentName": "Header",
          // The field is added to the parent
          // component's collection.
          "propertyName": "items",
          // The name of the column of the schema to which the component is
          // bound.
          "name": "UsrMeetingPlace",
          "values": {
            // Field title.
            "caption": {"bindTo":
              "Resources.Strings.MeetingPlaceCaption"},
            // Location of the field.
            "layout": { "column": 0, "row": 5, "colSpan": 12 }
          }
        }
      ]/**SCHEMA_DIFF*/,
      // Rules of the edit page view model.
      rules: {
        // A set of rules for the [Meeting place] column of the view model.
        "UsrMeetingPlace": {

```

```

        // The dependence of visibility of the [Meeting Place] field from
the value in [Category] field.
        "BindParametrVisiblePlaceByType": {
            // The type of the BINDPARAMETER rule.
            "ruleType": BusinessRuleModule.enums.RuleType.BINDPARAMETER,
            // Rule regulates the VISIBLE property.
            "property": BusinessRuleModule.enums.Property.VISIBLE,
            // An array of conditions in which the rule is triggered.
            // Determines whether the value in the [Category] column is
equal to the value "Meeting".
            "conditions": [
                // Expression of the left side of the condition.
                "leftExpression": {
                    //The type of the expression is the attribute
(column) of the view model.
                    "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                    // Name of the view model column which value is
compared in the expression.
                    "attribute": "ActivityCategory"
                },
                // The type of comparison operation.
                "comparisonType": Terrasoft.ComparisonType.EQUAL,
                // Expression of the right side of the condition.
                "rightExpression": {
                    // Type of expression is a constant value.
                    "type": BusinessRuleModule.enums.ValueType.CONSTANT,
                    // The value with which the left side expression is
compared.
                    "value": ConfigurationConstants.Activity.ActivityCategory.Meeting
                }
            ]
        }
    }
};

});
```

After saving the schema and refreshing the application page, an additional [Meeting place] field will appear on the activity page if the activity category is “Meeting” (Fig. 5, 5).

Fig. 4. Case result. Activity type is “To do”, the [Meeting place] field is not visible

Subject *	Visit bpm'online Academy				
Start *	4/1/2017	3:15 PM	Owner *	Supervisor	
Due *	4/1/2017	3:45 PM	Reporter *	Supervisor	
Status *	Not started			Priority *	Medium
Show in calendar	<input checked="" type="checkbox"/>			Category *	To do

Fig. 5. Case result. Activity type is “To do”, the [Meeting place] field is visible

Subject *	Visit bpm'online Academy				
Start *	4/1/2017	3:15 PM	Owner *	Supervisor	
Start					
Due *	4/1/2017	3:45 PM	Reporter *	Supervisor	
Status *	Not started			Priority *	Medium
Show in calendar	<input checked="" type="checkbox"/>			Category *	Meeting
Meeting place	office				

See also

- **Setting the edit page fields using business rules**
- **The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to make a field required based on a specific condition**
- **The FILTRATION rule use case**
- **Business rules created via wizards**

The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition

Beginner

Easy

Medium

Advanced

Introduction

BINDPARAMETER rule is used for resolving the following tasks:

- hide and display fields
- lock and unlock fields
- make fields required or optional

For more information on business rules, see the “[Setting the edit page fields using business rules](#)” article.

In Creatio, you can configure business rules using developer tools as well as the section wizard. For more information please refer to the “[Setting up the business rules](#)”.

Case description

Configure fields on the contact edit page to make the [Business phone] field editable only if the [Mobile phone] field is filled.

Source code

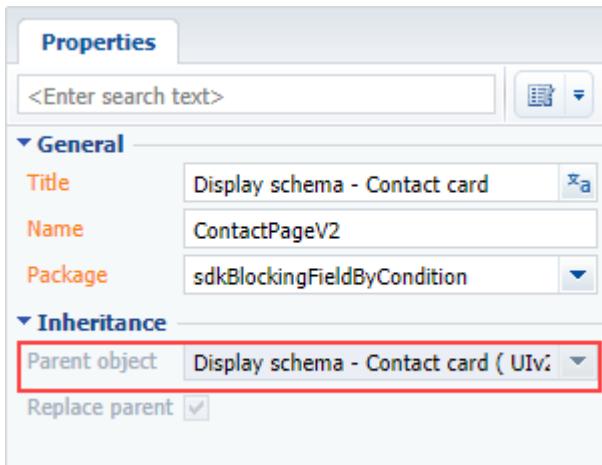
Use this [link](#) to download the case implementation package.

Case implementation algorithm

1. Create a replacing client module for the contact page

Create a replacing client module and specify the [Display schema – Contact card] schema as parent object (Fig. 1). The procedure for creating a replacing page is covered in the “[Creating a custom client module schema](#)” article.

Fig. 1. Order edit page replacing schema properties



2. In the rules property of the page view model, add the rule

For the *Phone* column, add the rule with the BINDPARAMETER type to *rules* property of the page view model. Set the BusinessRuleModule.enums.Property.ENABLED. Value for the rule's *property*. Add the following condition for rule execution to the *conditions* array: the value in the *MobilePhone* column should be filled.

The replacing schema source code is as follows:

```
// Add the module BusinessRuleModule to the list of dependent modules.
define("ContactPageV2", ["BusinessRuleModule"], function(BusinessRuleModule) {
    return {
        // Name of the page schema of the edit page.
        entitySchemaName: "Contact",
        // Rules of the edit page view model.
        rules: {
            // A set of rules for the [Business phone] column of the view model.
            "Phone": {
                // Dependence of the availability of the [Business phone] field from
                // the value of the [Mobile phone] field.
                "BindParameterEnabledPhoneByMobile": {
                    // The type of the BINDPARAMETER rule.
                    "ruleType": BusinessRuleModule.enums.RuleType.BINDPARAMETER,
                    // The rule regulates the ENABLED property.
                    "property": BusinessRuleModule.enums.Property.ENABLED,
                    // An array of conditions in which the rule is triggered.
                    // Determines whether the [Mobile Phone] field is populated.
                    "conditions": [
                        // Expression of the left side of the condition.
                        "leftExpression": {
                            // The type of the expression is the attribute (column)
                            // of the view model.
                            "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                            // The name of the column in the view model, whose value
                            // is compared in the expression.
                            "attribute": "MobilePhone"
                        },
                        // The type of comparison operation is "not equal to".
                        "comparisonType": Terrasoft.ComparisonType.NOT_EQUAL,
                        // Expression of the right side of the condition.
                        "rightExpression": {
                            // The expression type is a constant value.
                            "type": BusinessRuleModule.enums.ValueType.CONSTANT,
                            // The value with which the left side expression is
                            // compared.
                            "value": ""
                        }
                    ]
                }
            }
        }
    }
})
```

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

After saving the schema and refreshing the application page, the [Business phone] field will be non-editable until the [Mobile phone] field is empty (Fig. 2).

Fig. 2. Example result demonstration

100%

9:21 PM
-1d,
Atlanta

Full name*

Clayton Bruce

Full job title

Purchase Manager

Mobile phone

Business phone

Email

clayton@axiom-corp.com

See also

- [Setting the edit page fields using business rules](#)
- [The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition](#)
- [The BINDPARAMETER rule. How to make a field required based on a specific condition](#)
- [The FILTRATION rule use case](#)
- [Business rules created via wizards](#)

The BINDPARAMETER rule. How to make a field required based on a specific condition

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

Introduction

BINDPARAMETER rule is used for resolving the following tasks:

- hide and display fields

- lock and unlock fields
- make fields required or optional

For more information on business rules, see the “[Setting the edit page fields using business rules](#)” article.

In Creatio, you can configure business rules using developer tools as well as the section wizard. For more information please refer to the “[Setting up business rules](#)” article.

Case description

Set up the contact edit page fields so that the [Business phone] field is required on condition that the [Contact type] field is populated with the “Customer” value.

Source code

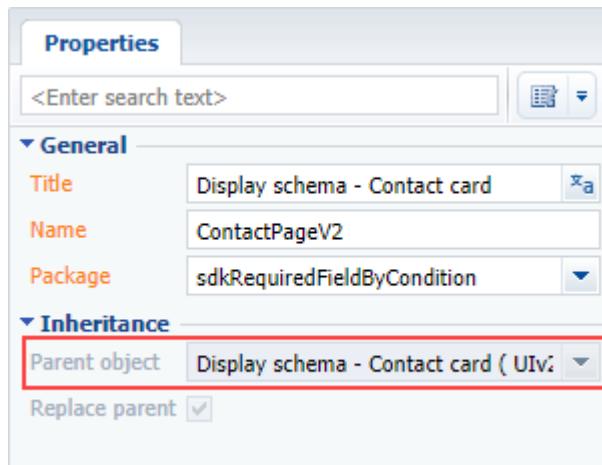
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Create a replacing client module for the contact edit page.

Create a replacing client module and specify the [Display schema — Contact card] schema as parent object (Fig. 1). The procedure for creating a replacing page is covered in the “[Creating a custom client module schema](#)” article.

Fig. 1. Replacing edit page properties



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

Add the BINDPARAMETER type rule for the *Phone* column to the *rules* property of the page view model. Set the *property* rule value to BusinessRuleModule.enums.Property.REQUIRED. Add a rule execution condition to the *conditions* array – the *Type* column value of the model should be equal to the ConfigurationConstants.ContactType.Client configuration constant.

The ConfigurationConstants.ContactType.Client configuration constant contains the “Client” record identifier of the [Contact type] lookup.

The replacing schema source code is as follows:

```
// Add the BusinessRuleModule and ConfigurationConstants modules to the module dependency list.
define("ContactPageV2", ["BusinessRuleModule", "ConfigurationConstants"],
    function(BusinessRuleModule, ConfigurationConstants) {
        return {
            // Name of the edit page object schema.
            entitySchemaName: "Contact",
            // Rules of the edit page view model.
            rules: [
                // Set of rules of the [Business rule] view model column.
```

```
"Phone": {
    // Dependency of the [Business phone] field "required" property
on the [Type] field value.
    "BindParameterRequiredAccountByType": {
        // BINDPARAMETER rule type.
        "ruleType": BusinessRuleModule.enums.RuleType.BINDPARAMETER,
        // The rule regulates the REQUIRED property.
        "property": BusinessRuleModule.enums.Property.REQUIRED,
        // Condition array, whose performance triggers the rule
execution.
        // Defines if the [Type] column value is equal to the
"Client" value.
        "conditions": [
            // Expression of the left side of the condition.
            "leftExpression": {
                // Expression type - view model attribute(column).
                "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                // Name of the view model column whose value is
compared in the expression.
                "attribute": "Type"
            },
            // Comparison operation type.
            "comparisonType": Terasoft.ComparisonType.EQUAL,
            // Expression of the right side of the condition.
            "rightExpression": {
                // Expression type - constant value.
                "type": BusinessRuleModule.enums.ValueType.CONSTANT,
                // The comparison value for the left side of the
expression.
                "value": ConfigurationConstants.ContactType.Client
            }
        ]
    }
},
};

});
```

After you save the schema and update the application web page, the [Business phone] filed of the contact edit page will be required on condition the contact type is the “Customer”.

Fig. 2. Case result The [Business phone] field – optional

The screenshot shows a contact record for Clayton Bruce. The left panel displays basic contact information: Full name (Clayton Bruce), Full job title (Purchase Manager), Mobile phone (+1 404 389 0476), Business phone (+1 404 532 3976), and Email (clayton@axiom-corp.com). The Business phone field is highlighted with a red border. The right panel shows the 'NEXT STEPS' section with a task to 'Prepare quotation' scheduled for 1/11/2018 by John Best. Below this, the contact type is listed as 'Contact person' (Type: Mr.). Communication options include mobile and business phones, each with a green call icon.

Fig. 3. Case result The [Business phone] field – required

This screenshot shows the same contact record for Clayton Bruce, but the Business phone field is now empty and not highlighted. The contact type is now listed as 'Customer' (Type: Mr.). The communication options remain the same, with mobile and business phones each having a green call icon.

See also

- Setting the edit page fields using business rules
- The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition
- The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition
- The FILTRATION rule use case
- Business rules created via wizards

Business rules created via wizards

Beginner

Easy

Medium

Advanced

Introduction

Starting from version 7.10.0 besides the **business rules created by developer tools**, there exist the [business rules generated by section or detail wizards](#).

- Wizard generated business rules are added to the *businessRules* client module property.
- The generated business rules have higher priority at execution.
- The *BusinessRuleModule* enumerations are not used when describing the generated business rules.

See the "[Setting up business rules](#)" article for more information on business rule setup via wizards. The manual setup of the generated business rules is covered in the "**Business rules. The *businessRules* property**" article.

Additional properties

You can find additional properties of the wizard generated business rules in table 1.

Table 1. Additional properties

Property	Details
<i>uId</i> .	Unique rule identifier. The "GUID" type value.
<i>enabled</i>	Enabling checkbox. Can take the <i>true</i> or <i>false</i> values.
<i>removed</i>	The checkbox indicating if the rule is removed. Can take the <i>true</i> or <i>false</i> values.
<i>invalid</i>	The checkbox indicating if the rule is valid. Can take the <i>true</i> or <i>false</i> values.

Cases

Case of a master generated business rule connected with a field property (whether it is visible, enabled or required):

```
define("SomePage", [], function() {
    return {
        // ...
        businessRules: /**SCHEMA_BUSINESS_RULES*/{
            // Set of rules for the [Type] column of the view model.
            "Type": {
                // Wizard generated rule code.
                "ca246daa-6634-4416-ae8b-2c24ea61d1f0": {
                    // Unique rule identifier.
                    "uId": "ca246daa-6634-4416-ae8b-2c24ea61d1f0",
                    // Enabling checkbox.
                    "enabled": true,
                    // Checkbox indicating if the rule is removed.
                    "removed": false,
                    // Checkbox indicating if the rule is valid.
                    "invalid": false,
                    // Rule type.
                    "ruleType": 0,
                }
            }
        }
    }
})
```

```

    // The property code, regulating the rule.
    "property": 0,
    // Logical connection between several rule conditions.
    "logical": 0,
    // Condition array, whose performance triggers the rule implementation.
    // Compares the [Account.PrimaryContact.Type] value with the [Type]
column value.
    "conditions": [
        {
            // Comparison operation type.
            "comparisonType": 3,
            // Expression of the left side of the condition.
            "leftExpression": {
                // Expression type – the view model column (attribute).
                "type": 1,
                // The view model column name.
                "attribute": "Account",
                // The path to the [Account] lookup schema, whose value
                // is compared in the expression.
                "attributePath": "PrimaryContact.Type"
            },
            // Expression of the right side of the condition.
            "rightExpression": {
                // Expression type – the view model column (attribute).
                "type": 1,
                // The view model column name.
                "attribute": "Type"
            }
        }
    ]
}
}/**SCHEMA_BUSINESS_RULES*/
// ..
);
});

```

Case of a master generated business rule for field filtration:

```

define("SomePage", [], function() {
    return {
        // ...
        businessRules: /**SCHEMA_BUSINESS_RULES*/{
            // Set of rules for the [Type] column of the view model.
            "Account": {
                // Master generated rule code.
                "a78b898c-c999-437f-9102-34c85779340d": {
                    // Unique rule identifier.
                    "uId": "a78b898c-c999-437f-9102-34c85779340d",
                    // Enabling checkbox.
                    "enabled": true,
                    // Checkbox indicating if the rule is removed.
                    "removed": false,
                    // Checkbox indicating if the rule is valid.
                    "invalid": false,
                    // Rule type.
                    "ruleType": 1,
                    // Path to the filtration column of the [Account] lookup schema,
                    // that the [Type] column of the edit page view model
                    // refers to.
                    "baseAttributePatch": "PrimaryContact.Type",
                    // Filter comparison operation type.
                }
            }
        }
    }
});

```

```

        "comparisonType": 3,
        // Expression type – the view model column (attribute).
        "type": 1,
        // The view model column name,
        // whose value will be used for filtration.
        "attribute": "Type"
    }
}
} /**SCHEMA_BUSINESS_RULES*/
// ..
};

}) ;
}
);

```

See also

- **Setting the edit page fields using business rules**
- **The BINDPARAMETER rule. How to lock a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to hide a field on an edit page based on a specific condition**
- **The BINDPARAMETER rule. How to make a field required based on a specific condition**
- **The FILTRATION rule use case**

Adding an action to the edit page

Beginner

Easy

Medium

Advanced

Introduction

Creatio has the possibility to set up a list of actions from the standard [Actions] menu on the edit page.

The list of page actions is an instance of the *Terrasoft.BaseViewModelCollection* class. Each item of the actions list is a view model.

An action is set up in the configuration object where both properties of the actions view model may be set explicitly and the base binding mechanism may be used.

The base content of the [Actions] menu for the edit page is implemented in the base class of the *BasePageV2* pages. The list of section actions returns the *getActions()* protected virtual method from the *BasePageV2* schema.

A separate action is added to the collection by calling the *addItem()* method. The *getButtonMenuItem()* callback method is passed to it as a parameter. The method creates an instance of the actions view model by the configuration object passed to it as the parameter.

Base implementation of addig the action

```

/**
 * Returns the collection of edit page actions
 * @protected
 * @virtual
 * @return {Terrasoft.BaseViewModelCollection} Returns the collection of page actions
 */
getActions: function() {
    // List of actions - Terrasoft.BaseViewModelCollection instance
    var actionMenuItems = this.Ext.create("Terrasoft.BaseViewModelCollection");
    // Adding an action to the collection. The method instantiating the action model
    // instance by the passed configuration object is passed as callback.
    actionMenuItems.addItem(this.getButtonMenuItem({
        // Configuration object for action setting.
        ...
    }));
}
);

```

```
// Returns a new collection of actions.
return actionMenuItems;
}
```

Below are the properties of the configuration object of the section action to be passed as a parameter to the `getButtonMenuItem()` method:

Table 1. Property of the configuration object

Property	Details
<code>Type</code>	a type of the [Actions] menu item A horizontal line for separating the menu blocks may be added to the action menu using this property. For this purpose, the <code>Terrasoft.MenuSeparator</code> string must be specified as the property value. If no property value is specified, the menu item will be added by default.
<code>Caption</code>	the title of the [Actions] menu item. To set titles, the use of localizable schema strings is recommended.
<code>Tag</code>	the name of the action handler method is set in this property
<code>Enabled</code>	a logical property controlling the menu item availability
<code>Visible</code>	a logical property controlling the menu item visibility

Procedure for adding a custom action

1. Create replacing schema of existing page or a new page.
2. Override the `getActions()` method.
3. Add an action to the actions collection using the `addItem()` method.
4. Pass a configuration object with the added action settings to the `getButtonMenuItem()` callback method.

When base sections are replaced in the `getActions()` method of the replacing module, the parent implementation of this method must be called first to initialize actions of the parent section. For this, execute the `this.callParent(arguments)` method that returns collection of base page actions.

Case description

The [Show execution date] which will display the scheduled order execution date in the data window must be added to the edit page. The action will be available only for orders at the [In progress] stage.

The [Orders] section is available in Sales Creatio products.

The edit page action is used to edit a specific object opened on the page. To have access to values of the edit page object fields in the action handler method, the following view model methods must be used: `get()` – to receive a value and `set()` - to set a value.

Source code

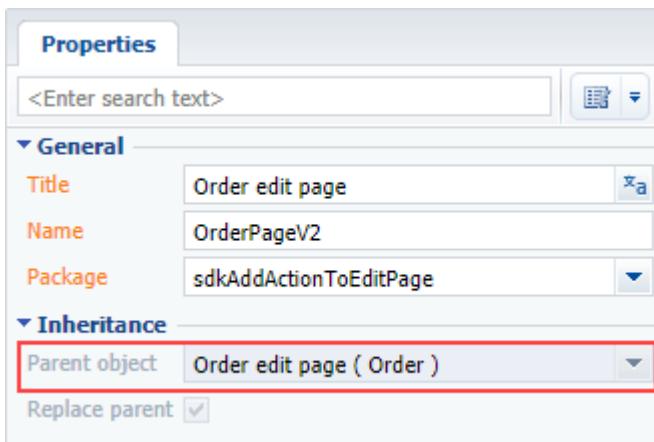
Use this [link](#) to download the case implementation package.

Case implementation algorithm

1. Create a replacing edit page for an order in a custom package

A replacing client module must be created and [Order edit page] (`OrderPageV2`) must be specified as the parent object in it (Fig. 1). The procedure of creating a replacing page is covered in the “**Creating a custom client module schema**” article.

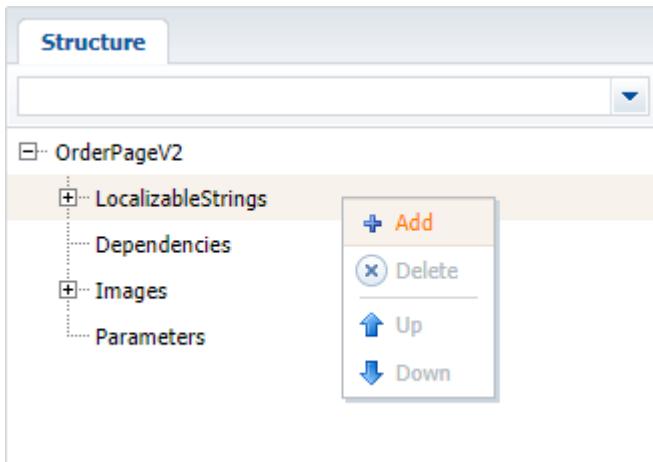
Fig. 1. Properties of the replacing edit page



2. Add a string with the [Actions] menu title to the localized string collection of the page replacing schema

Create a new localizable string (Fig. 2).

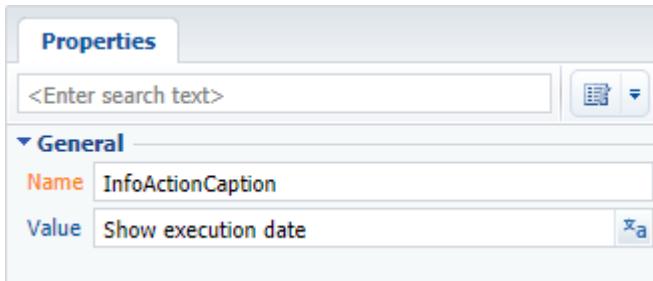
Fig. 2 – Adding the localized string to the schema



Populate the following values for the created string (Fig.3):

- [Name] – “InfoActionCaption”.
- [Value] – “Show execution date”.

Fig. 3. Properties of the custom localizable string



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

- *isRunning()* – checks whether the order is at the [In progress] stage and defines availability of the added menu item.
- *showOrderInfo()* – the action handler method that displays the scheduled end date of the order in the message window.

- `getActions()` – an overridden parent schema method that gets the page action collection.

The replacing schema source code is as follows:

```
define("OrderPageV2", ["OrderConfigurationConstants"],
function(OrderConfigurationConstants) {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Order",
        // Methods of the edit page view model.
        methods: {
            // Method which checks the stage of the order execution to define
            availability of menu item.
            isRunning: function() {
                // The method returns true if the order status is [In progress],
                otherwise it returns false.
                if (this.get("Status")) {
                    return this.get("Status").value ===
OrderConfigurationConstants.Order.OrderStatus.Running;
                }
                return false;
            },
            // Action handler method which displays the order end date in the data
            window.
            showOrderInfo: function() {
                // Receiving the order end date.
                var dueDate = this.get("DueDate");
                // Calling the standard system method for the data window display.
                this.showInformationDialog(dueDate);
            },
            // Override the base virtual method which returns the actions collection
            of the edit page.
            getActions: function() {
                // The parent implementation of the method is called to receive
                // the initiated actions collection of the base page.
                var actionMenuItems = this.callParent(arguments);
                // Adding the separator line.
                actionMenuItems.addItem(this.getButtonItem({
                    Type: "Terrasoft.MenuSeparator",
                    Caption: ""
                }));
                // Adding a menu item to the actions list of the edit page.
                actionMenuItems.addItem(this.getButtonItem({
                    // Binding the title of the menu item to the localizable string
                    of the schema.
                    "Caption": {bindTo: "Resources.Strings.InfoActionCaption"},
                    // Binding the action handler method.
                    "Tag": "showOrderInfo",
                    // Binding the visibility property of the menu item to the value
                    returning the isRunning() method.
                    "Enabled": {bindTo: "isRunning"}
                }));
                return actionMenuItems;
            }
        };
    };
});
```

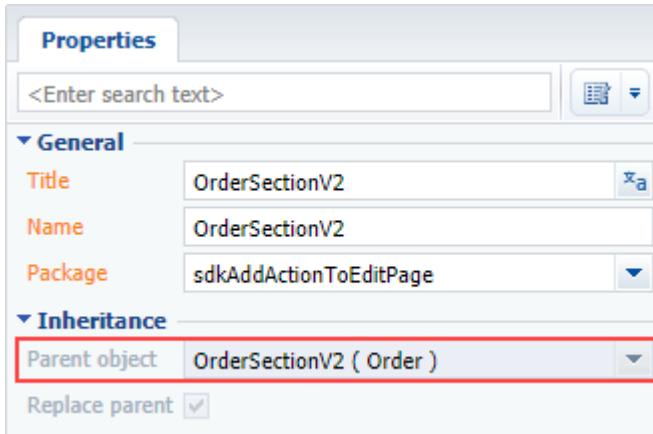
The previous steps are enough to add an action to the edit page. But the custom page action will not be displayed in the vertical view of the list.

For correct displaying of the page action, add the localizable string of the title and a method that defines menu item availability to the section view model schema.

4. Create a replacing schema of the [Orders] section

Create a replacing client module and specify the *OrderSectionV2* schema as parent object (Fig. 4). The procedure for creating a replacing page is described in the “**Creating a custom client module schema**” article.

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



5. Add a localizable string with the [Actions] menu item title

For this purpose, repeat the actions in par. 2

6. Add the method implementation

Add the implementation of the *isRunning()* method to the view model collection of methods. The method will check whether the selected order is at the [In progress] stage and defines availability of the added menu item.

The replacing schema source code is as follows:

```
define("OrderSectionV2", ["OrderConfigurationConstants"],
function(OrderConfigurationConstants) {
    return {
        // Name of the section schema.
        entitySchemaName: "Order",
        // Methods collection of the section view model.
        methods: {
            // Method checking the stage of the selected order to determine the menu
            // item visibility.
            isRunning: function(activeRowId) {
                activeRowId = this.get("ActiveRow");
                // Receiving the data collection of the section list view.
                var gridData = this.get("GridData");
                // Receiving the model of the selected order by a value in the
                primary column.
                var selectedOrder = gridData.get(activeRowId);
                // Receiving the model property - selected order status.
                var selectedOrderStatus = selectedOrder.get("Status");
                // Value of the selected order status is compared to the value of the
                [In-progress] type and
                // returns true or false depending on the comparison result.
                return selectedOrderStatus.value ===
OrderConfigurationConstants.Order.OrderStatus.Running;
            }
        };
    });
});
```

After you save the schema and update the application page with clearing the cache, a new action will appear on the

order edit page when selecting an order at the [In progress] stage (Fig. 5,6).

Fig. 5. Case result. The order is at the [Completed] stage and the action is inactive

The screenshot shows the 'Orders' module in the Creatio application. On the left, there's a sidebar with filters like 'Start date' and 'Owner'. The main area displays two orders: 'ORD-15' (Status: In progress) and 'ORD-14' (Status: Completed). Order 'ORD-14' is selected. The 'Actions' dropdown for this order is open, showing options like 'Set up access rights', 'Follow the feed', 'Send for approval', 'New invoice based on this order', 'New contract based on order', and 'Show execution date'. The 'Show execution date' option is highlighted with a red box. To the right, the order details for 'ORD-14' are shown, including a summary table with total amounts (\$11,725.00) and payment information (\$11,725.00), and a detailed table of items with prices, quantities, and discounts.

Fig. 6. Case result. The order is at the [In progress] stage and the action is active

This screenshot is similar to Fig. 5 but shows order 'ORD-15' instead of 'ORD-14'. The status of 'ORD-15' is listed as '3. In progress'. The 'Actions' dropdown for this order is open, and the 'Show execution date' option is highlighted with a red box. The rest of the interface is identical to Fig. 5, showing the order details and summary tables.

Control elements

Contents

- **[Adding a new field to the edit page](#)**
- **[Adding a button to the edit page](#)**
- **[How to add a field with an image to the edit page](#)**
- **[How to add the color select button to the edit page](#)**
- **[How to add multi-currency field](#)**
- **[How to add custom logic to the existing controls](#)**

Adding a new field to the edit page

Beginner

Easy

Medium

Advanced

Introduction

You can add fields to the edit page in two ways:

1. Via section wizard (see the ["Section wizard"](#), ["How to set up page fields"](#) articles).

A base object (for example, the [Activity] object) replacing schema and a base edit page (for example, the *ActivityPageV2*) replacing schema will be created in the custom package as a result of the section wizard operation. A new column will be added in the object replacing schema. A configuration object with the settings of the new field location on a page will be added to the *diff* array in the edit page replacing schema.

When creating new sections via the wizard, it will create new schemas instead of replacing schemas in your current custom package.

You should implement additional edit page business logic or develop new client interface elements in the created edit page replacing schema.

2. Via creating replacing base object and replacing base page by developer tools.

Source code

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

Example 1

Case description

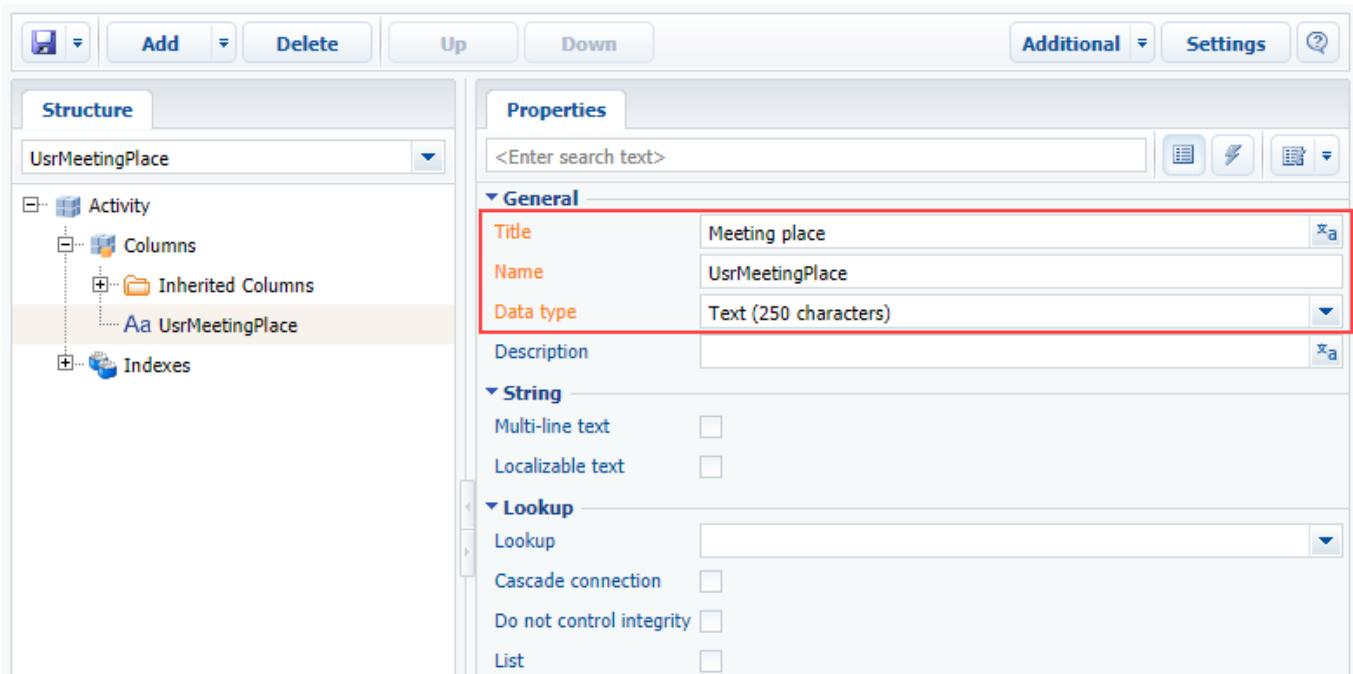
Manually add a new [Meeting place] field to the activity edit page.

Case implementation algorithm

1. Create a replacing object and add a new column to it.

Create an [Activity] replacing object and add the new [Meeting place] column of the “string” type to it (Fig. 1). Learn more about creating a replacing object schema in the [“Creating the entity schema”](#) article.

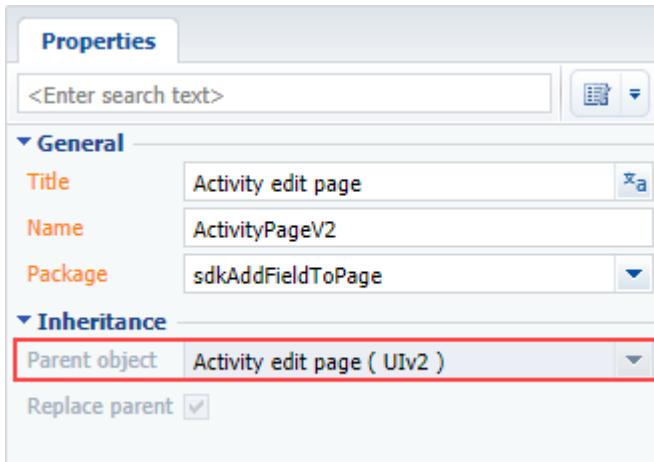
Fig. 1. Adding a custom column to the replacing object



2. Create a replacing client module for the activity page

Create a replacing client module and specify the [Activity edit page] schema as parent object (Fig. 2). The procedure of creating a replacing page is covered in the “**Creating a custom client module schema**” article.

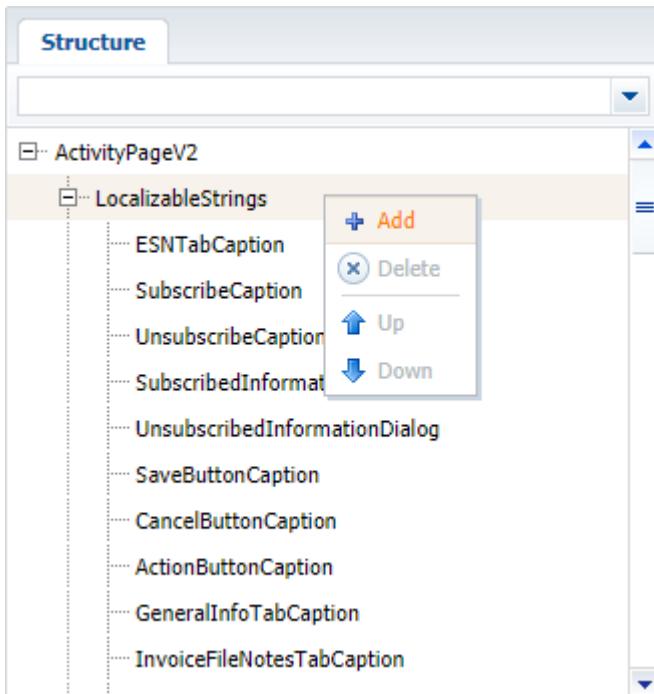
Fig. 2. Replacing edit page properties



3. Add a localized string with the field caption.

Add a string containing the added field caption to the localized string collection of the replacing page schema (fig.3).

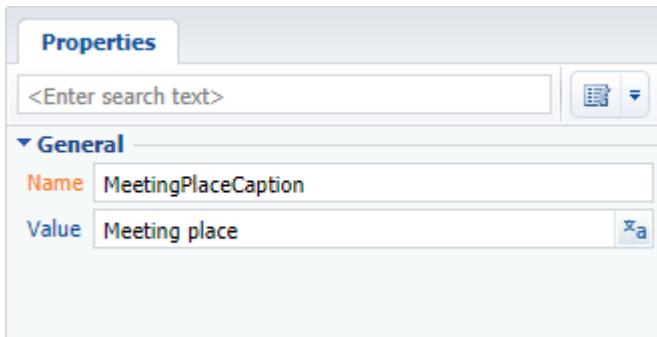
Fig. 3. Adding localized string to the schema



For the created string specify (Fig. 4):

- [Name] – "MeetingPlaceCaption";
- [Value] – “Meeting place”.

Fig. 4. Properties of the custom localized string



4. Add a new field to the activity edit page.

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

More information about the *diff* array properties is available in the "[The "diff" array](#)" article.

The replacing schema source code is as follows:

```
define("ActivityPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Activity",
        // Displaying of a new field on the edit page.
        diff: /**SCHEMA_DIFF*/[
            // Meta data for adding the [Meeting place] field.
            {
                // Operation of adding a component to the page.
                "operation": "insert",
                // Meta name of a parent container where a field is added.
                "parentName": "Header",
                // The field is added to the component collection
                // of a parent element.
                "propertyName": "items",
                // The name of a schema column that the component is linked to.
                "name": "UsrMeetingPlace",
                "values": {
                    // Field caption.
                    "caption": {"bindTo": "Resources.Strings.MeetingPlaceCaption"},
                    // Field location.
                    "layout": {
                        // Column number.
                        "column": 0,
                        // String number.
                        "row": 5,
                        // Span of the occupied columns.
                        "colSpan": 12
                    }
                }
            }
        ] /**SCHEMA_DIFF*/
    };
});
```

After you save the schema and update the application page with clearing the cache, you will see a new field appear on the activity edit page (fig.5).

Fig. 5. Case result demonstration

The screenshot shows a 'Prepare quotation' case in the Creatio application. The top navigation bar includes 'CLOSE', 'ACTIONS', a search bar ('What can I do for you?'), and 'VIEW'. The main area displays case details:

- Subject:** Prepare quotation
- Start:** 1/4/2018, 4:00 PM
- Due:** 1/4/2018, 6:00 PM
- Status:** Completed
- Owner:** Murphy Valerie
- Reporter:** John Best
- Priority:** Medium
- Category:** Paper work
- Show in calendar:**
- Meeting place:** [Text input field]

Example 2

Case description

Manually add a [Country] field to the contact profile edit page. The difference of this case is that you already have the [Country] column in your object schema.

Case implementation algorithm

1. Create a replacing contact page

Create a replacing client module and specify the [Display schema – Contact card], *ContactPageV2* schema as parent object. The procedure of creating a replacing page is covered in the “**Creating a custom client module schema**” article.

2. Add the [Country] field to the contact profile.

Add a configuration object containing the field property settings to the diff array. Indicate the *ProfileContainer*. element as a parent schema element where the field will be located.

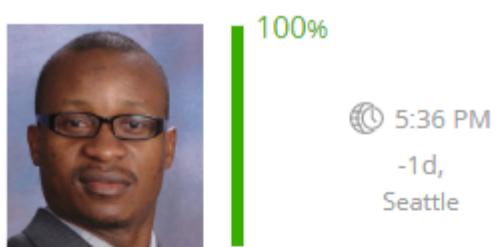
The replacing schema source code is as follows:

```
define("ContactPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Contact",
        diff: [
            // Meta data for adding the [Country] field.
            {
                // Operation of adding a component to the page.
                "operation": "insert",
                // Meta name of a parent container where a field is added.
                "parentName": "ProfileContainer",
                // The field is added to the component collection
                // of a parent element.
                "propertyName": "items",
                // The name of a schema column that the component is linked to.
                "name": "Country",
                "values": {
                    // Value of the [Country] field.
                    "value": "USA"
                }
            }
        ]
    }
});
```

```
// Field type - lookup.  
"contentType": Terrasoft.ContentType.LOOKUP,  
// Field location.  
"layout": {  
    // Column number.  
    "column": 0,  
    // String number.  
    "row": 6,  
    // Span of the occupied columns.  
    "colSpan": 24  
}  
}  
}  
]  
};  
});
```

After you save the schema and update the application page with clearing the cache, you will see a new field appear on the contact edit page (fig.6).

Fig. 6. Case result demonstration



The screenshot shows a contact edit page. At the top right, there is a green progress bar labeled '100%' and a timestamp '5:36 PM' with a globe icon, indicating the record was created '1d, Seattle'. Below this, there are several input fields: 'Full name*' with value 'Barber Andrew'; 'Full job title' with value 'Project Manager'; 'Mobile phone' with value '+1 206 587 1036'; 'Business phone' with value '+1 206 480 3801'; 'Email' with value 'a.barber@gros.com'; and a 'Country' dropdown menu with value 'United States' highlighted. A red rectangular box surrounds the 'Country' dropdown.

Full name*	Barber Andrew
Full job title	Project Manager
Mobile phone	+1 206 587 1036
Business phone	+1 206 480 3801
Email	a.barber@gros.com
Country	United States

Adding a button to the edit page

Contents

- **Introduction**

- **How to add the button on the edit page in the combined mode**
- **How to add a button to an edit page in the new record add mode**

Adding a button to the edit page

Beginner

Easy

Medium

Advanced

To add a custom button to the view model on the edit page, two properties must be modified:

- Methods collection. The implementation of the handler method to be called when the button is clicked must be added to the methods collection. Other auxiliary methods required for the control item functioning must be also added. These may be methods for regulating the visibility or availability of the control item depending on conditions.
- diff configuration objects array. Add a configuration object in the diff configuration objects array to set up the visual location of the control item on the edit page.

To display the button on the page in the existing record edit mode, the section schema must be modified.

To display the button in the new record addition mode, similar modifications are made in the schema of the page view model itself.

DOM model of standard page buttons

The html-container hierarchical structure is used to locate standard functional buttons of the Creatio edit page.

CombinedModeActionButtonsCardContainer is a top level container on the existing record edit page. Two more containers are inside it:

- CombinedModeActionButtonsCardLeftContainer - where the Save, Cancel and Actions standard buttons are located;
- CombinedModeActionButtonsCardRightContainer - where the Print and View buttons are located.

Similarly, for the new record edit page: ActionButtonsContainer — a top level container.

Two more containers are inside it:

- LeftContainer - where the Save, Cancel and standard Actions buttons are located;
- RightContainer - where the Print and View buttons are located.

Depending on the exact position required for a button, the corresponding container is specified when setting the button visualization in the diff array.

Meta-names of html-containers are used here.

These names are specified when setting the control item visualization in the configuration object of the diff array.

The actual ID of corresponding html-items of the page are formed by the system automatically, based on such meta-names.

Setting the button visualization properties

To set the visual location of a custom button on the edit page, a configuration object with the following properties must be added to the *diff* array of the view model:

Property

Description

operation

A type of operation with a control item (*insert*, *move*, *remove*, *merge*, *set*). This is set by a string with the corresponding operation name. The *insert* value is specified to add a new control item.

parentName

The meta-name of the parent control item where a custom item is to be placed. If this is a functional button, *LeftContainer* and *RightContainer* may act as the parent containers.

propertyName

The *items* value is specified for the custom control item.

Property

name

Description

The meta-name of the added control item.

values

A configuration object with settings of supplementary properties for a control item.

Setting properties of the *values* object:

Property

itemType

Description

A type of an item. This is set by a value of the *Terrasoft.ViewItemType* list. The *BUTTON* value is used for the button.

caption

The button title. It is recommended to set title values by binding to a localizable string of the schema.

click

Binding of the button handler method.

layout

The object of setting a control item location on the grid.

enabled

Controls the the button availability (activity).

visible

Controls the button visibility.

style

Component style. The property should contain the value of the *Terrasoft.controls.ButtonEnums.style* enumeration.

The *Terrasoft.controls.ButtonEnums.style* enumeration contains following values:

Property

Terrasoft.controls.ButtonEnums.style.DEFAULT

Description

Default style.

Terrasoft.controls.ButtonEnums.style.GREEN

Green color button.

Terrasoft.controls.ButtonEnums.style.RED

Red color button.

Terrasoft.controls.ButtonEnums.style.BLUE

Blue color button.

Terrasoft.controls.ButtonEnums.style.GREY

Transparent button. This value is from previous versions of Creatio.

Terrasoft.controls.ButtonEnums.style.TRANSPARENT Transparent button.

You can read more about the diff array in the "[The "diff" array](#)" article.

Examples of implementing a button adding to the edit page

- [How to add the button on the edit page in the combined mode](#)
- [How to add a button to an edit page in the new record add mode](#)
- [How to add a button to a section](#)

How to add a button to an edit page in the new record add mode

Beginner

Easy

Medium

Advanced

Case description

The button opening the primary contact edit page must be added to the new account add page.

By default a pop-up summary is used to add new account. To use the page to add an account select the [Default value] checkbox in the [*Enable account mini page add mode*] system setting . In order for the system setting change to take effect, you must log off and log on to the user.

To complete these case, you need to use the page to add the account.

The edit page mode can be accessed at the creation of the record and when the page is refreshed in the combined mode by pressing F5 key. The vertical list should be disabled.

Source code

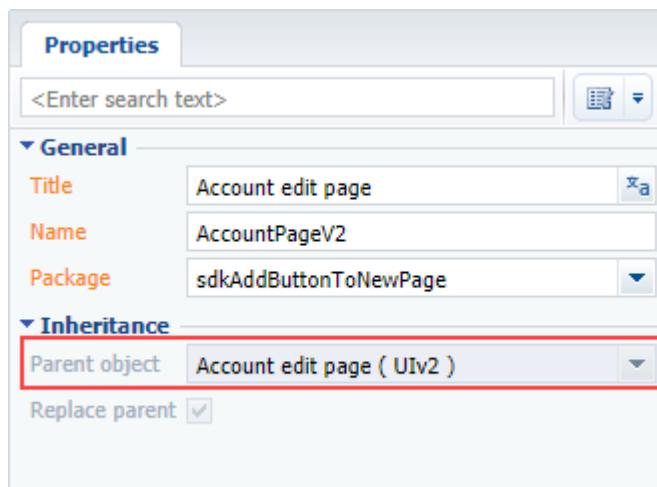
Use this [link](#) to download the case implementation package.

Case implementation algorithm

1. Create a replacing account edit page

A replacing client module must be created and [AccountPageV2] must be specified as the parent object in it (Fig. 1). The procedure of creating a replacing page is covered in the “**Creating a custom client module schema**” article.

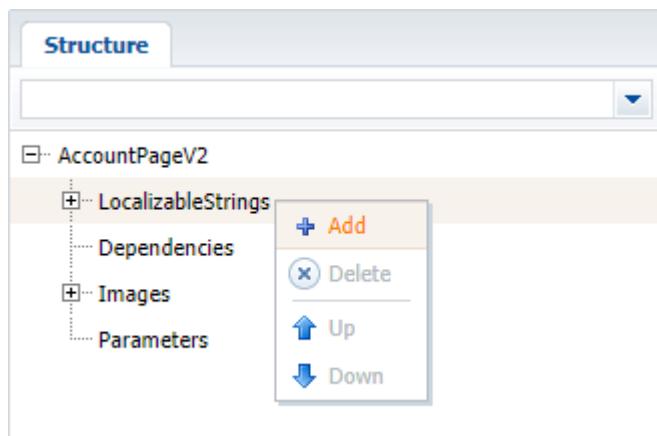
Fig. 1. Properties of the replacing edit page



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

Create a new localizable string (Fig. 2).

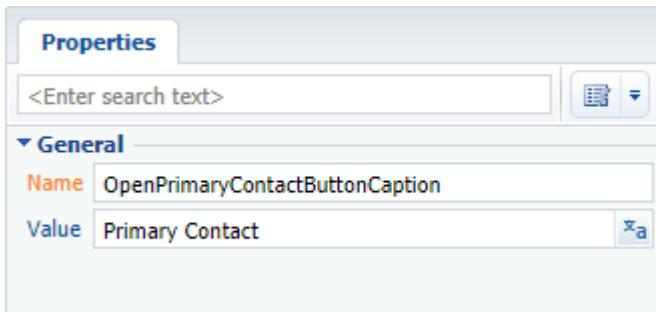
Fig. 2. Adding localized string to the schema



For the created string specify (Fig. 3):

- [Name] – “OpenPrimaryContactButtonCaption”.
- [Value] – “Primary Contact”.

Fig. 3. Properties of the custom localizable string



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

- `isAccountPrimaryContactSet()` – checks if the [Primary contact] field is filled.
- `onOpenPrimaryContactClick()` – button pressing handler method which performs passing to the base contact edit page.

4. Add a button on the edit page

Add an object with the settings determining the button position on the account edit page in the `diff` array.

The replacing schema source code is as follows:

```
define("AccountPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Account",
        // Methods collection of the edit page view model.
        methods: {
            // The method checks whether the [Primary contact] field is completed.
            isAccountPrimaryContactSet: function() {
                return this.get("PrimaryContact") ? true : false;
            },
            // Button press handler method.
            onOpenPrimaryContactClick: function() {
                var primaryContactObject = this.get("PrimaryContact");
                if (primaryContactObject) {
                    // Determining the base contact Id.
                    var primaryContactId = primaryContactObject.value;
                    // Forming the address string.
                    var requestUrl = "CardModuleV2/ContactPageV2/edit/" +
primaryContactId;
                    // Publishing a message and going to the
                    // base contact edit page.
                    this.sandbox.publish("PushHistoryState", {
                        hash: requestUrl
                    });
                }
            }
        },
        // Displaying a button on the edit page.
        diff: [
            // Metadata for adding a new control item - custom button to the page.
            {
                // Indicates that an operation of adding an item to the page is being
                // executed.
                "operation": "insert",
                // Metadata of the parent control item the button is added.
                "parentName": "LeftContainer",
                // Indicates that the button is added to the control items
                collection
            }
        ]
    }
});
```

```

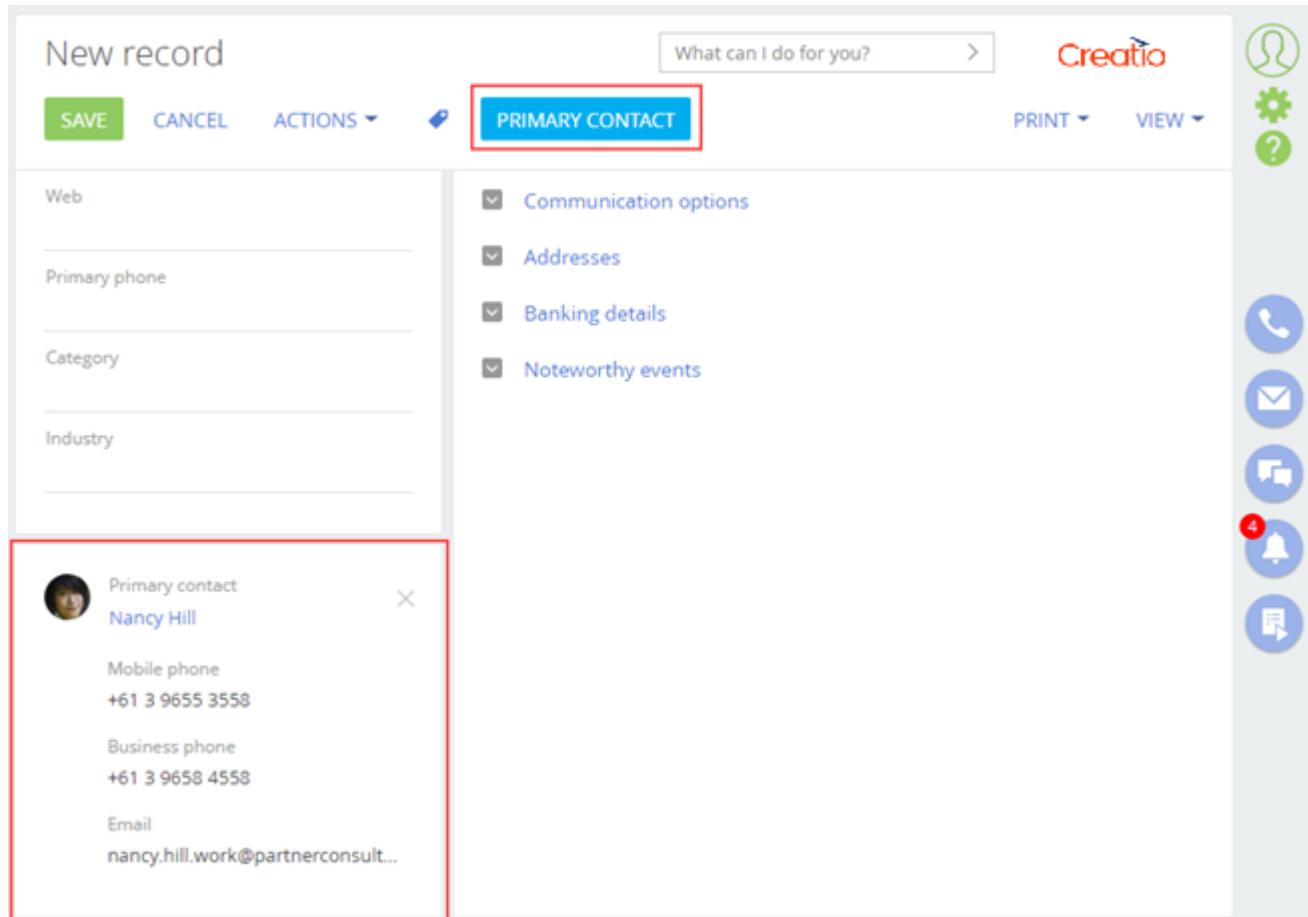
        // of the parent item (which meta-name is specified in the
parentName).
    "propertyName": "items",
    // Meta-name of the added button.
    "name": "PrimaryContactButton",
    // Supplementary properties of the item.
    "values": {
        // Type of the added item is button.
        itemType: Terrasoft.ViewItemType.BUTTON,
        // Binding the button title to a localizable string of the
schema..
        caption: {bindTo:
"Resources.Strings.OpenPrimaryContactButtonCaption"},
        // Binding the button press handler method.
        click: {bindTo: "onOpenPrimaryContactClick"},
        // Binding the property of the button availability.
        enabled: {bindTo: "isAccountPrimaryContactSet"},
        // Setting the button style.
        "style": Terrasoft.controls.ButtonEnums.style.BLUE
    }
}
];
};

});
}
);

```

When the schema is saved and the system web-page is updated, the [Primary contact] button will appear on the new account create page. The button will be activated after filling the [Primary contact] field in the account (Fig. 4).

Fig. 4. Demonstrating the case implementation result



See also

- Adding a button to the edit page
- How to add the button on the edit page in the combined mode

How to add the button on the edit page in the combined mode

Beginner

Easy

Medium

Advanced

Example of implementing the button adding to an edit page

Case description

The button opening the base contact edit page must be added to the account edit page.

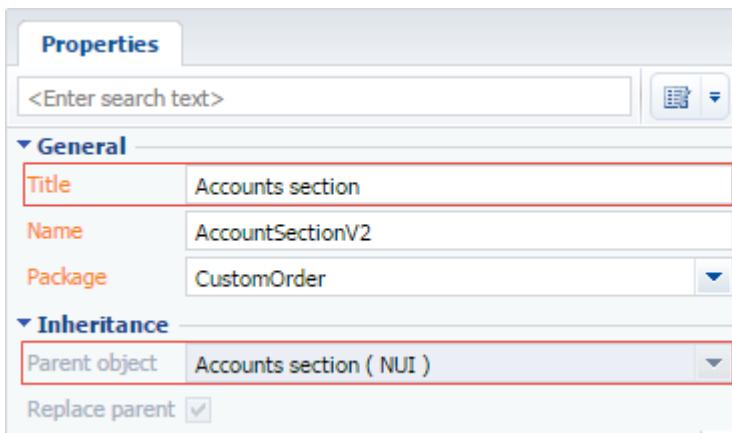
Case implementation algorithm

1. Create the [Accounts] section replacing schema

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

The procedure for creating the replacing page is described in the article [Creating a custom client module schema](#).

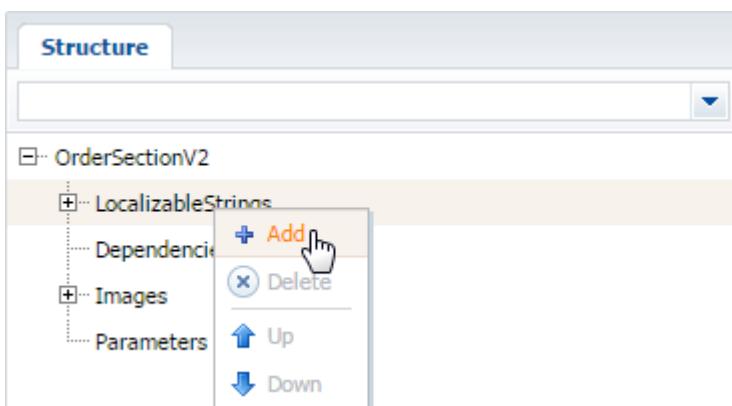
Fig. 1. — Properties of the section replacing page



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

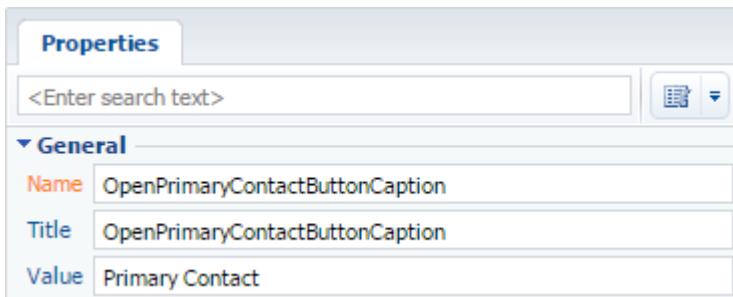
For this purpose, select [Add] by right-clicking the [LocalizableStrings] structure node (Fig. 2).

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



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

Fig. 3. – Properties of a custom localized string



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

- `isAccountPrimaryContactSet` - checks whether the [Primary contact] field of the page is filled;
- `onOpenPrimaryContactClick` - button pressing handler method which goes to the base contact edit page.

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

```
define("AccountSectionV2", [],
  function() {
    return {
      // Name of the edit page object schema.
      entitySchemaName: "Account",
      // Methods collection of the edit page view model.
      methods: {
        // Button press handler method.
        onOpenPrimaryContactClick: function() {
          var activeRow = this.get("ActiveRow");
          if (activeRow) {
            // Determining the base contact Id.
            var primaryId =
this.get("GridData").get(activeRow).get("PrimaryContact").value;
            if (primaryId) {
              // Forming the address string.
              var requestUrl = "CardModuleV2/ContactPageV2/edit/" +
primaryId;
              // Publishing the message and going to the
              // base contact edit page.
              this.sandbox.publish("PushHistoryState", {
                hash: requestUrl
              });
            }
          }
        },
        // The method checks whether the [Base Contact] field is filled.
        isAccountPrimaryContactSet: function() {
          debugger;
          var activeRow = this.get("ActiveRow");
          if (activeRow)
          {
            var pc =
this.get("GridData").get(activeRow).get("PrimaryContact");
            return (pc || pc !== "") ? true : false;
          }
          return false;
        }
      };
    });
});
```

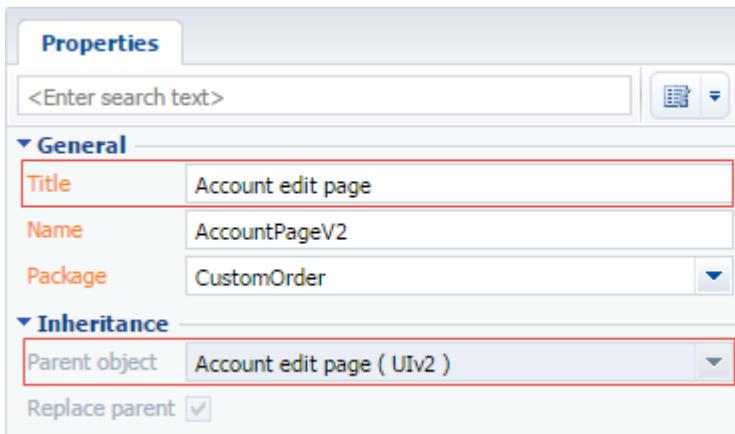
4. Add a configuration object with button location settings on the edit page to the diff array

```
define("AccountSectionV2", [],
  function() {
    return {
      // Name of the edit page object schema.
      entitySchemaName: "Account",
      // Methods collection of the edit page view model.
      methods: {
        // onOpenPrimaryContactClick, isAccountPrimaryContactSet method
      },
      // Setting a button visualization on the edit page.
      diff: [
        // Metadata for adding a custom button to the page.
        {
          // Indicates that an operation of adding an item to the page is
          being executed.
          "operation": "insert",
          // Meta-name of the parent control item where the button is
          added.
          "parentName": "CombinedModeActionButtonsCardLeftContainer",
          // Indicates that the button is added to the control items
          collection
          // of the parent item (which name is specified in the
          parentName).
          "propertyName": "items",
          // Meta-name of the added button. .
          "name": "MainContactButton",
          // Supplementary properties of the item.
          "values": {
            // Type of the added item is button.
            itemType: Terrasoft.ViewItemType.BUTTON,
            // Binding the button title to a localizable string of the
            schema.
            caption: {bindTo:
              "Resources.Strings.OpenPrimaryContactButtonCaption"},
            // Binding the button press handler method.
            click: {bindTo: "onOpenPrimaryContactClick"},
            // Binding the property of the button availability.
            enabled: {bindTo: "isAccountPrimaryContactSet"},
            // Setting the field location.
            "layout": {
              "column": 1,
              "row": 6,
              "colSpan": 1
            }
          }
        }
      ],
    });
});
```

5. Save the created replacingpage schema**6. Create a replacingaccount edit page**

A replacing client module must be created and [Account edit page] must be specified as the parent object (Fig. 4). The procedure for creating the replacingpage is described in the article **Creating a custom client module schema**.

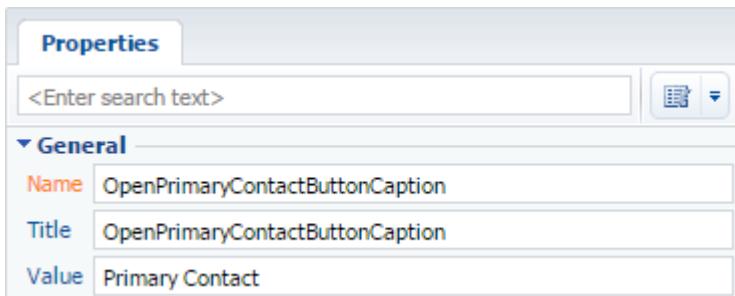
Fig. 4. – Properties of the replacing edit page



7. Add a string with the button title to the localizable strings collection of the replacing page schema

Fill in properties for the created string as shown on Fig. 5.

Fig. 5. – Properties of a custom localized string



8. Add the method implementation to the methods collection of the page view model

- `isAccountPrimaryContactSet` – checks whether the [Base Contact] field of the page is filled;
- `onOpenPrimaryContactClick` – button pressing handler method which goes to the base contact edit page.

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

```
define("AccountPageV2", [],
  function() {
    return {
      // Name of the edit page object schema.
      entitySchemaName: "Account",
      // Methods collection of the edit page view model.
      methods: {
        // The method checks whether the [Base Contact] field is filled.
        isAccountPrimaryContactSet: function() {
          return this.get("PrimaryContact") ? true : false;
        },
        // Button press handler method.
        onOpenPrimaryContactClick: function() {
          // Determining the base contact Id.
          var primaryId = this.get("PrimaryContact").value;
          if (primaryId) {
            // Forming the address string.
            var requestUrl = "CardModuleV2/ContactPageV2/edit/" +
              primaryId;
            // Publishing a message and going to the
            // base contact edit page.
          }
        }
      }
    }
  }
);
```

```
        this.sandbox.publish("PushHistoryState", {
            hash: requestUrl
        });
    }
}
);
});
```

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

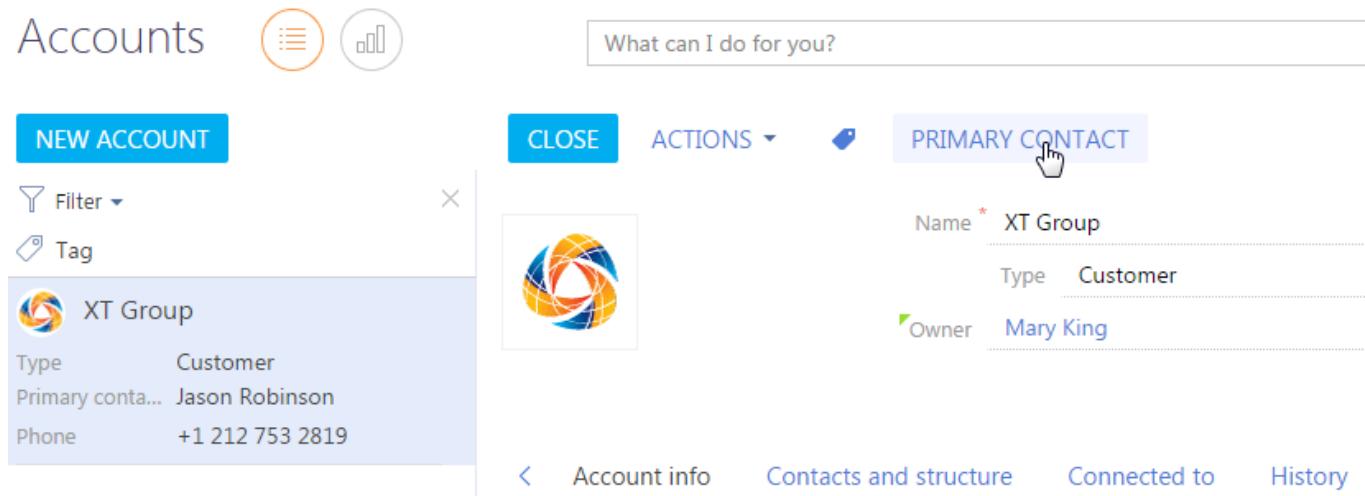
```
define("AccountPageV2", [],
function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Account",
        // Methods collection of the edit page view model.
        methods: {
            // onOpenPrimaryContactClick, isAccountPrimaryContactSet method
            implementation
        },
        // Setting a button visualization on the edit page.
        diff: [
            // Metadata for adding a new control item - custom button - to the
            page.
            {
                // Indicates that an operation of adding an item to the page is
                // being executed.
                "operation": "insert",
                // Meta-name of the parent control item where the button is
                // added.
                "parentName": "LeftContainer",
                // Indicates that the button is added to the control items
                // collection
                // of the parent item (whose name is specified in the
                // parentName).
                "propertyName": "items",
                // Meta-name of the added button.
                "name": "MainContactButton",
                // Supplementary properties of the item.
                "values": {
                    // Type of the added item is button.
                    itemType: Terrasoft.ViewItemType.BUTTON,
                    // Binding the button title to a localizable string of the
                    // schema.
                    caption: {bindTo:
                        "Resources.Strings.OpenPrimaryContactButtonCaption"
                    },
                    // Binding the button press handler method.
                    click: {bindTo: "onOpenPrimaryContactClick"},
                    // Binding the property of the button availability.
                    enabled: {bindTo: "isAccountPrimaryContactSet"},
                    // Setting the field location.
                    "layout": {
                        "column": 1,
                        "row": 6,
                        "colSpan": 1
                    }
                }
            }
        ]
    };
});
```

```
});
```

10. Save the created replacing page schema

When the schema is saved and the system web-page is updated, the [Base Contact] button will appear on the account edit page. The button will be activated if the [Base Contact] field of the account is filled (Fig. 6).

Fig. 6. – Demonstrating the case implementation result



How to add a field with an image to the edit page

Beginner Easy Medium **Advanced**

Introduction

There are certain peculiarities of adding a field with image (contact's photo, product picture, account logo, etc.) to an edit page:

1. The object column used for a field with image should have the “Link to image” type. Specify it as the [Image] system column of the object.
2. Add the default image to the edit page image schema collection.
3. A field with image is added to the *diff* edit page schema array with usage of the additional *image-edit-container* CSS-class container-wrapper.
4. The *values* property of the configuration object containing settings of a field with image must include the following properties:
 - *getSrcMethod* – method receiving image by link
 - *onPhotoChange* – method called upon image modification
 - *Readonly* – property defining the capability of image editing (changing, deleting)
 - *Generator* – control element generator. Indicate the *ImageCustomGeneratorV2.generateCustomImageControl* for the field with image
 - *beforeFileSelected* – method called before opening the image selection dialog box
5. Add the following to the edit page schema collection:
 - method receiving image by link
 - method called upon image modification
 - method saving the link to a modified image in the object column
 - method called before opening the image selection dialog box

Case description

Adding a field with logo to the knowledge base article edit page.

Source code

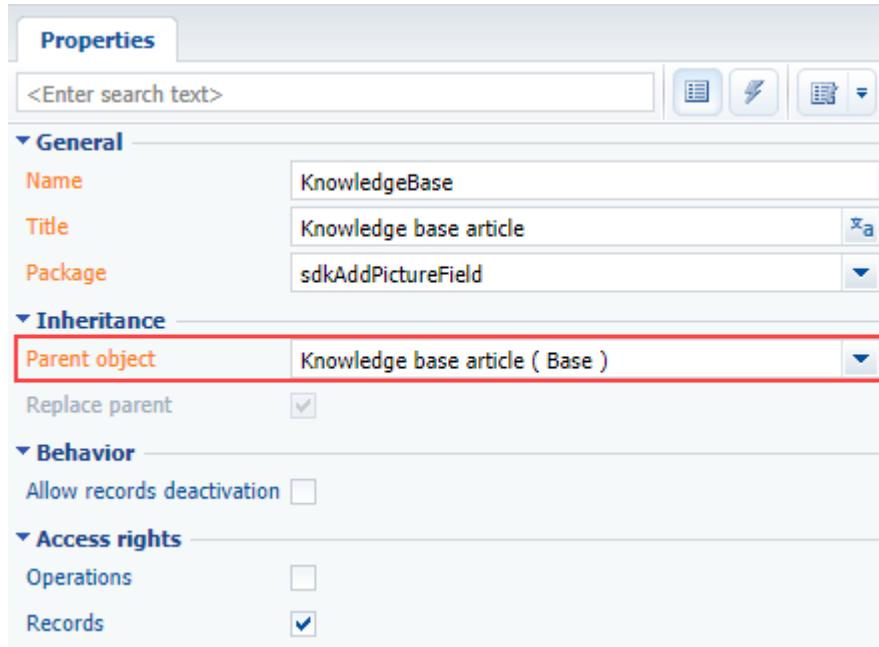
You can download the package with case implementation using the following [link](#)..

Case implementation algorithm

1. Creating the [Knowledge base] replacing object.

Create the [Knowledge base article] replacing object (Fig.1). Learn more about creating a replacing object in the “[Creating the entity schema](#)” article.

Fig. 1. Properties of the object replacing schema

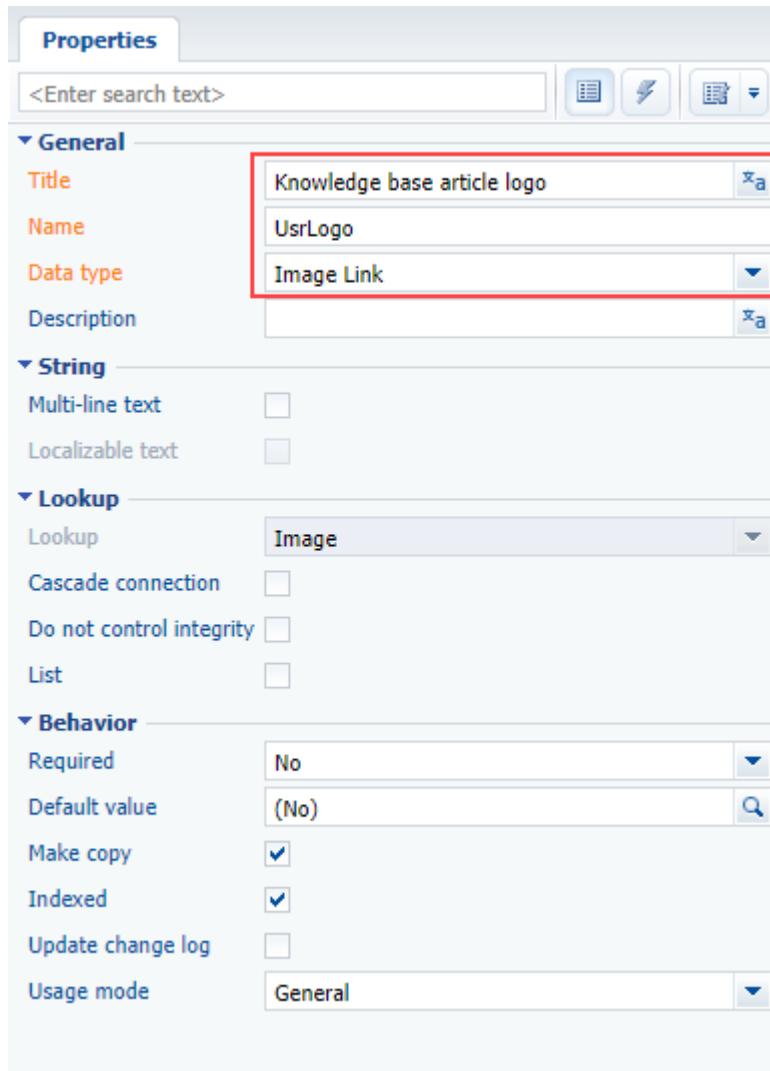


2. Adding a new column to the replacing object.

For the created column specify (Fig. 2):

- [Title] – "Knowledge base article logo"
- [Name] – "UsrLogo"
- [Data type] – "Image Link"

Fig. 2. Adding a custom column to the replacing object



Indicate the created column as the [Image] object system column (Fig.3).

Fig. 3. Setting up the created column as the system column

The screenshot shows the 'Properties' tab of the Object Designer. Under the 'General' section, the 'Name' is set to 'KnowledgeBase' and the 'Title' is 'Knowledge base article'. In the 'System Columns' section, the 'Image' field is set to 'Knowledge base article logo' and is highlighted with a red border. Other columns like 'Id', 'Displayed value', and 'Owner' are also listed.

Properties	
<Enter search text>	
Name	KnowledgeBase
Title	Knowledge base article x
Change Log Object Name	
Permission Object Name	
Localization Object Name	
Description	
Package	sdkAddPictureField
Inheritance	
Behavior	
Access rights	
System Columns	
Id	Id
Displayed value	Name
Image	Knowledge base article logo
Sorting in Lists	
Parent in Hierarchy	
Owner	
History Columns	

To view all object properties, switch to the object property advanced view mode. You can learn more about object designer capabilities in the "["Workspace of the Object Designer"](#)" article.

Save and publish the object schema after you set up all properties.

3. Creating a replacing client module for the edit page.

Create a replacing client module and specify the [Knowledge base edit page] (*KnowledgeBasePageV2*) as the parent object in it (Fig. 4). The procedure of creating a replacing page is covered in the "["Creating a custom client module schema"](#)" article.

Fig. 4. Properties of the replacing edit page

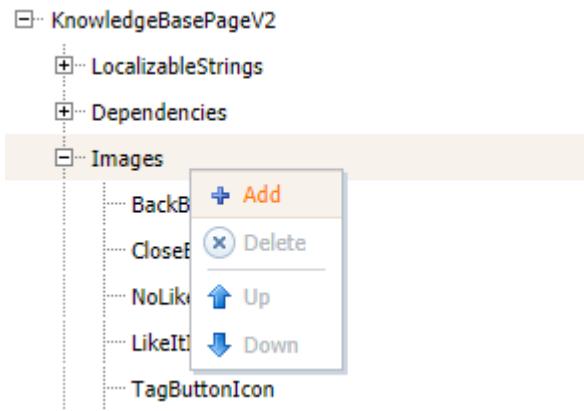
The screenshot shows the 'Properties' tab of the Object Designer. Under the 'General' section, the 'Title' is 'Knowledge base edit page' and the 'Name' is 'KnowledgeBasePageV2'. In the 'Inheritance' section, the 'Parent object' field is set to 'Knowledge base edit page (UIv2)' and is highlighted with a red border. A checkbox 'Replace parent' is also present.

Properties	
<Enter search text>	
Title	Knowledge base edit page x
Name	KnowledgeBasePageV2
Package	sdkAddPictureField
Inheritance	
Parent object	Knowledge base edit page (UIv2)
<input checked="" type="checkbox"/> Replace parent	

4. Adding a default image to the [Images] resources of the edit page schema.

Add the default image to the page replacing schema image collection (Fig.5).

Fig. 5. Adding default image to the image schema resources



For the created image specify (Fig. 6):

- [Name] – "DefaultLogo"
- [Image] – file containing the default image (Fig.7)

Fig. 6. Schema resource properties

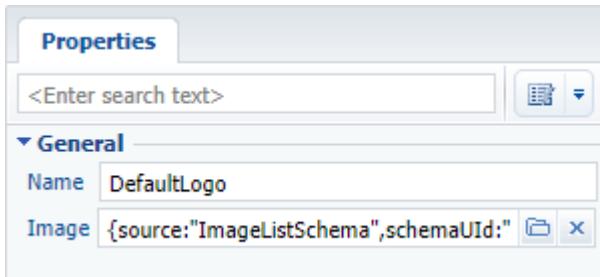


Fig. 7. Default image for the knowledge base article



5. Setting up displaying of a field with logo on the edit page

The field with logo should be placed in the upper part of the account edit page. In the base implementation the fields are placed in such a way that adding a logo can violate the page interface. That is why you need to rearrange the location of existing fields when locating a new field.

Add the configuration object of the field with logo with the necessary parameters to the *diff* array property of the view model (see the source code below) and describe the modifications of the fields located in the upper part of the page: [Name], [ModifiedBy] and [Type].

The field with image is added to the page by using the additional *PhotoContainer* container-wrapper with the ["image-edit-container"] class.

6. Adding implementation of the following methods to the page view model method collection:

- *getPhotoSrcMethod()* – receives image by link
- *beforePhotoFileSelected()* – is called before opening the image selection dialog box
- *onPhotoChange* – is called upon image modification
- *onPhotoUploaded()* – saves the link to a modified image in the object column

The replacing schema source code is as follows:

```
define("KnowledgeBasePageV2", ["KnowledgeBasePageV2Resources",
"ConfigurationConstants"],
function(resources, ConfigurationConstants) {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "KnowledgeBase",
        // Edit page view model methods.
        methods: {
            // Called before opening the image selection dialog box.
            beforePhotoFileSelected: function() {
                return true;
            },
            // Receives image by link.
            getPhotoSrcMethod: function() {
                // Receiving a link to the image in the object column.
                var imageColumnValue = this.get("UsrLogo");
                // If the link is set, the method returns the url of the image
file.
                if (imageColumnValue) {
                    return this.getSchemaImageUrl(imageColumnValue);
                }
                // If the link is not set, it returns the default image.
                return
            }
        }
    },
    // Processes the image modification.
    // photo – image file.
    onPhotoChange: function(photo) {
        if (!photo) {
            this.set("UsrLogo", null);
            return;
        }
        // The file is uploaded to the database. onPhotoUploaded is
called when uploading is finished.
        this.Terrasoft.ImageApi.upload({
            file: photo,
            onComplete: this.onPhotoUploaded,
            onError: this.Terrasoft.emptyFn,
            scope: this
        });
    },
    // Saves the link to a modified image.
    // imageId – Id of the saved file from the database.
    onPhotoUploaded: function(imageId) {
        var imageData = {
            value: imageId,
            displayValue: "Image"
        };
        // The image column is assigned a link to the image.
        this.set("UsrLogo", imageData);
    }
},
// diff: /**SCHEMA_DIFF*/
// Container-wrapper that the component will be located in.
```

```
{  
    // Adding operation.  
    "operation": "insert",  
    // Parent container meta-name, where the component is added.  
    "parentName": "Header",  
    // The image is added to the component collection of the  
    // parent container.  
    "propertyName": "items",  
    // Schema component meta-name, involved in the action.  
    "name": "PhotoContainer",  
    // Properties passed to the component structure.  
    "values": {  
        // Element type - container.  
        "itemType": Terrasoft.ViewItemType.CONTAINER,  
        // CSS-class name.  
        "wrapClass": ["image-edit-container"],  
        // Locating in the parent container.  
        "layout": { "column": 0, "row": 0, "rowSpan": 3, "colSpan": 3  
    },  
        // Child element array.  
        "items": []  
    }  
},  
// The [UsrLogo] field - the field with account logo.  
{  
    "operation": "insert",  
    "parentName": "PhotoContainer",  
    "propertyName": "items",  
    "name": "UsrLogo",  
    "values": {  
        // Method receiving image by link.  
        "getSrcMethod": "getPhotoSrcMethod",  
        // Method called upon image modification.  
        "onPhotoChange": "onPhotoChange",  
        // Method called before opening the image selection dialog  
        // box.  
        "beforeFileSelected": "beforePhotoFileSelected",  
        // Property defining the capability of image editing  
        // (changing, deleting).  
        "readonly": false,  
        // Control element view-generator.  
        "generator":  
    "ImageCustomGeneratorV2.generateCustomImageControl"  
    }  
},  
// Rearranging the location of the [Name] field.  
{  
    // Merge operation.  
    "operation": "merge",  
    "name": "Name",  
    "parentName": "Header",  
    "propertyName": "items",  
    "values": {  
        "bindTo": "Name",  
        "layout": {  
            "column": 3,  
            "row": 0,  
            "colSpan": 20  
        }  
    }  
},  
// Rearranging the location of the [ModifiedBy] field.
```

```
{
    "operation": "merge",
    "name": "ModifiedBy",
    "parentName": "Header",
    "propertyName": "items",
    "values": {
        "bindTo": "ModifiedBy",
        "layout": {
            "column": 3,
            "row": 2,
            "colSpan": 20
        }
    }
},
// Rearranging the location of the [Type] field.
{
    "operation": "merge",
    "name": "Type",
    "parentName": "Header",
    "propertyName": "items",
    "values": {
        "bindTo": "Type",
        "layout": {
            "column": 3,
            "row": 1,
            "colSpan": 20
        },
        "contentType": Terrasoft.ContentType.ENUM
    }
}
]/**SCHEMA_DIFF*/
);
}
);
```

The default logo will be displayed on the knowledge base article edit page after you save the schema and update the application page. When you hover over the image, you will see an action menu appear. You can use it to delete the image or set up a new one for a specific knowledge base article (Fig.9).

Fig. 8. Default logo

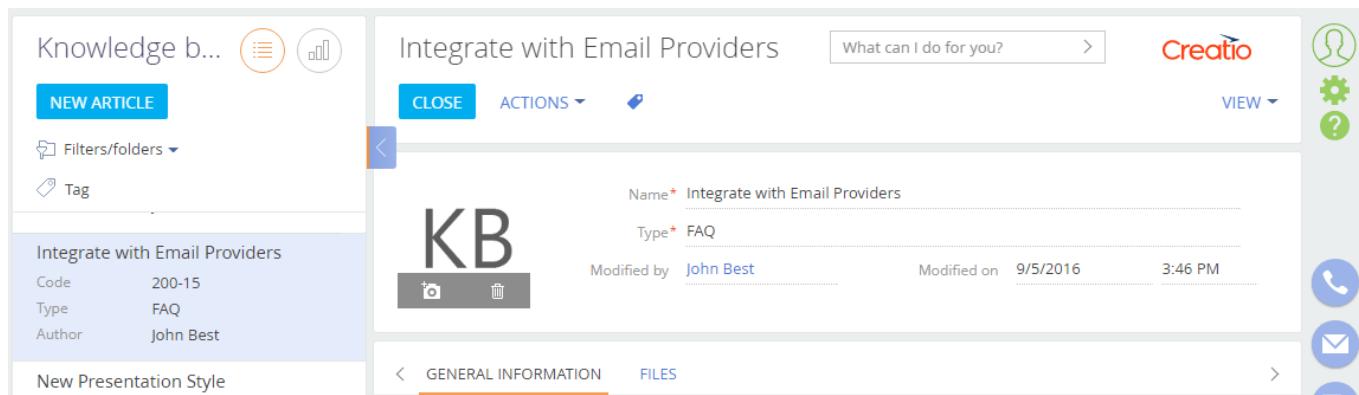
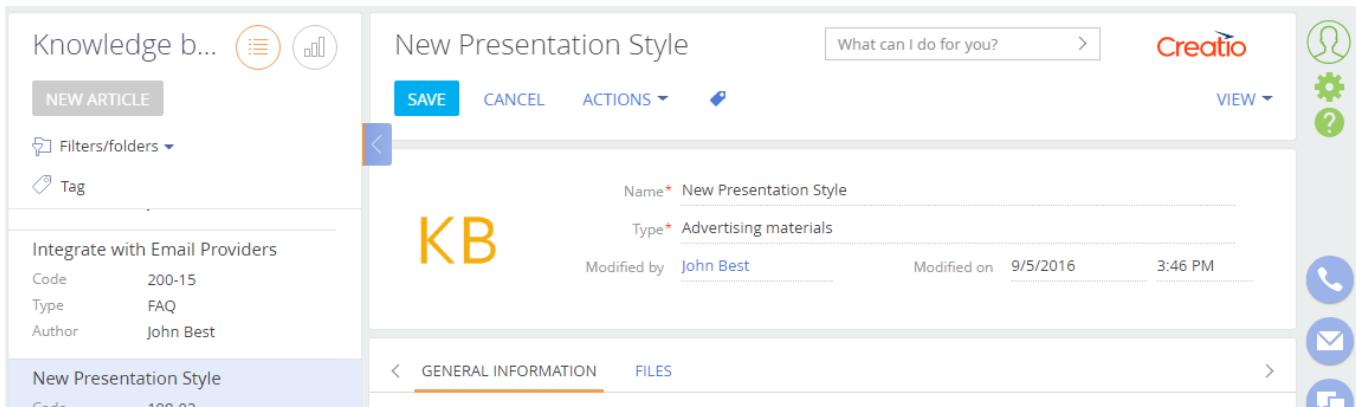


Fig. 9. Custom logo



How to add the color select button to the edit page

Beginner **Easy** **Medium** **Advanced**

Introduction

One of Creatio control elements is the color button (Fig.1).

Fig. 1. Color button



Algorithm of adding the color button to object edit page:

1. Add the [Text (50 characters)] data type column to the object that will store information about the selected color.
2. Add the [COLOR_BUTTON] type element description to the *diff* array. Set up binding to the column added on previous step for the *value* property of this element.
3. Add a button label via the [LABEL] control element if needed.

Case description

Adding color button to product edit page.

Source code

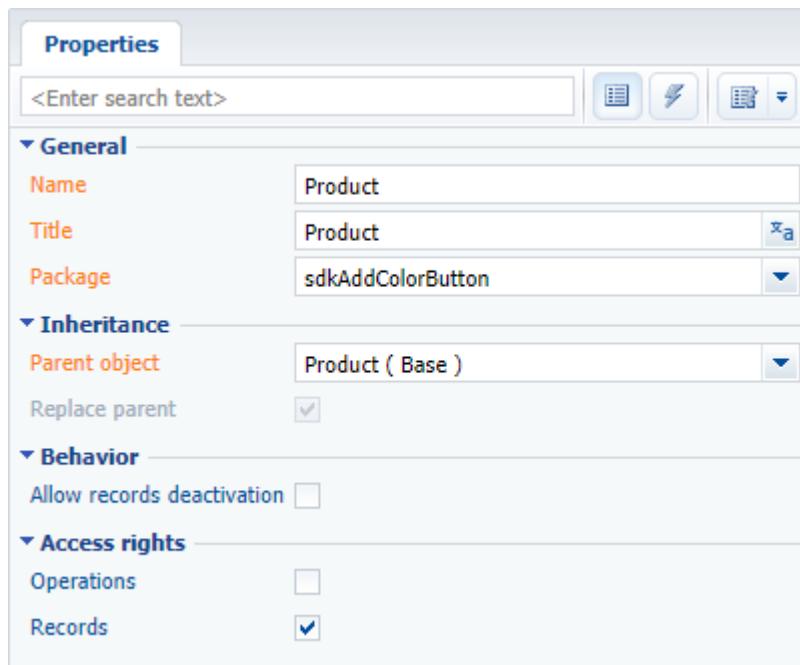
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Creating a “Product” replacing object and adding the [UsrColor] column to it.

Create the [Product] replacing object (Fig.2). Learn more about creating a replacing object in the “[Creating the entity schema](#)” article.

Fig. 2. Configuration object properties



Add a new column (Fig.3) and indicate the following properties (Fig.4):

- [Title] – “Color”
- [Name] – "UsrColor"
- [Data type] – "Text (50 characters)"

Fig. 3. Adding a new column

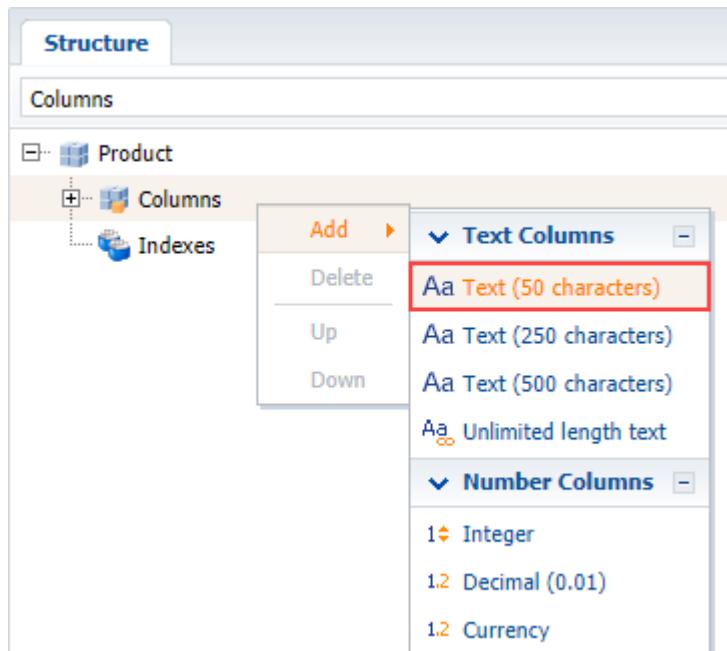


Fig. 4. Properties of the added column

The screenshot shows the 'Properties' dialog box for a column named 'UsrColor'. The 'General' section is expanded, showing the following settings:

- Title:** Color
- Name:** UsrColor
- Data type:** Text (50 characters) (highlighted with a red box)
- Description:** (empty)

The 'String' and 'Lookup' sections are collapsed. The 'Behavior' section is expanded, showing:

- Required:** No
- Default value:** (No) (with a search icon)
- Make copy:** checked
- Indexed:** (unchecked)
- Update change log:** (unchecked)
- Usage mode:** General

In the 'Structure' pane on the left, under the 'Product' category, the 'UsrColor' column is selected and highlighted with a red box.

Save and publish the object schema after you set up all properties.

2. Creating a product replacing edit page in custom package

Create a replacing client module and specify the [Edit page – Product], *ProductPageV2* schema as the parent object in it (Fig. 5). The procedure of creating a replacing page is covered in the “[Creating a client schema](#)“ article.

Fig. 5. Properties of the product edit page replacing schema

The screenshot shows the 'Properties' dialog box for an edit page schema. The 'General' section is expanded, showing the following settings:

- Title:** Edit page - Product
- Name:** ProductPageV2
- Package:** sdkAddColorButton (highlighted with a red box)

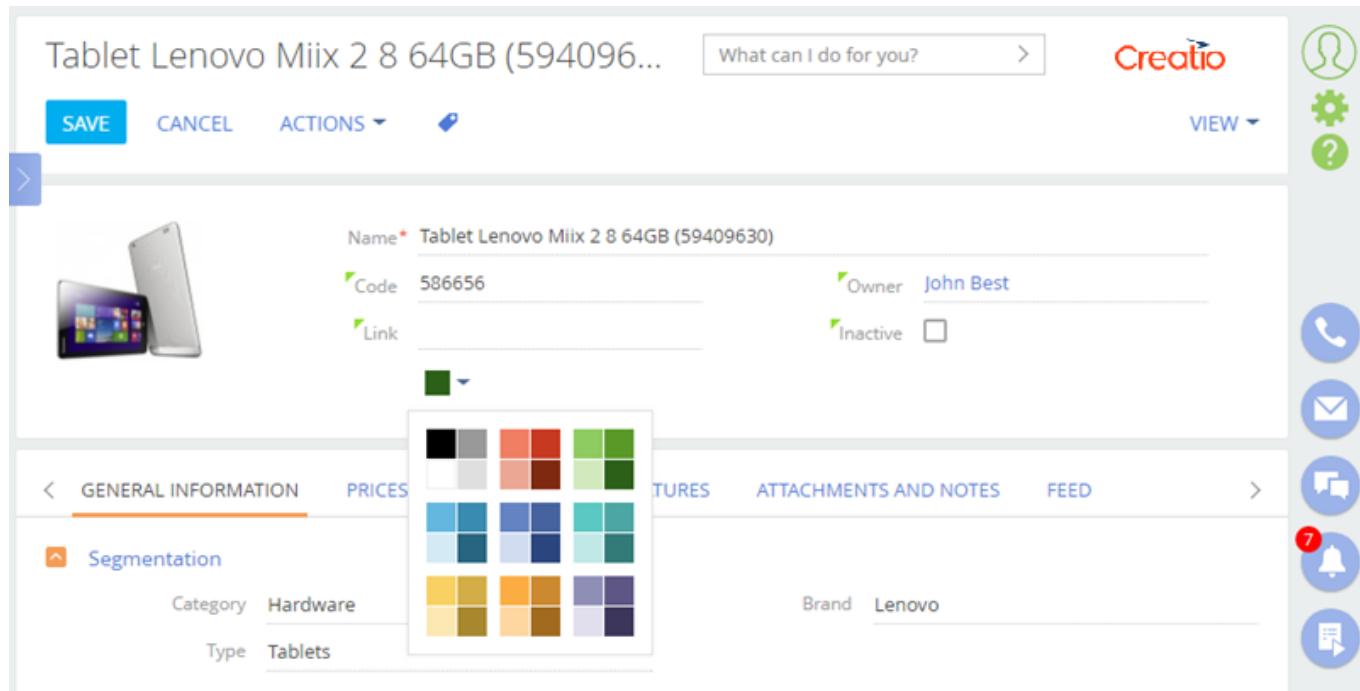
The 'Inheritance' section is collapsed. The 'Replace parent' checkbox is checked.

The replacing schema source code is as follows:

```
define("ProductPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Product",
        diff: /**SCHEMA_DIFF*/ [
            // Color button.
            {
                // Operation of adding.
                "operation": "insert",
                // Meta-name of the parent container where the component is added.
                "parentName": "ProductGeneralInfoBlock",
                // The button is added to component collection
                // of the parent container.
                "propertyName": "items",
                // The name of schema component involved in action.
                "name": "ColorButton",
                // Properties transferred to the component constructor.
                "values": {
                    // Element type - color button.
                    "itemType": this.Terrasoft.ViewItemType.COLOR_BUTTON,
                    // Binding of the control element value to the view model column.
                    "value": { "bindTo": "UsrColor" },
                    // Button location.
                    "layout": { "column": 5, "row": 6, "colSpan": 12 }
                }
            }
        ] /**SCHEMA_DIFF*/
    };
});
```

After saving the schema and refreshing the application page the color button will be displayed on the product edit page (Fig.6).

Fig. 6. Case result



How to add multi-currency field

Beginner

Easy

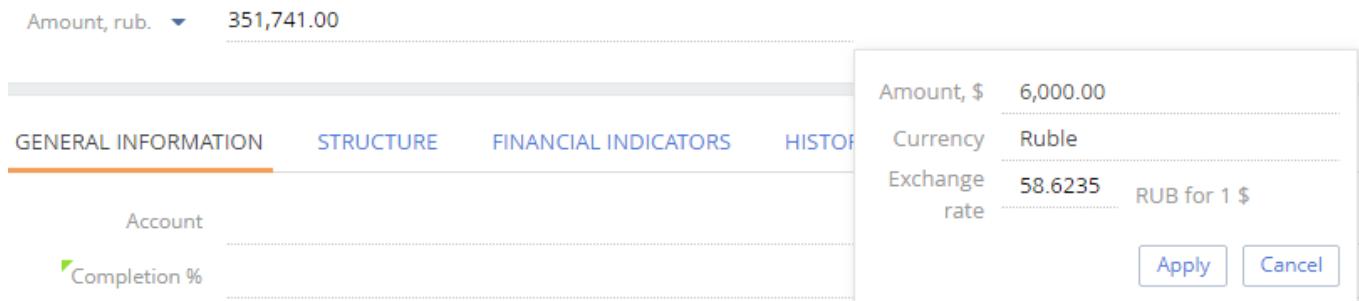
Medium

Advanced

Introduction

One of the common Creatio configuration tasks is adding a [Multi-currency field] control on a page. This control enables users to specify the currency when entering monetary sums. It also enables fixing the sum equivalent in the base currency specified in the system settings. The sum is automatically converted as per the exchange rate when the currency changes. Fig.1 displays a multi-currency field in the Creatio interface.

Fig. 1. Multi-currency field



To add a multi-currency field on an edit page:

1. Add 4 columns to the object schema:

- [Currency] lookup column
- [Exchange rate] column
- [Amount] column to store the total sum in the selected currency
- [Amount in the base currency] column to store the sum in the base currency

The object schema should only contain one column – to store the total sum in the selected currency. The rest of columns can be virtual, unless the business task requires their values to be stored in the database. They can be determined as attributes in the view model schema.

2. Specify the following 3 modules as dependencies in declaring the view model class:

- *MoneyModule*,
- *MultiCurrencyEdit*,
- *MultiCurrencyEditUtilities*.

3. Connect the *Terrasoft.MultiCurrencyEditUtilities* mixin to the view model and initialize it in the *init()* overridden method.

4. Add a configuration object with the multi-currency field settings to the *diff* array of the edit page view model schema. In addition to common control properties, the *values* property must contain:

- *primaryAmount* – name of the column that contains the amount in the base currency
- *currency* – name of the column with reference to the currency lookup
- *rate* – name of the column that contains the currency exchange rate
- *generator* – control generator Specify *MultiCurrencyEditViewGenerator.generate* for a multi-currency field.

5. Add logic for recalculating the sum according to the currency. Apply the calculated field mechanism, as described in the **Adding calculated fields** article.

Case description

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

Source code

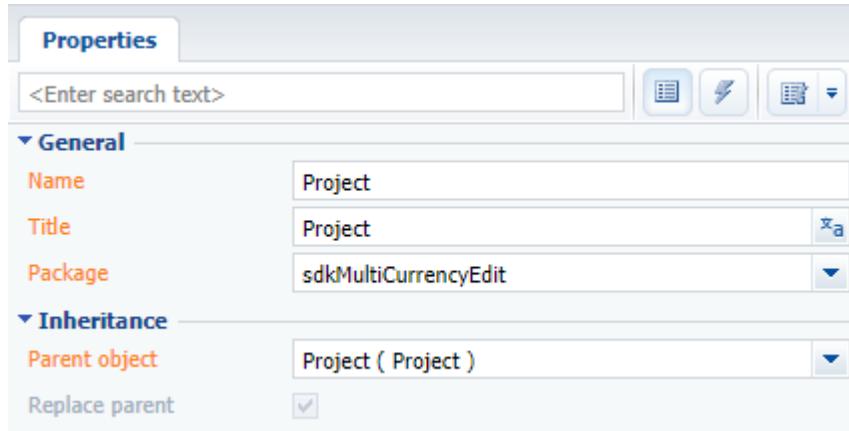
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Add the necessary columns to the object replacing schema

Create the replacing schema of the [Project] object in the custom package (Fig. 2). More information about creating a replacing object and adding columns is available in the “[Adding a new field to the edit page](#)” article.

Fig. 2. Properties of the object replacing schema



Add 4 columns with properties (see Fig. 3 – Fig. 6) to the replacing schema. Column properties in the **Object Designer** are displayed in the extended mode.

Fig. 3. The [UsrCurrency] column properties

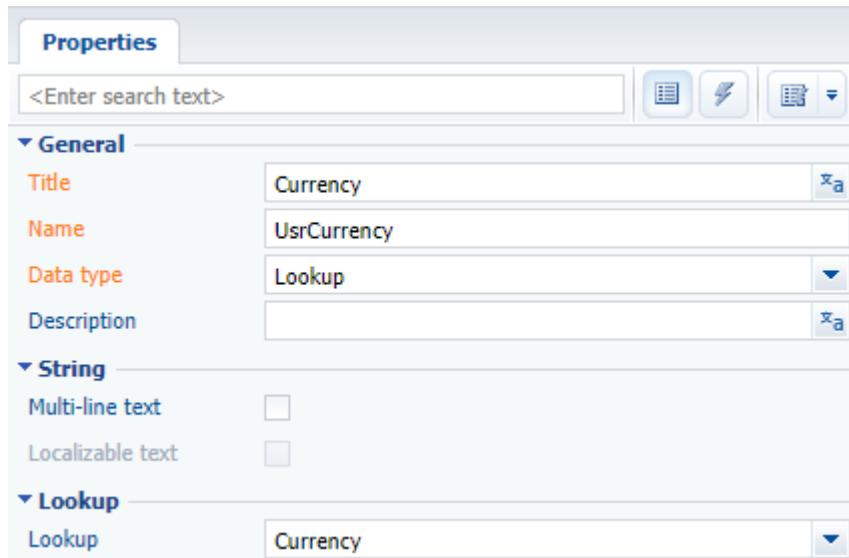


Fig. 4. The [UsrAmount] column properties

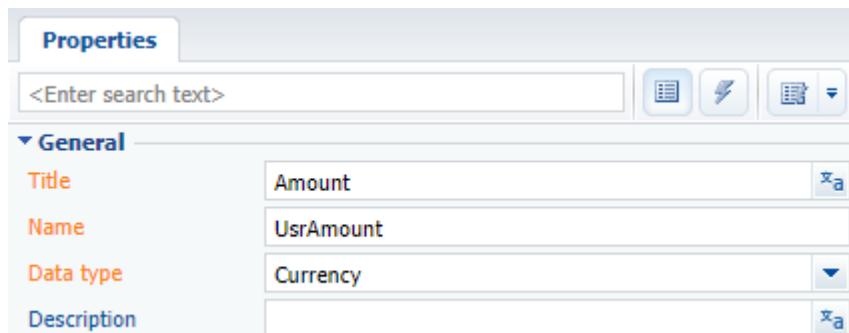


Fig. 5. The [UsrPrimaryAmount] column properties

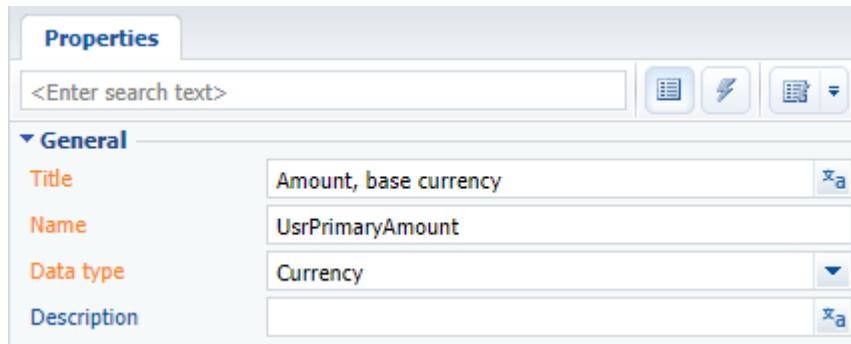
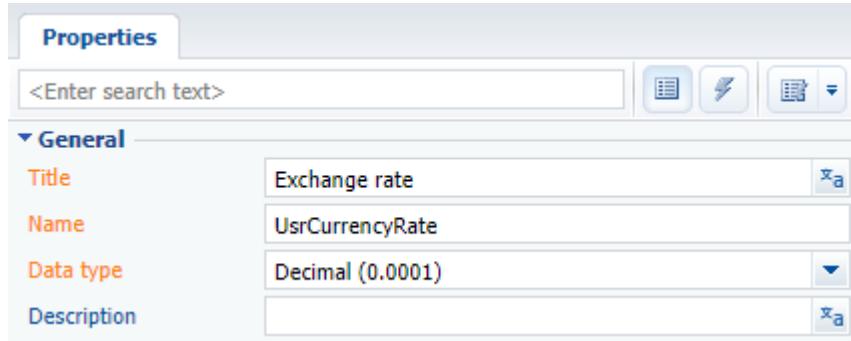


Fig. 6. The [UsrCurrencyRate] column properties



To the [UsrCurrency] column, add the default value – the [Base currency] system setting (Fig.7).

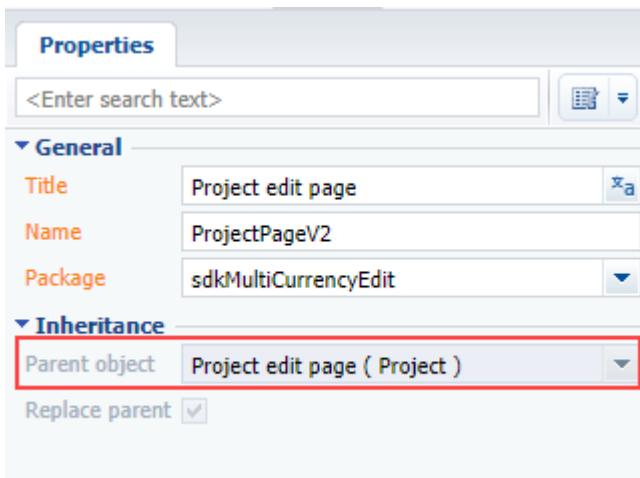
Fig. 7. The default value for the [UsrCurrency] column

The screenshot shows the 'Properties' dialog for a column named 'UsrCurrency'. In the 'General' tab, the 'Default value' field is configured to 'Select from System Settings' with the name 'Base currency'. The 'Properties' panel on the right shows the 'Name' as 'UsrCurrency', 'Data type' as 'Currency', and the 'Default value' as '(System Setting)'. Other settings like 'Required' (No), 'Behavior', and 'Indexes' are also visible.

2. Create a project replacing edit page in custom package

Create a replacing client module and specify the [Project edit page], *ProjectPageV2* schema as its parent object (Fig. 8). The procedure of creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 8. Properties of the [Projects] replacing edit page



Add the following modules as dependencies when declaring view model class: *MoneyModule*, *MultiCurrencyEdit*, *MultiCurrencyEditUtilities* (see the source code below).

3. Add the necessary attributes

Specify the *UsrCurrency*, *UsrCurrencyRate*, *UsrAmount* and *UsrPrimaryAmount* attributes that correspond to the added columns of the object schema in the *attributes* property of the edit page view model schema.

The multi-currency module operates only with the *Currency* column, so in addition you need to create a *Currency* attribute and declare a virtual column in it. Connect this column with the previously created *UsrCurrency* column via a handler method.

To ensure the correct operation of the multi-currency module, add the *CurrencyRateList* (currency rate collection) and *CurrencyButtonMenuList* (collection for the button of selecting the currency) attributes (see the source code below).

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

Declare the *Terrasoft.MultiCurrencyEditUtilities* mixin in the *mixins* property of the page view model schema. Initialize it in the *init()* overridden method of the view model schema (see the following code below).

6. Implement the recalculation logic according to the currency

Add handler methods of the attribute dependencies in the *methods* collection of the page view model schema (see the following code below).

7. Add the multi-currency field on the page

Add the configuration object with the multi-currency field settings to the *diff* array of the edit page view model schema.

The replacing schema source code is as follows:

```
// Specify modules as dependencies
// MoneyModule, MultiCurrencyEdit and MultiCurrencyEditUtilities.
define("ProjectPageV2", ["MoneyModule", "MultiCurrencyEdit",
"MultiCurrencyEditUtilities"],
function(MoneyModule, MultiCurrencyEdit, MultiCurrencyEditUtilities) {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Project",
        // Attributes of the view model.
        attributes: {
            // Currency.
            "UsrCurrency": {

```

```
// Attribute data type is a lookup.
"dataValueType": this.Terrasoft.DataValueType.LOOKUP,
// Configuration of the lookup.
"lookupListConfig": {
    "columns": ["Division", "Symbol"]
}
},
// Exchange rate.
"UsrCurrencyRate": {
    "dataValueType": this.Terrasoft.DataValueType.FLOAT,
    // Attribute dependencies.
    "dependencies": [
        {
            // Columns on which the attribute depends
            "columns": ["UsrCurrency"],
            // Handler method.
            "methodName": "setCurrencyRate"
        }
    ]
},
// Amount.
"UsrAmount": {
    "dataValueType": this.Terrasoft.DataValueType.FLOAT,
    "dependencies": [
        {
            "columns": ["UsrCurrencyRate", "UsrCurrency"],
            "methodName": "recalculateAmount"
        }
    ]
},
// Amount in base currency.
"UsrPrimaryAmount": {
    "dependencies": [
        {
            "columns": ["UsrAmount"],
            "methodName": "recalculatePrimaryAmount"
        }
    ]
},
// Currency is a virtual column for compatibility with the
MultiCurrencyEditUtilities module.
"Currency": {
    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
    "dataValueType": this.Terrasoft.DataValueType.LOOKUP,
    "lookupListConfig": {
        "columns": ["Division"]
    },
    "dependencies": [
        {
            "columns": ["Currency"],
            "methodName": "onVirtualCurrencyChange"
        }
    ]
},
// Currency rate collection
"CurrencyRateList": {
    dataValueType: this.Terrasoft.DataValueType.COLLECTION,
    value: this.Ext.create("Terrasoft.Collection")
},
// Collection for the currency button menu
"CurrencyButtonMenuList": {
    dataValueType: this.Terrasoft.DataValueType.COLLECTION,
```

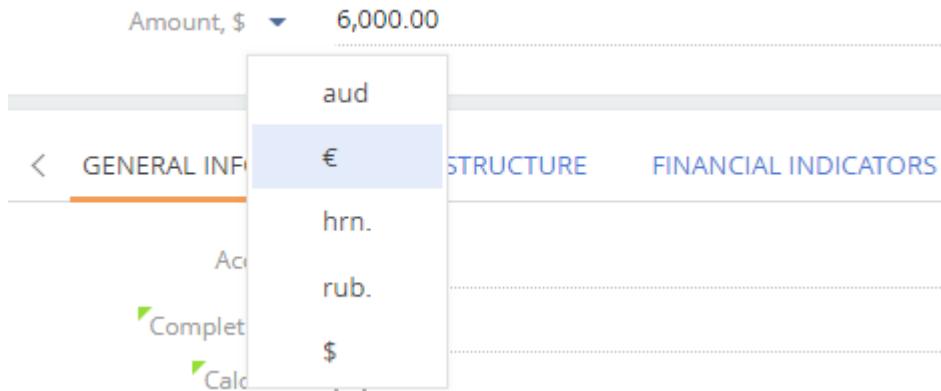
```
        value: this.Ext.create("Terrasoft.BaseViewModelCollection")
    }
},
// View model mixins.
mixins: {
    // Mixin that controls multicurrency on the edit page.
    MultiCurrencyEditUtilities: "Terrasoft.MultiCurrencyEditUtilities"
},
// Methods of the page view model
methods: {
    // Overriding the Terrasoft.BasePageV2.onEntityInitialized() basic
method.
    onEntityInitialized: function() {
        // Calling the parent implementation of the onEntityInitialized
method.
        this.callParent(arguments);
        this.set("Currency", this.get("UsrCurrency"), {silent: true});
        // Initialization of the mixin controlling the multi-currency.
        this.mixins.MultiCurrencyEditUtilities.init.call(this);
    },
    // Sets the exchange rate.
    setCurrencyRate: function() {
        // Loads the exchange rate at the beginning of the project.
        MoneyModule.LoadCurrencyRate.call(this, "UsrCurrency",
"UsrCurrencyRate", this.get("StartDate"));
    },
    // Recalculates the amount.
    recalculateAmount: function() {
        var currency = this.get("UsrCurrency");
        var division = currency ? currency.Division : null;
        MoneyModule.RecalcCurrencyValue.call(this, "UsrCurrencyRate",
"UsrAmount", "UsrPrimaryAmount", division);
    },
    // Recalculates the amount in base currency.
    recalculatePrimaryAmount: function() {
        var currency = this.get("UsrCurrency");
        var division = currency ? currency.Division : null;
        MoneyModule.RecalcBaseValue.call(this, "UsrCurrencyRate",
"UsrAmount", "UsrPrimaryAmount", division);
    },
    // The handler of the currency virtual column change.
    onVirtualCurrencyChange: function() {
        var currency = this.get("Currency");
        this.set("UsrCurrency", currency);
    }
},
// Setting up the visualization of a multi-currency field on the edit
page.
diff: /**SCHEMA_DIFF*/
{
    // Metadata for adding the [Amount] field.
    {
        // Adding operation.
        "operation": "insert",
        // The meta-name of the parent container to which the component
is added.
        "parentName": "Header",
        // The field is added to the collection of the
        // parent container.
        "propertyName": "items",
        // The meta-name of the schema component above which the action
is performed.
        "name": "UsrAmount",
    }
}
```

```

        // Properties passed to the constructor of the component.
        "values": {
            // The name of the column of the view model to which the
            binding is performed.
            "bindTo": "UsrAmount",
            // Element location in the container.
            "layout": { "column": 0, "row": 2, "colSpan": 12 },
            // The name of the column that contains the amount in the
            base currency.
            "primaryAmount": "UsrPrimaryAmount",
            // The name of the column that contains the currency of the
            amount.
            "currency": "UsrCurrency",
            // The name of the column that contains the exchange rate.
            "rate": "UsrCurrencyRate",
            // The property that defines whether the amount field is
            enabled and editable in the base currency.
            "primaryAmountEnabled": false,
            // Generator of the control view.
            "generator": "MultiCurrencyEditViewGenerator.generate"
        }
    }
} /**SCHEMA_DIFF*/
);
});
```

After saving the schema and updating the application page the [Amount] multi-currency field will be displayed on the project edit page (Fig. 1). The value of the field will be automatically recalculated after selecting a currency from the drop-down list (Fig. 9).

Fig. 9. Drop-down list of currencies



How to add custom logic to the existing controls

Beginner

Easy

Medium

Advanced

Introduction

Controls are objects used to create an interface between the user and a Creatio application. For example, buttons, fields, checkboxes, etc.

All controls in Creatio are inherited from the *Terrasoft.controls.Component* class. Full list of classes that implement Creatio components is available by link in the “**[JavaScript API for platform core \(on-line documentation\)](#)**”.

According to the [Open–closed](#) principle, you cannot add custom logic to the existing control. For this you need to create a new class that inherits functions of the existing class of the control. And implement new functions in the successor class.

Steps to add new functions:

1. Create new client module.
2. In the client module, declare a class that inherits the existing control. Implement necessary functions in the class.
3. Add a new item to the Creatio interface.

Case description

Create control which enables to enter only integer values in the specified range. Perform the checking of the entered value by pressing the Enter key and display the corresponding message if the number is outside the range. Use the *Terrasoft.controls.IntegerEdit* control as parent.

Source code

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

Case implementation algorithm

1. Create a client module

The procedure for creating a custom schema is covered in the “**Creating a custom client module schema**”.

Run the [Add] – [Module] menu command on the [Schemas] tab of the [Configuration] section.

Specify following properties of the schema:

- [Name] – “UsrLimitedIntegerEdit”
- [Title] – "UsrLimitedIntegerEdit"

Add the following source code to the schema:

```
// Declaration of the module.  
define("UsrLimitedIntegerEdit", [], function () {  
});
```

2. Create a class of the control

Modify the source code according to the example below.

```
define("UsrLimitedIntegerEdit", [], function () {  
    // Declaration of the class of the control.  
    Ext.define("Terrasoft.controls.UsrLimitedIntegerEdit", {  
        // Base class.  
        extend: "Terrasoft.controls.IntegerEdit",  
        // Alias (abbreviated name of the class)..  
        alternateClassName: "Terrasoft.UsrLimitedIntegerEdit",  
        // The smallest allowed value.  
        minLimit: -1000,  
        // The highest value allowed.  
        maxLimit: 1000,  
        // A method for checking for an occurrence in the range of valid values.  
        isOutOfLimits: function (numericValue) {  
            if (numericValue < this.minLimit || numericValue > this.maxLimit) {  
                return true;  
            }  
            return false;  
        },  
        // Override the method of the event handler for pressing the Enter key.  
        onEnterKeyPressed: function () {  
            // Call the basic functionality.  
            this.callParent(arguments);  
            // Get the entered value.  
            var value = this.getTypedValue();  
            // Reduction to a numeric type.
```

```
        var numericValue = this.parseNumber(value);
        // Check for occurrence in the range of acceptable values.
        var outOfLimits = this.isOutOfLimits(numericValue);
        if (outOfLimits) {
            // Form the warning message.
            var msg = "Value " + numericValue + " is out of limits [" +
this.minLimit + ", " + this.maxLimit + "]";
            // Modify the configuration object to display a warning message.
            this.validationInfo.isValid = false;
            this.validationInfo.invalidMessage = msg;
        }
        else{
            // Modify the configuration object to hide the warning message.
            this.validationInfo.isValid = true;
            this.validationInfo.invalidMessage ="";
        }
        // Call the logic for displaying the warning message.
        this.setMarkOut();
    },
});
```

You can use the logic of the `onEnterKeyPressed()` method in the `onBlur()` event handler.

Save the schema.

Except the *extend* and *alternateClassName* standard properties, the *minLimit* and *maxLimit* properties that specify the range of allowed values are added to the class. Default values are used for these properties.

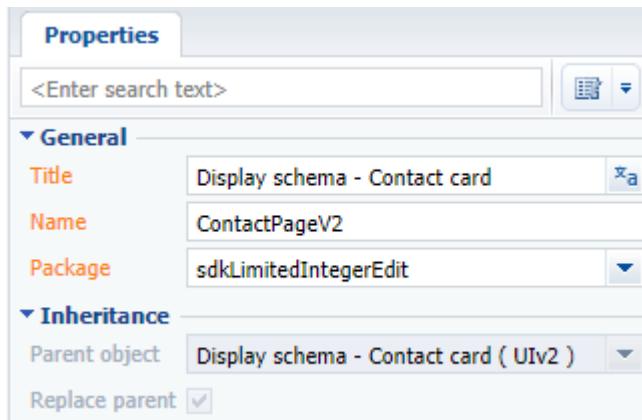
The required control logic is implemented in the `onEnterKeyPressed` override method. After calling the base logic in which the generation of the value change events is performed, the entered value is checked for validity. If the number is not valid, the corresponding warning message is displayed in the input field. The `isOutOfLimits` method is provided to check the occurrence of the entered value in the range of allowed values.

With this implementation, despite the output of the corresponding warning, the entered value is still stored and transferred to the schema view model in which the component will be used.

3. Add the control to the Creatio interface

To add the created control to the Creatio, create the replacing schema, for example, the contact record page. Create a replacing client module and specify the [Display schema – Contact card] (*ContactPageV2*) schema as parent object (Fig. 1). Creating a replacing page is covered in the **“Creating a custom client module schema”** article.

Fig. 1. Properties of the replacing edit page



Add the following source code to the schema:

```
// Declaration of the module. Be sure to specify the dependency  
// of the module in which the class of the control is declared.  
define("ContactPageV2", ["UsrLimitedIntegerEdit"],
```

```

function () {
    return {
        attributes: {
            // Attribute associated with the value in the control.
            "ScoresAttribute": {
                // Attribute data type is integer.
                "dataValueType": this.Terrasoft.DataValueType.INTEGER,
                // Attribute type is a virtual column.
                "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                // The default value.
                "value": 0
            }
        },
        diff: /**SCHEMA_DIFF*/ [
            {
                // The type of operation is the addition.
                "operation": "insert",
                // The name of the container to which the control is added.
                "parentName": "ProfileContainer",
                // The name of the property in the container to which you want to
                add
                // instance of the control.
                "propertyName": "items",
                // The name of the control.
                "name": "Scores",
                // Header.
                "caption": "Scores",
                // Values passed to the properties of the control.
                "values": {
                    // The type of the control is the component.
                    "itemType": Terrasoft.ViewItemType.COMPONENT,
                    // The name of the class.
                    "className": "Terrasoft.UsrLimitedIntegerEdit",
                    // The value property of the component is associated with the
                    ScoresAttribute attribute.
                    "value": { "bindTo": "ScoresAttribute" },
                    // Values for the minLimit property.
                    "minLimit": -300,
                    // Values for the maxLimit property.
                    "maxLimit": 300,
                    // The location of the component in the container.
                    "layout": {
                        "column": 0,
                        "row": 6,
                        "colSpan": 24,
                        "rowSpan": 1
                    }
                }
            }
        ],
        /**SCHEMA_DIFF*/
    };
});
}
);

```

Save the schema.

The added *ScoresAttribute* attribute contains the value connected to the value entered in input field of the control. You can use an integer column of the object connected to the view model of the record edit page instead of the attribute.

The configuration object determining the values of the properties of control entity is added to the *diff* array. The value of the “value” property is connected to the *ScoresAttribute* attribute. The values that specify a valid input range are assigned to the *minLimit* and *maxLimit* properties.

If the `minLimit` and `maxLimit` properties are not explicitly specified in the configuration object, the default range (-1000, 1000) will be applied.

As a result, the integer field will be added to the contact record page (Fig. 2). The warning message will be displayed in the field if the invalid message will be entered (Fig. 3).

Fig. 2. Case result

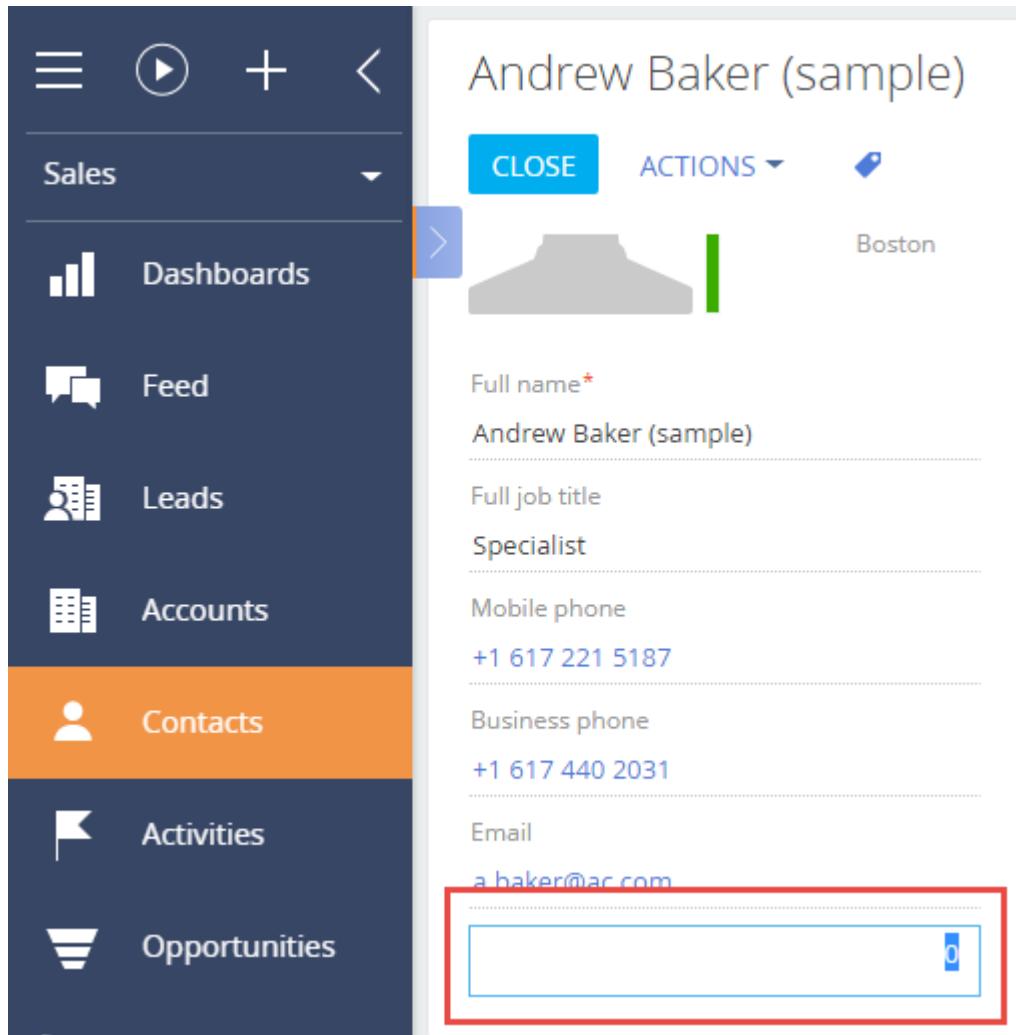
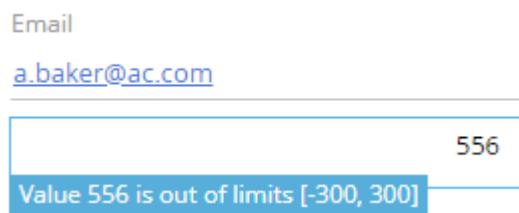


Fig. 3. Displaying of warning message



Adding calculated fields

Beginner Easy **Medium** Advanced

Introduction

A calculated field is a page control element, whose value is generated based on the status and values in other elements on this page.

In Creatio, calculated fields are based on the Creatio client mechanism, which uses subscriptions to change events in view model schema attributes. For any attribute, you can set a configuration object and specify object schema column names. If the values in these columns change, the value of the calculated column will be updated. You can also specify the handler method for this event.

The general sequence of adding a calculated field is as follows:

1. Add a column for storing the values of the calculated field to the page object schema.
2. In the page view model, set up attribute dependencies by specifying column names from which the attribute depends and the handler name.
3. Add the implementation of the handler method to the method collection of the view model.
4. Set up the display of the calculated field in the *diff* property of the view model.

Setting up dependencies of the calculated field

In the *attributes* view model property, add an attribute for which the dependency is configured.

Declare a *dependencies* property, which is an array of configuration objects, each of which contains the following properties:

- *Columns* – an array of columns, whose values determine the value of the current column.
- *methodName* – handler method name.

If the value of at least one of these columns changes in the view model, the event handler method (whose name is specified in the *methodName* property) will be called.

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

Case description

Add [Payment balance] to display the balance between order amount and payment amount on the order edit page.

You can add fields to the edit page manually or using the section wizard. Learn more about adding fields to edit pages in the “**Adding a new field to the edit page**” article.

Source code

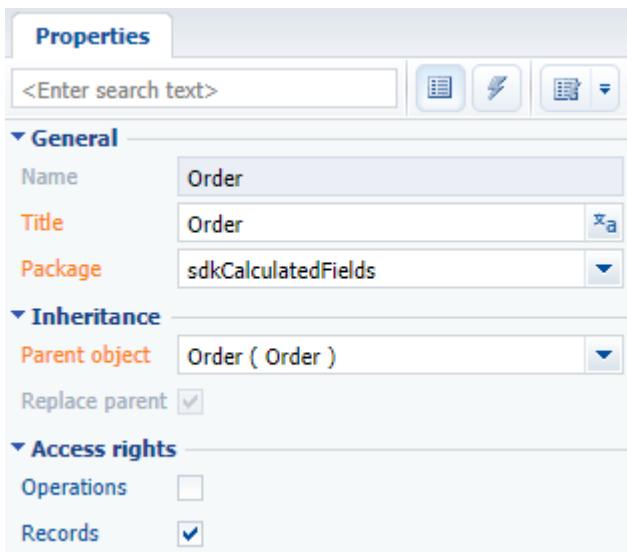
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Create a replacing object

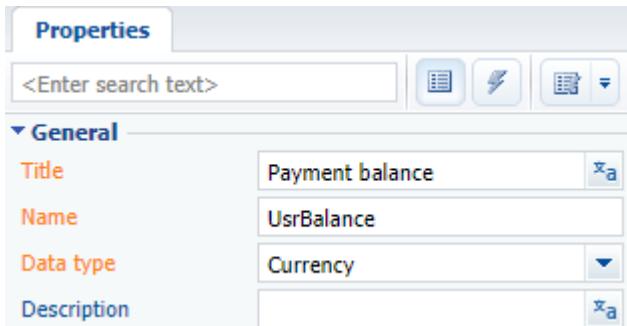
Select the custom package on the [Schemas] tab and click the [Replacing object] in the [Add] menu. Select the [Order] object as the parent object (Fig. 1).

Fig. 1. Properties of the [Order] replacing object



Add a new [Payment balance] column of the [Currency] type to the replacing object (Fig. 2).

Fig. 2. Adding a custom column to the replacing object

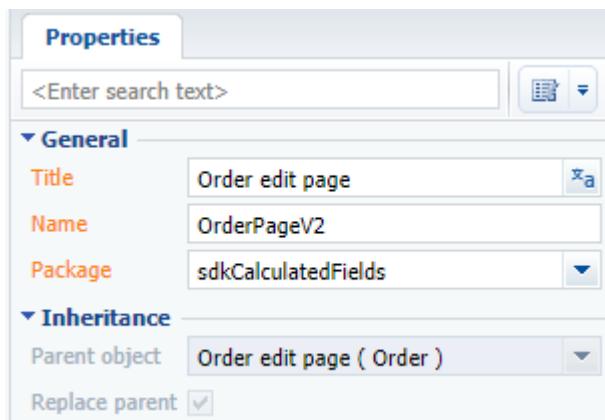


Publish the object.

2. Create a replacing client module for the order edit page

A replacing client module must be created and [Order Edit Page] (*OrderPageV2*) must be specified as the parent object in it (Fig. 3). Creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 3. Properties of the replacing edit page



3. Set up the display of the [Payment balance] field

Describe the configuration object with the required parameters in the *diff* property of the view model. Page schema source code is available below

4. Add the *UsrBalance* attribute to the view model schema

Add the *UsrBalance* attribute to the collection of the *attributes* property in the source code of the page view model. Specify dependency from the [Amount] and [PaymentAmount] columns, as well as the *calculateBalance()* handler method (which will be calculating the value of the [UsrBalance] column) in the configuration object of the *UsrBalance* attribute.

5. Add the needed methods in the *methods* collection of the view model

In the *methods* collection of the view model, add the *calculateBalance()* handler method that will handle the editing of the [Amount] and [PaymentAmount] columns. This method is used in the *UsrBalance* attribute.

Make sure you take into consideration the type of data in the handler method that need to be returned in the calculation field as a result. For example, the [Decimal (0.01)] data type assumes a number with two decimals. Before recording the result in the object field, convert it via the *toFixed()* function. The example code in this case could look as follows:

```
// The calculation of difference between the [Amount] and [PaymentAmount] column values.  
var result = amount - paymentAmount;  
// The calculation result is assigned as a value to the [UsrBalance] column.  
this.set("UsrBalance", result.toFixed(2));
```

Override the *onEntityInitialized()* base virtual method. The *onEntityInitialized()* method is triggered after the edit page object schema is initialized. Calling the *calculateBalance()* handler method to this method will ensure the calculation of the amount to be paid at the moment the order page opens and not only when the dependency columns are edited.

The complete source code of the module is available below:

```
define("OrderPageV2", [], function() {  
    return {  
        // Edit page object schema name.  
        entitySchemaName: "Order",  
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,  
        // The attributes property of the view model.  
        attributes: {  
            // Name of the view model attribute.  
            "UsrBalance": {  
                // Data type of the view model column.  
                dataValueType: Terrasoft.DataValueType.FLOAT,  
                // Array of configuration objects that determines [UsrBalance] column  
                dependencies: [  
                    {  
                        // The value in the [UsrBalance] column depends on the  
                        [Amount]  
                        // and [PaymentAmount] columns.  
                        columns: ["Amount", "PaymentAmount"],  
                        // Handler method, which is called on modifying the value of  
                        the on of the columns: [Amount]  
                        // and [PaymentAmount].  
                        // methodName: "calculateBalance"  
                    }  
                ]  
            },  
            // Collection of the edit page view model methods.  
            methods: {  
                // Overriding the base Terrasoft.BasePageV2.onEntityInitialized method,  
                which  
                // is triggered after the edit page object schema has been initialized.  
                onEntityInitialized: function() {  
                    // Method parent implementation is called.  
                    this.callParent(arguments);  
                }  
            }  
        }  
    }  
},  
// Collection of the edit page view model methods.  
methods: {  
    // Overriding the base Terrasoft.BasePageV2.onEntityInitialized method,  
    which  
    // is triggered after the edit page object schema has been initialized.  
    onEntityInitialized: function() {  
        // Method parent implementation is called.  
        this.callParent(arguments);  
    }  
}
```

```

        // Calling the handler method, which calculates the value in the
[UsrBalance] column.
        this.calculateBalance();
    },
    // Handler method that calculates the value in the [UsrBalance] column.
    calculateBalance: function() {
        // Checking whether the [Amount] and [PaymentAmount] columns are
initialized
        // when the edit page is opened. If not, then zero values are set for
them.
        var amount = this.get("Amount");
        if (!amount) {
            amount = 0;
        }
        var paymentAmount = this.get("PaymentAmount");
        if (!paymentAmount) {
            paymentAmount = 0;
        }
        // Calculating the margin between the values in the [Amount] and
[PaymentAmount] columns.
        var result = amount - paymentAmount;
        // The calculation result is set as the value in the [UsrBalance]
column.
        this.set("UsrBalance", result);
    }
},
// Visual display of the [UsrBalance] column on the edit page.
diff: /**SCHEMA_DIFF*/ [
{
    "operation": "insert",
    "parentName": "Header",
    "propertyName": "items",
    "name": "UsrBalance",
    "values": {
        "bindTo": "UsrBalance",
        "layout": {"column": 12, "row": 2, "colSpan": 12}
    }
}
] /**SCHEMA_DIFF*/
};

}) ;
});

```

After saving the schema, updating the web page and clearing the cache, a new [Payment balance] will appear on the order page. The value in this field will be calculated based on the values in the [Total] and [Payment amount] fields (Fig. 4).

Fig. 4. Case result demonstration

The screenshot shows a web-based form titled "ORD-1 (sample)". At the top, there are buttons for "SAVE", "CANCEL", "ACTIONS ▾", "PRINT ▾", and "VIEW ▾". The main area contains the following data:

Customer *	Accom (sample)	Total, \$	3,492.00
Status	1. Draft	Payment amount, \$	0.00
		Payment balance	3,492.00

How to set a default value for a field

Beginner

Easy

Medium

Advanced

Introduction

In Creatio, you can define the default values for edit page control elements.

You can set a default value in two ways:

1. Set the value on the business object column level. When creating a new object, its certain page fields should be populated with some initially known values. In such cases, indicate these values for the corresponding object columns as the default values in object designer.

Types of default values.

Name	Description
<i>Set constant</i>	String, number, lookup value, Boolean.
<i>Set from system setting</i>	The complete list of system settings is available in the [System settings] section. It can be supplemented with custom system settings.
<i>Set from system variable</i>	Creatio system variables are global variables that store information about system-wide setting values. Unlike system settings, whose values can differ depending on different users, system variable values always remain the same for all users. The full list of system variables is implemented on the kernel level and cannot be changed by user: <ul style="list-style-type: none">• New identifier• New sequential identifier• Current user• Contact of the current user• Account of the current user• Current date and time value• Current date value• Current time value

Default value not set

2. Specify in the edit page source code. In some cases, it is impossible to set a default value via the object column properties. For example, these can be estimated values which are calculated by other column values of the object, etc. In such a case, you can set a default value only via programming means.

Source code

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

Example of setting a field default value via object column properties

Case description

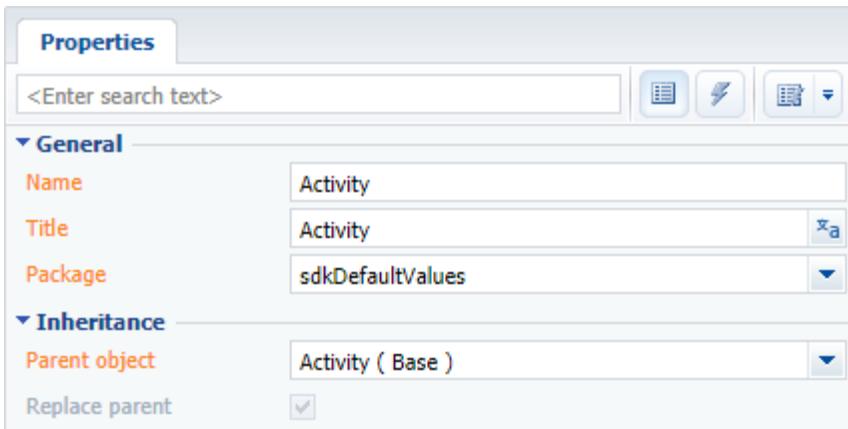
When creating a new activity, the [Show in calendar] checkbox should be set by default.

Case implementation algorithm

1. Creating the [Activity] replacing object in the custom package

Create the [Activity] replacing object (Fig.1). Learn more about creating a replacing object in the “[Creating the entity schema](#)” article.

Fig. 1. The [Activity] replacing object properties



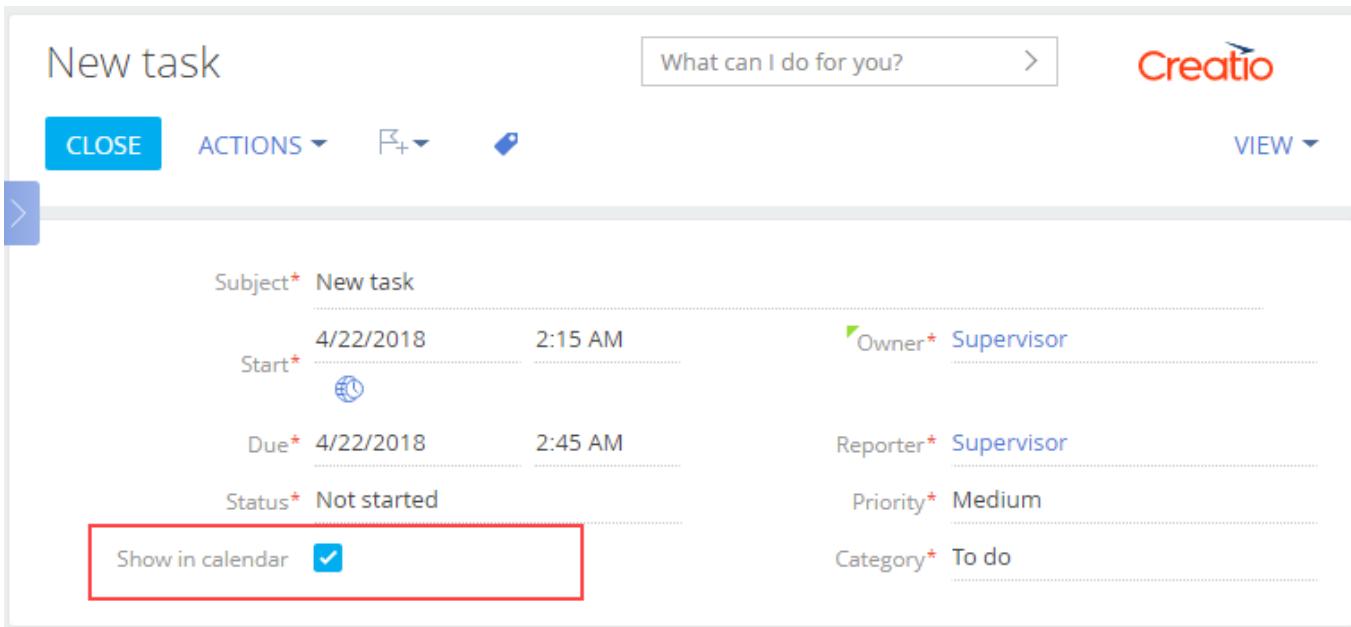
2. Setting the default value for the [Show in calendar] column

Select the [Show in calendar] column from the inherited column list and edit its [Default value] property as shown in fig.2. To implement the case, select a constant value as the default one.

Fig. 2. Setting the default value for the [Show in calendar] column

After you publish the schema, update the page and clear the cache. The [Show in calendar] field will be selected on the activity edit page when adding a new activity.

Fig. 3. Demonstration of setting the default value



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

Case description

The default value in the [Deadline] field on the project edit page should be as follows: the [Start] field value plus 10 days.

Case implementation algorithm

1. Create a project replacing edit page in custom package

Create a replacing client module and specify the [Project edit page], *ProjectPageV2* schema as its parent object (Fig. 4). The procedure of creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 4. Replacing edit page properties

The 'Properties' dialog shows the following settings:

- General** section:
 - Title: Project edit page
 - Name: ProjectPageV2
 - Package: sdkDefaultValues
- Inheritance** section:
 - Parent object: Project edit page (Project)
 - Replace parent:

2. Add the implementation of the following methods to the method collection of the page view model

- *setDeadline()* – handler method. Calculates the [Deadline] field value.
- *onEntityInitialized()* – an overridden base virtual method. Triggered upon termination of object schema initialization. Add the handler method call to set the [Deadline] field value to it when opening the edit page.

The replacing schema source code is as follows:

```
define("ProjectPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Project",
        methods: {
            // Overriding the Terrasoft.BasePageV2.onEntityInitialized() base method.
            // Triggered upon termination of edit page object schema initialization.
            onEntityInitialized: function() {
                // Calling of method parent implementation.
                this.callParent(arguments);
                // Calling of handler method that calculates the [Deadline] field
                value.
                this.setDeadline();
            },
            // Handler method. Calculates the [Deadline] field value.
            setDeadline: function() {
                // The [Deadline] column value.
                var deadline = this.get("Deadline");
                // Is a new record mode set?
                var newmode = this.isNewMode();
                // If the value is not set and the new record mode is set.
                if (!deadline && newmode) {
                    // Receipt of the [Start] column value.
                    var newDate = new Date(this.get("StartDate"));
                    newDate.setDate(newDate.getDate() + 10);
                    // Setting of the [Deadline] column value.
                    this.set("Deadline", newDate);
                }
            }
        }
    };
});
```

Save the schema and update the application page. A date that equals the [Start] field date plus 10 days will be set in the [Deadline] field (Fig.5).

Fig. 5. Demonstration of setting the calculated default value

The screenshot shows a 'New record' screen in the Creatio application. At the top, there are buttons for 'SAVE', 'CANCEL', 'ACTIONS', and 'VIEW'. The main area contains fields for 'Name*', 'Status*' (Planned), 'Owner*' (Supervisor), and 'Amount'. Below these are several tabs: 'GENERAL INFORMATION' (which is selected and highlighted in orange), 'STRUCTURE', 'FINANCIAL INDICATORS', 'HISTORY', and 'ATTACHMENTS AND NOTES'. Under the 'GENERAL INFORMATION' tab, there are sections for 'Account' and 'Contact'. In the 'Account' section, there is a 'Completion %' field with a 'Calculate automatically' checkbox and a 'Start' date field set to '4/22/2018', which is enclosed in a red box. In the 'Contact' section, there is a 'Type*' field and a 'Duration' field set to '0 min', followed by a 'Deadline' field set to '5/2/2018', also enclosed in a red box.

How to add the field validation

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

Introduction

Validation is the verification of field values for their compliance with certain requirements. Values of the Creatio page fields are validated at the level of the page view model columns. The logic of the field value validation is implemented in the custom validation method.

Validator is the method of the view model where values of the view model column are analyzed for compliance with business requirements. This method must return validation results as an object with the following property:

- *invalidMessage* – a message string displayed under the field when making an attempt to save a page with an invalid field value and in the data window when saving a page with the field that did not pass validation.

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

To start the field validation, the corresponding view model column must be bound to a specific validator. For this purpose, override the *setValidationConfig()* base method and call the *addColumnValidator()* method in it.

The *addColumnValidator()* method accepts two parameters:

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

If the field is validated in the replacement client schema of the base page, the parent implementation of the *setValidationConfig()* method must be called before calling the *addColumnValidator()* method to correctly initialize validators of the base page fields.

To add validation of field values:

1. Add the validator method to the collection of methods of the view model that will check a field value.
2. Override the *setValidationConfig()* method and connect the validator to the corresponding view model column in it.

Source code

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

Example 1

Case description

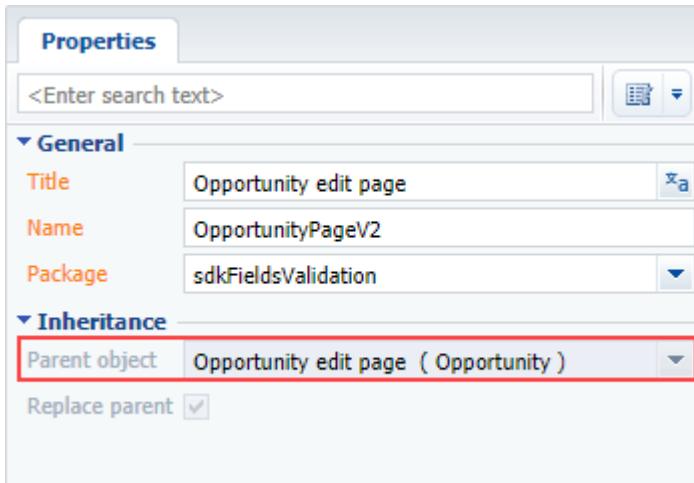
Set the validation on the opportunity page as follows: the date in the [Created on] field must be earlier than the date in the [Closed on] field.

Case implementation algorithm

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

A replacing client module must be created and *[OpportunityPageV2]* must be specified as the parent object in it (Fig. 1). Creating a replacing page is covered in the “**Creating a custom client module schema**” article.

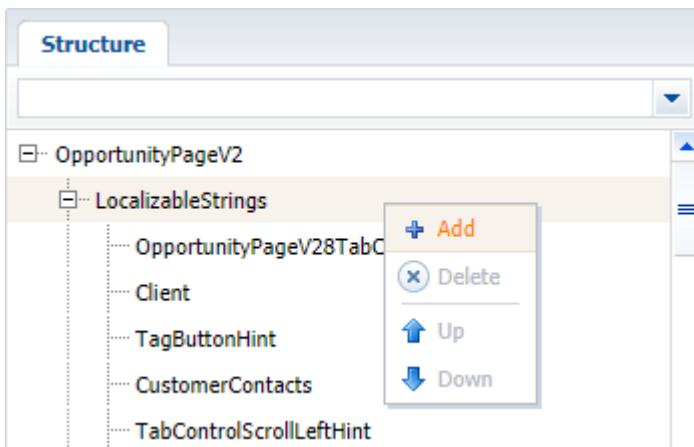
Fig. 1. Properties of the replacing edit page



2. Add an error string to the collection of localizable strings of the page replacing schema

Create a new localizable string (Fig. 2).

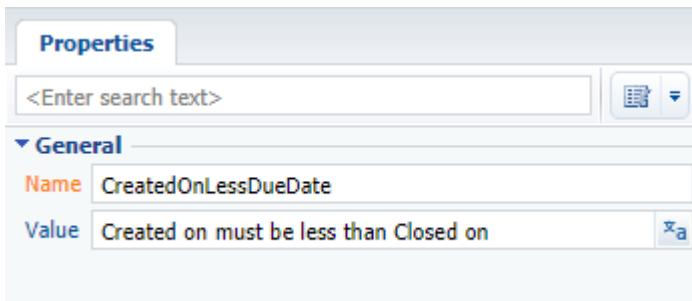
Fig. 2. Adding localized string to the schema



For the created string, specify (Fig. 3):

- [Name] – “CreatedOnLessDueDate”.
- [Value] – “Created on must be less than Closed on”.

Fig. 3. Properties of the custom localizable string



3. Add the implementation of methods in the *methods* collection of the view model

- *dueDateValidator()* – validator method that determines if the condition is fulfilled.
- *setValidationConfig()* – an overridden base method in which the validator method is bound to the [DueDate] and [CreatedOn] columns.

The replacing schema source code is as follows:

```
define("OpportunityPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Opportunity",
        methods: {
            // Validate method for values in the [DueDate] and [CreatedOn] columns.
            dueDateValidator: function() {
                // Variable for storing a validation error message.
                var invalidMessage = "";
                // Checking values in the [DueDate] and [CreatedOn] columns.
                if (this.get("DueDate") < this.get("CreatedOn")) {
                    // If the value of the [DueDate] column is less than the value
                    // of the [CreatedOn] column a value of the localizable string
                    // is
                    // placed into the variable along with the validation error
                    // message
                    // in the invalidMessage variable.
                    invalidMessage =
                        this.get("Resources.Strings.CreatedOnLessDueDate");
                }
                // Object whose properties contain validation error messages.
                // If the validation is successful, empty strings are returned to
                // the
                // object.
                return {
                    // Validation error message.
                    invalidMessage: invalidMessage
                };
            },
            // Redefining the base method initiating custom validators.
            setValidationConfig: function() {
                // This calls the initialization of validators for the parent view
                // model.
                this.callParent(arguments);
                // The dueDateValidator() validate method is added for the [DueDate]
                // column.
                this.addColumnValidator("DueDate", this.dueDateValidator);
                // The dueDateValidator() validate method is added for the
                // [CreatedOn] column.
                this.addColumnValidator("CreatedOn", this.dueDateValidator);
            }
        }
    }
});
```

```

        }
    } ;
}) ;

```

After you save the schema and refresh Creatio page, a string with the corresponding message (Fig. 4) will appear on the opportunity edit page when entering the date of closing or date of creation which does not satisfy the validation condition (the date of creation must be before than the date of closing). The data window will appear when making an attempt to save the opportunity (Fig. 5).

Fig. 4. Case results: invalid date message

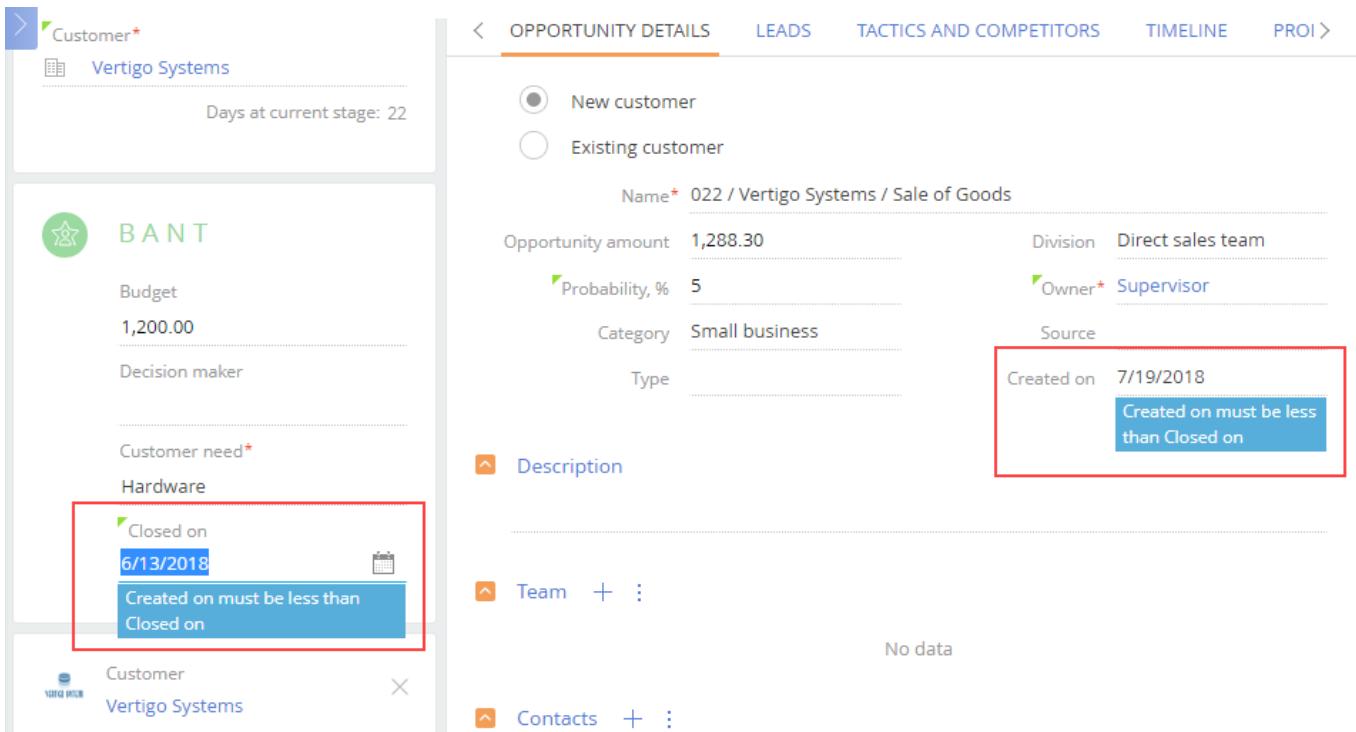
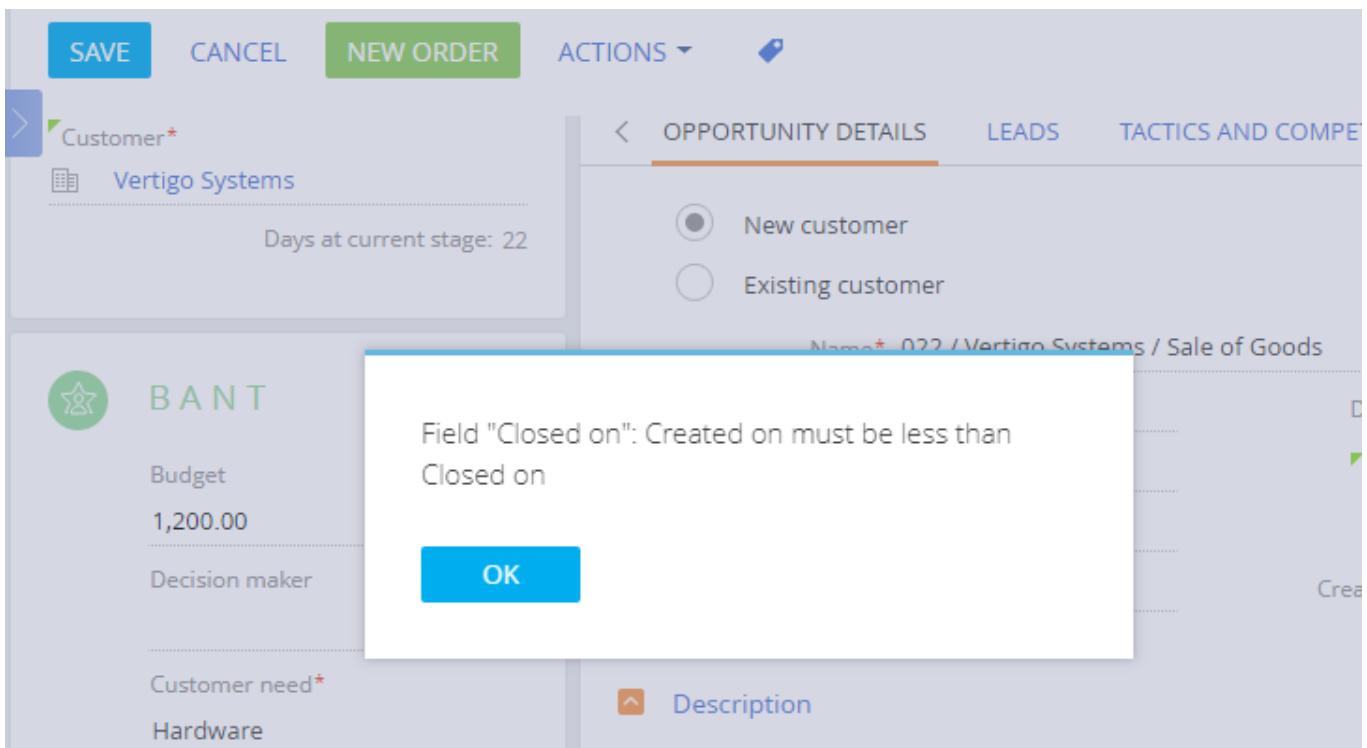


Fig. 5. Case results: message when saving



Example 2

Case description

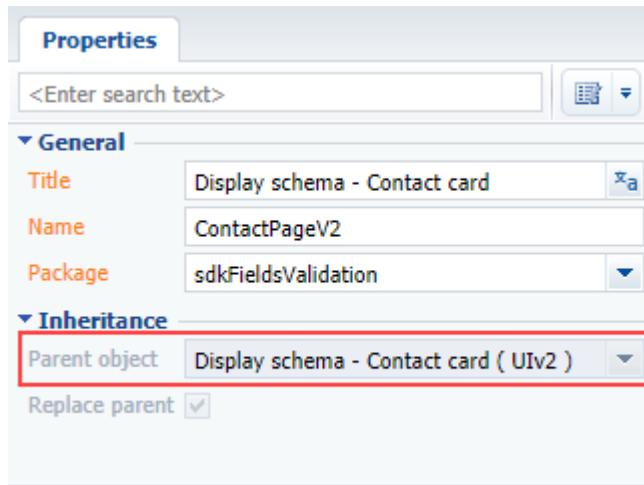
Set the [Business phone] field validation as follows: phone number must correspond to the following mask: +44 XXX XXX XXXX, otherwise the “Enter the number in the “+44 XXX XXX XXXX” format” message appears.

Case implementation algorithm

1. Create a replacing client module

Create a replacing client module and specify the [Display schema – Contact card] (*ContactPageV2*) schema as parent object (Fig. 6). Creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 6. Properties of the replacing edit page



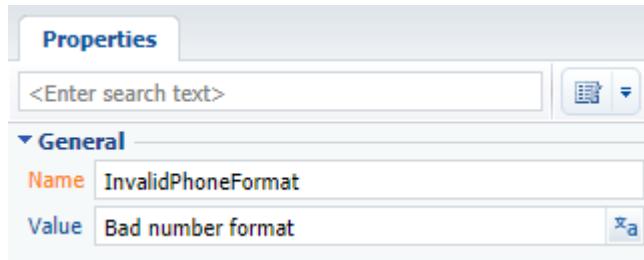
2. Add an error string to the collection of localizable strings of the page replacing schema

Create a new localizable string (Fig. 2).

For the created string, specify (Fig. 7):

- [Name – "InvalidPhoneFormat"]
- [Value] – “Enter the number in the “+44 XXX XXX XXXX” format”.

Fig. 7. Properties of the custom localizable string



3. Add the implementation of methods in the *methods* collection of the view model

- *phoneValidator()* – validator method that determines if the condition is fulfilled.
- *setValidationConfig()* – an overridden base method in which the validator method is bound to the [Phone] column.

The replacing schema source code is as follows:

```
define("ContactPageV2", ["ConfigurationConstants"], function(ConfigurationConstants)
```

```
{  
    return {  
        entitySchemaName: "Contact",  
        methods: {  
            // Redefining the base method initiating custom validators.  
            setValidationConfig: function() {  
                // Calls the initialization of validators for the parent view model.  
                this.callParent(arguments);  
                // The phoneValidator() validate method is added to the [Phone]  
                column.  
                this.addColumnValidator("Phone", this.phoneValidator);  
            },  
            phoneValidator: function(value) {  
                // Variable for stroing a validation error message.  
                var invalidMessage = "";  
                // Variable for stroing the number check result.  
                var isValid = true;  
                // Variable for the phone number.  
                var number = value || this.get("Phone");  
                // Determining the correctness of the number format using a regular  
                expression.  
                isValid = (Ext.isEmpty(number) ||  
                    new RegExp("^\+\+44\s[0-9]{3}\s[0-9]{3}\s[0-9]  
{4}$").test(number));  
                // If the format of the number is incorrect, then an error message is  
                filled in.  
                if (!isValid) {  
                    invalidMessage =  
this.get("Resources.Strings.InvalidPhoneFormat");  
                }  
                // Object which properties contain validation error messages.  
                // If the validation is successful, empty strings are returned to the  
                object.  
                return {  
                    invalidMessage: invalidMessage  
                };  
            }  
        };  
    };  
});
```

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

Fig. 8. Case results: Message about the incorrect format

The screenshot shows a contact edit page. At the top right, there is a green progress bar labeled '95%' and a timestamp '7:24 AM, Perth'. The contact's name 'Jane Russel' and job title 'Managing Partner' are listed. Below these, fields for 'Mobile phone' (+44 (0) 121 414 6351) and 'Business phone' (+44 1922 158457) are shown. A blue callout box highlights the 'Business phone' field with the instruction 'Enter the number in format +44 XXX XXX XXXX'. An email address 'russel@np.com' is also present.

Fig. 9. Case results: message when saving

The screenshot shows a contact edit page with a modal dialog box in the center. The dialog contains the message 'Field "Business phone": Enter the number in format +44 XXX XXX XXXX' and a blue 'OK' button. The background shows the contact details and a timeline tab.

Using filtration for lookup fields. Examples

Beginner Easy **Medium** Advanced

Introduction

There are two methods of using the filtration in Creatio for lookup fields of the edit page:

1. The [FILTRATION] business rule.
2. Explicit indication of filters in the column description of the attributes model property.

The use of the [FILTRATION] business rule is expedient if a simple filter by a specific value or attribute must be used for the field. The business rules are detailed in the **Setting the edit page fields using business rules**. The detailed case for using the [FILTRATION] business rule is set forth in the **The FILTRATION rule use case** article.

If arbitrary filtration (sorting and addition of supplementary columns to a query when a drop-down list is displayed) is required, the explicit description should be used in the *attributes* model property.

Setting lookup field filters in the *attributes* model property:

1. The name of the column for which filters are set must be added to the *attributes* property of the view model.
2. The *lookupListConfig* property must be declared for this column. It represents a configuration item containing the following properties (not required):
 - *columns* – an array of column names to be added to a request in addition to Id and the primary display column.
 - *orders* – an array of configuration objects determining the data sorting when displayed.
 - *filter* – the method for returning the object of the *Terrasoft.BaseFilter* class or its inheritor, will be applied, in turn, to a request.
 - *or filters* – an array of filters (methods for returning collections of the *Terrasoft.FilterGroup* class).

Filters are added to a collection using the *add()* method which has the following parameters:

Name	Data type	Description
<i>key</i>	String	key
<i>item</i>	Mixed	Element.
<i>index</i>	Number	Index for insert. If not entered, the index to be inserted is not rated.

The object of the *Terrasoft.BaseFilter* class or its inheritor is the *item* parameter. The methods for creating filters with descriptions are given in Table 1 of the **The EntitySchemaQuery class. Filters handling** article.

Filters are combined by default in the collection using the AND logic operator. If the OR operator is to be used, this must be indicated explicitly in the *logicalOperation* property of the *Terrasoft.FilterGroup* object.

Case description

When a value is added to the [Owner] field of the account edit page, display only those contact lookup values for which the following conditions are fulfilled:

- a system user associated with this contact is available
- this user is active.

Source code

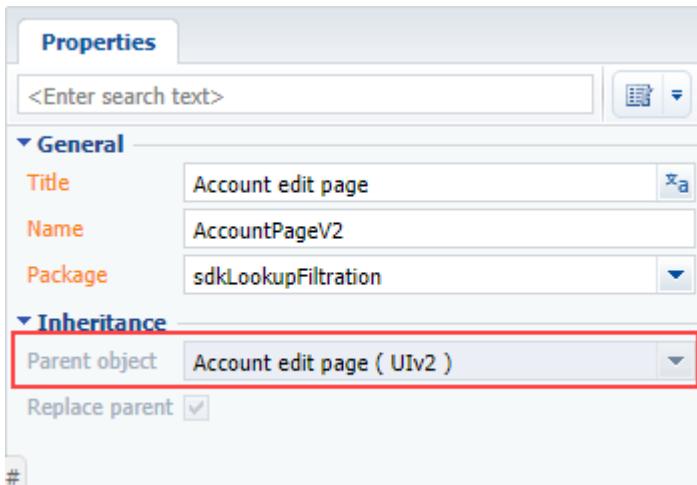
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Create a replacing account edit page

A replacing client module must be created and *[AccountPageV2]* must be specified as the parent object in it (Fig. 1). The procedure of creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 1. Properties of the replacing edit page



2. Add the attribute with filtration to the attributes property of the view model

Specify the type of column data – the *Terrasoft.DataValueType.LOOKUP* in the configuration object, in the [Owner] attribute and describe the configuration object of the *lookupListConfig* lookup field. Add the filters property to *lookupListConfig*, which represents the function for returning the *filters* collection. Add a function that returns a collection of filters to the array.

The replacing schema source code is as follows:

```
define("AccountPageV2", [], function() {
    return {
        // Name of the edit page object schema
        "entitySchemaName": "Account",
        // List of the schema attributes.
        "attributes": {
            // Name of the view model column.
            "Owner": {
                // Attribute data type.
                "dataValueType": Terrasoft.DataValueType.LOOKUP,
                // The configuration object of the LOOKUP type.
                "lookupListConfig": {
                    // Array of filters used for the query that forms the lookup
                    // field data.
                    "filters": [
                        function() {
                            var filterGroup = Ext.create("Terrasoft.FilterGroup");
                            // Adding the "IsUser" filter to the resulting filters
                            // collection.
                            // The filter provides for the selection of all records
                            // in the Contact core schema
                            // to which the Id column from the SysAdminUnit schema is
                            // connected, for which
                            // Id is not equal to null.
                            filterGroup.add("IsUser",
                                Terrasoft.createColumnIsNotNullFilter([
                                    [SysAdminUnit:Contact].Id]));
                            // Adding the "IsActive" filter to the resultant filters collection.
                            // The filter provides for the selection of all records
                            // from the core schema.
                            // Contact to which the Id column from the SysAdminUnit
                            // schema, for which
                            // Active=true, is connected.
                            filterGroup.add("IsActive",
                                Terrasoft.createColumnFilterWithParameter(

```

```
        Terrasoft.ComparisonType.EQUAL,
        "[SysAdminUnit>Contact].Active",
        true));
    return filterGroup;
}
]
}
}
);
})
});
```

When the schema is saved and the system web-page is updated, only values from the contact lookup which comply with custom conditions will be displayed on the account edit page when adding a value to the [Owner] field on the account edit page (Fig. 2, Fig. 3). I.e:

- a system user associated with this contact is available
- this user is active.

Fig. 2. Account profile with the owner

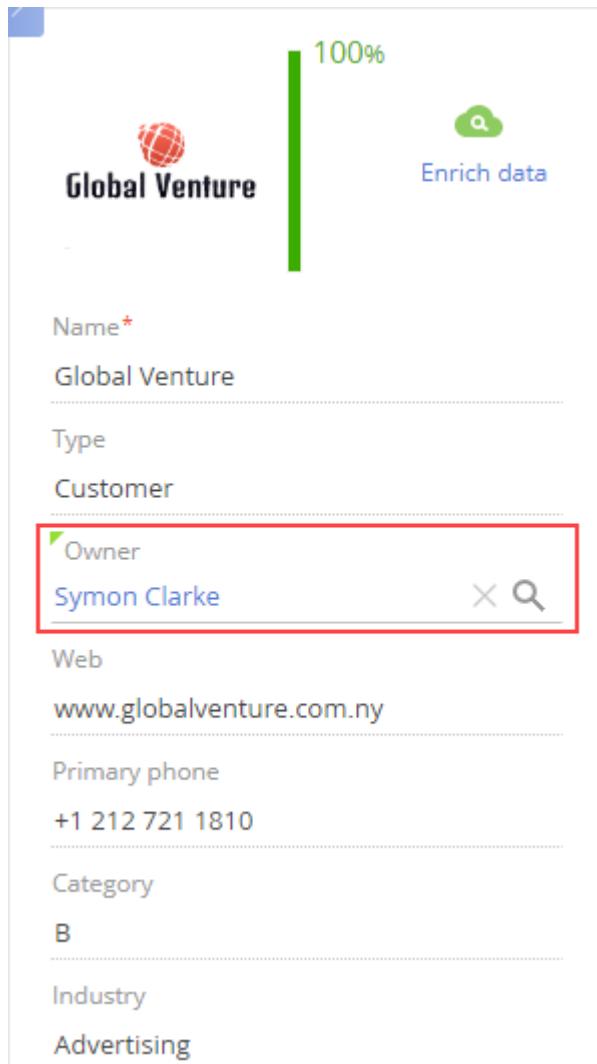


Fig. 3. The owner is disable in the filtered contact lookup

Select: Contact

The screenshot shows a search interface for contacts. At the top, there are buttons for 'SELECT', 'CANCEL', 'NEW', 'ACTIONS ▾', and 'VIEW ▾'. Below these are two input fields: 'Full name' containing 'Symon Clarke' and a dropdown arrow, and another field also containing 'Symon Clarke'. To the right of these is a blue 'SEARCH' button. Below the search area, the text 'No data' is displayed.

Adding an action panel

Beginner Easy **Medium** Advanced

Introduction

Starting with version 7.8.0, Creatio has a new edit page module – the "Action panel" (ActionsDashboard). An action panel displays information about the current status of and actions for working with the current record.

For more information about action panel, please see the "**Action dashboard**" article.

General procedure of adding an action panel on a page:

1. Create a Schema of the Edit Page View Model inherited from the *SectionActionsDashboard* module.
2. Create a replacing page schema.
3. Set up the module in the *modules* property of the page view model.
4. In the "diff" array of the page view model, add the module on the page.

Case description

Add an action panel to the order edit page.

Source code

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

Case implementation algorithm

1. Create a client schema of the OrderActionsDashboard view model

Specify the *SectionActionsDashboard* schema as a parent object (Fig. 1).

Fig. 1. Properties of the client schema

The screenshot shows the 'Properties' dialog for a client schema. The 'General' section includes fields for 'Title' (OrderActionsDashboard), 'Name' (UsrOrderActionsDashboard), and 'Package' (sdkActionsDashboardAdding). The 'Inheritance' section shows 'Parent object' set to 'SectionActionsDashboard (ActionsDashboard)'. A red box highlights this selection. There is also a 'Replace parent' checkbox at the bottom.

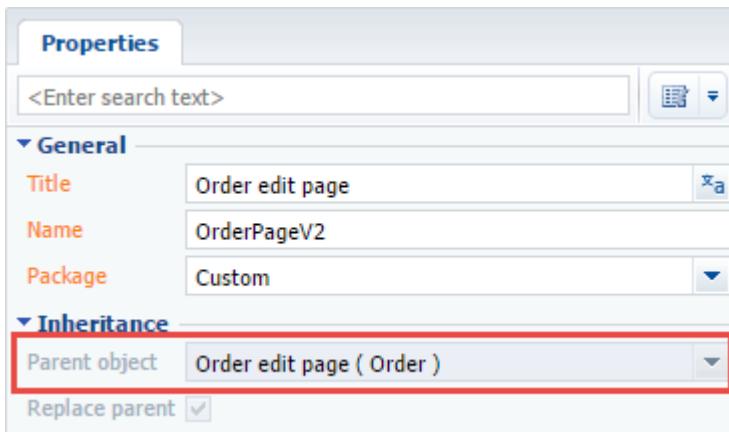
The client schema source code is as follows:

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

2. Create a replacing order edit page

A replacing client module must be created and [Order edit page] (*OrderPageV2*) must be specified as the parent object in it (Fig. 2). Creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 2. Properties of the replacing edit page



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

Add the code of the page replacing module to the [Source code] tab: Add a configuration object with the module settings in it to the *modules* collection of the view model.

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

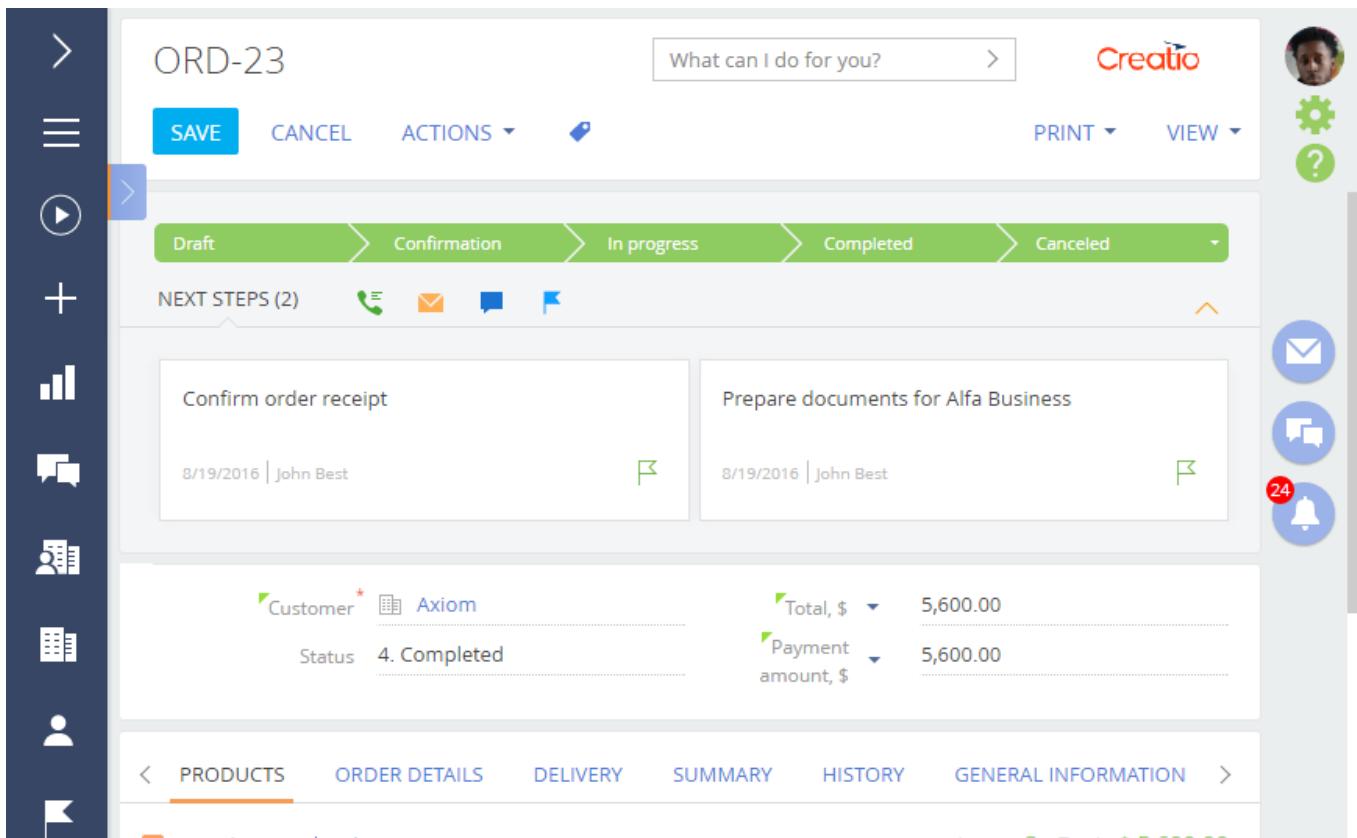
The replacing schema source code is as follows:

```
define("OrderPageV2", [],
    function () {
        return {
            entitySchemaName: "Order",
            attributes: {},
            modules: /**SCHEMA_MODULES*/{
                "ActionsDashboardModule": {
                    "config": {
                        "isSchemaConfigInitialized": true,
                        // Schema name.
                        "schemaName": "UsrOrderActionsDashboard",
                        "useHistoryState": false,
                        "parameters": {
                            // Configuration object of the view model.
                            "viewModelConfig": {
                                // Schema name of the page entity.
                                "entitySchemaName": "Order",
                                // Configuration object of the Actions block.
                                "actionsConfig": {

```

After saving the schema and updating the Creatio web page, the action panel will be added to the order page. The action panel will contain the order status and connected uncompleted activities (Fig. 3).

Fig. 3. Demonstrating the case implementation result



Adding a new channel to the action panel

Beginner **Easy** **Medium** **Advanced**

Introduction

Starting with version 7.8.0, Creatio has a new edit page module – the "Action panel" (ActionsDashboard). An action panel displays information about the current status of and actions for working with the current record.

For more information about action panel, please see the "[Action dashboard](#)" article. The ways of adding the action panel to the section page are described in the "[Adding an action panel](#)" article.

ActionsDashboard channels are a way of communicating with a contact. A channel is created for every section in which it's connected to, for example, a case, contact or lead.

Case description

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

Source code

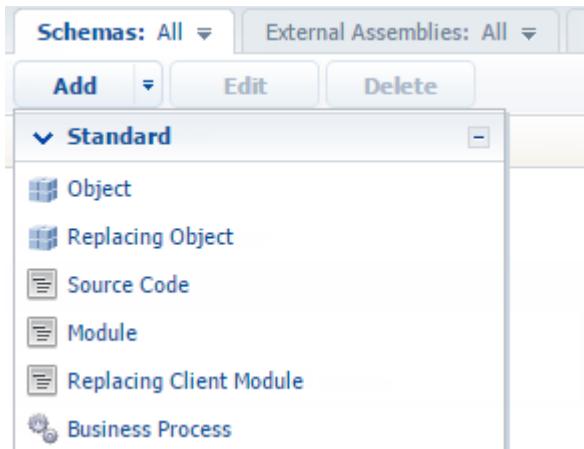
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Add the *UsrCallsMessagePublisher* source code schema

Perform the [Add] > [Source code] menu command on the [Schema] tab in the [Configuration] section (Fig. 1).

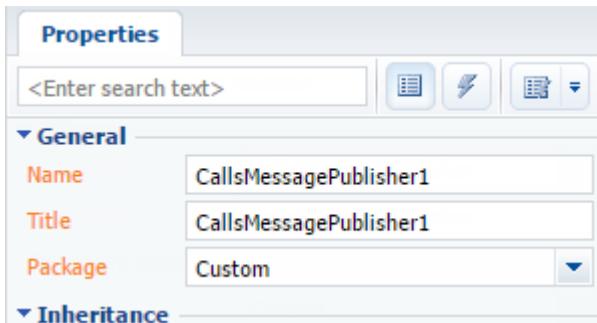
Fig. 1. Adding source code schema



For the created schema specify (Fig. 2):

- [Title] – "Call message logging publisher"
- [Name] – "UsrCallsMessagePublisher"

Fig. 2. The Source code schema properties



In the created schema, add the new *CallsMessagePublisher* class inherited from the *BaseMessagePublisher* class to the *Terrasoft.Configuration* namespace. The *BaseMessagePublisher* class contains the basic logic to save an object in the database and the basic logic of event handlers. The inheritor class will contain the logic for a particular sender, for example, filling of columns of the *Activity* object and the subsequent sending of the message.

To implement the new *CallsMessagePublisher* class, you must add the following source code in the created schema.

```
using System.Collections.Generic;
using Terrasoft.Core;

namespace Terrasoft.Configuration
{
    // The BaseMessagePublisher heir class.
    public class CallsMessagePublisher : BaseMessagePublisher
    {
        // Class constructor.
        public CallsMessagePublisher(UserConnection userConnection,
Dictionary<string, string> entityFieldsData)
            : base(userConnection, entityFieldsData) {
            //The schema the CallsMessagePublisher works with.
            EntitySchemaName = "Activity";
        }
    }
}
```

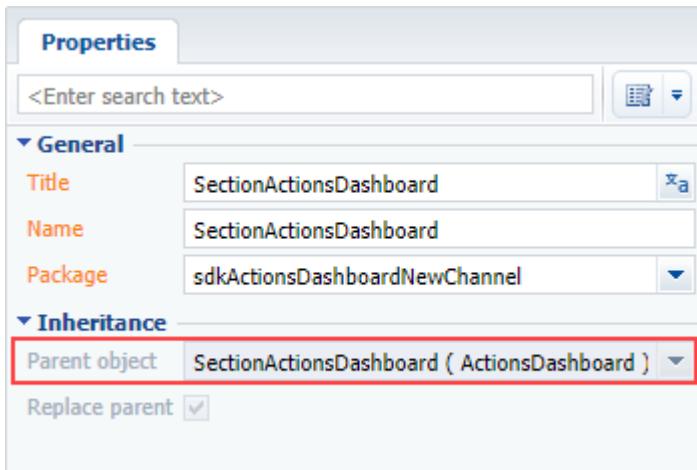
Save and publish the schema.

2. Create the SectionActionsDashboard replacing client schema

Create a replacing client module and specify the *SectionActionsDashboard* as parent object (Fig. 3). The procedure

of creating a replacing page is covered in the "[Creating a custom client module schema](#)" article.

Fig. 3. Properties of the replacing schema



If you want to add a channel to only one edit page, you must create a new module named `[section_name]SectionActionsDashboard` (e.g. `BooksSectionActionsDashboard`) and set `SectionActionsDashboard` as the parent schema.

Specify the module which should be rendered in this channel on one of the tabs in the replacing schema `diff` property. Set the operations of inserting the `CallsMessageTab` tab and message container in this property. The new channel will be visible on the edit pages of those sections, which are connected to `SectionActionsDashboard`.

In the `methods` property override the `getSectionPublishers()` method that will add the new channel to the list of message publishers, and the `getExtendedConfig()` method, in which the tab settings are configured.

For the `getExtendedConfig()` method to run correctly, you must upload the channel icon and specify it in the `ImageSrc` parameter. The icons used in this example can be downloaded [here](#) ('CallsMessageTabImage.svg' in the **on-line documentation**).

You should also override the `onGetRecordInfoForPublisher()` method and add the `getContactEntityParameterValue()` method that defines the contact value from the edit page.

The replacing schema source code is as follows:

```
define("SectionActionsDashboard", ["SectionActionsDashboardResources",
"UsrCallsMessagePublisherModule"],
  function(resources) {
    return {
      attributes: {},
      messages: {},
      methods: {
        // Method sets the channel tab display settings in the action
        dashboard: {
          getExtendedConfig: function() {
            // Parent method calling.
            var config = this.callParent(arguments);
            var lcziImages = resources.localizableImages;
            config.CallsMessageTab = {
              // Tab image.
              "ImageSrc": "ImageSrc": this.Terrasoft.ImageUrlBuilder.getUrl(lcziImages.CallsMessageTabImage),
              // Marker value.
              "MarkerValue": "calls-message-tab",
              // Alignment.
              "Align": this.Terrasoft.Align.RIGHT,
              // Tag.
              "Tag": "UsrCalls"
            };
            return config;
          }
        }
      }
    }
  }
);
```

```
        },
        // Redefines the parent object and adds the contact value from the
edit page
        // of the section that contains the action dashboard.
onGetRecordInfoForPublisher: function() {
    var info = this.callParent(arguments);
    info.additionalInfo.contact =
this.getContactEntityParameterValue(info.relationSchemaName);
    return info;
},
// Defines the contact value from the section edit page
// that contains the action dashboard.
getContactEntityParameterValue: function(relationSchemaName) {
    var contact;
    if (relationSchemaName === "Contact") {
        var id = this.getMasterEntityParameterValue("Id");
        var name = this.getMasterEntityParameterValue("Name");
        if (id && name) {
            contact = {value: id, displayValue: name};
        }
    } else {
        contact = this.getMasterEntityParameterValue("Contact");
    }
    return contact;
},
// Adds the created channel to the message publisher list.
getSectionPublishers: function() {
    var publishers = this.callParent(arguments);
    publishers.push("UsrCalls");
    return publishers;
}
},
// An array of modifications, with which the representation of the module
is built in the interface of the system.
diff: /**SCHEMA_DIFF*/
    // Adding the CallsMessageTab tab.
{
    // operation type - insertion.
    "operation": "insert",
    // Tab name.
    "name": "CallsMessageTab",
    // Parent element name.
    "parentName": "Tabs",
    // Property name.
    "propertyName": "tabs",
    // Property configuration object.
    "values": {
        // Child elements array.
        "items": []
    }
},
// Adding message container.
{
    "operation": "insert",
    "name": "CallsMessageTabContainer",
    "parentName": "CallsMessageTab",
    "propertyName": "items",
    "values": {
        // Element type - container.
        "itemType": this.Terrasoft.ViewItemType.CONTAINER,
        // Container CSS class.
        "classes": {

```

```
        "wrapClassName": ["calls-message-content"]
    },
    "items": []
}
},
// Adding the UsrCallsMessageModule module.
{
    "operation": "insert",
    "name": "UsrCallsMessageModule",
    "parentName": "CallsMessageTab",
    "propertyName": "items",
    "values": {
        // Tab module CSS class.
        "classes": {
            "wrapClassName": ["calls-message-module", "message-
module"]
        },
        // Element type - module.
        "itemType": this.Terrasoft.ViewItemType.MODULE,
        // Module name.
        "moduleName": "UsrCallsMessagePublisherModule",
        // Binding the method executed after the element has been
rendered.
        "afterrender": {
            "bindTo": "onMessageModuleRendered"
        },
        // Binding the method executed after the element has been
rerendered.
        "afterrerender": {
            "bindTo": "onMessageModuleRendered"
        }
    }
}
]/**SCHEMA_DIFF*/
);
}
);
)
```

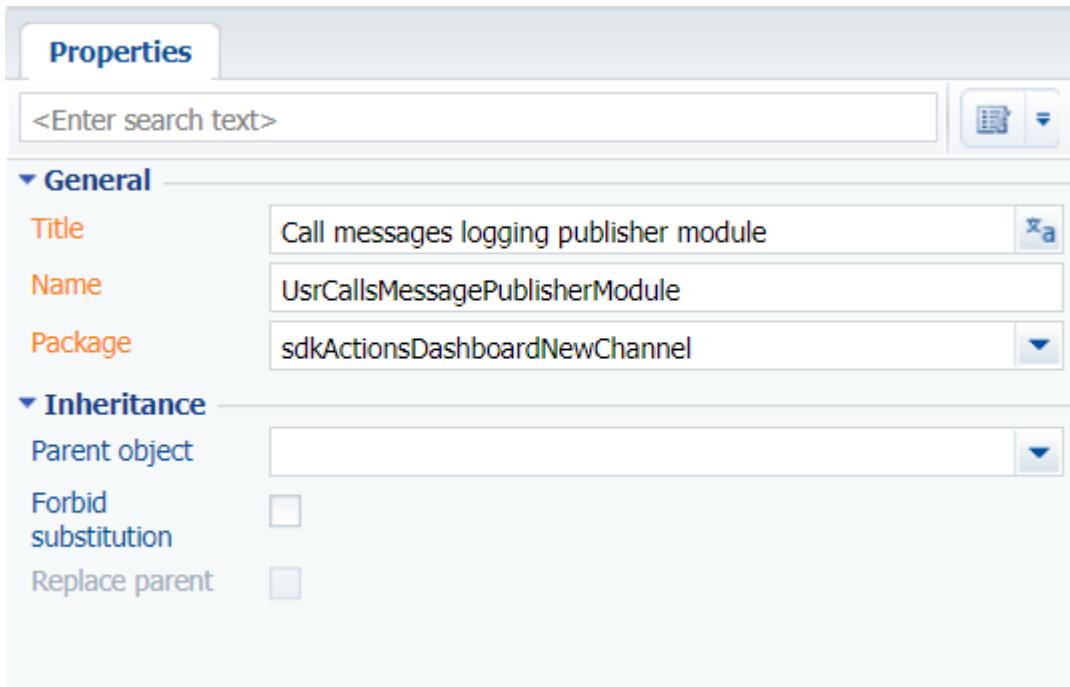
3. Create the UsrCallsMessagePublisherModule module

The *UsrCallsMessagePublisherModule* serves as container that renders the *SectionActionsDashboard* page with implemented logic of added channel in the *UsrCallsMessagePublisherPage*.

Set following properties for the module (Fig. 4):

- [Title] – "Call messages logging publisher module"
- [Name] – “UsrCallsMessagePublisherModule”
- [Parent object] – *BaseMessagePublisherModule*.

Fig. 4. Properties of the module



The module source code:

```
define("UsrCallsMessagePublisherModule", ["BaseMessagePublisherModule"],
    function() {
        // Defining the class.
        Ext.define("Terrasoft.configuration.UsrCallsMessagePublisherModule", {
            // Basic class.
            extend: "Terrasoft.BaseMessagePublisherModule",
            // Short class name.
            alternateClassName: "Terrasoft.UsrCallsMessagePublisherModule",
            // Initialization of the page that will be rendered in this module.
            initSchemaName: function() {
                this.schemaName = "UsrCallsMessagePublisherPage";
            }
        });
        // Returns the class object defined in the module.
        return Terrasoft.UsrCallsMessagePublisherModule;
    });
});
```

4. Create the UsrCallsMessagePublisherPage page

For the created page set the *BaseMessagePublisherPage* schema of the *MessagePublisher* package as parent object. Set the "UsrCallsMessagePublisherPage" value as the title and name.

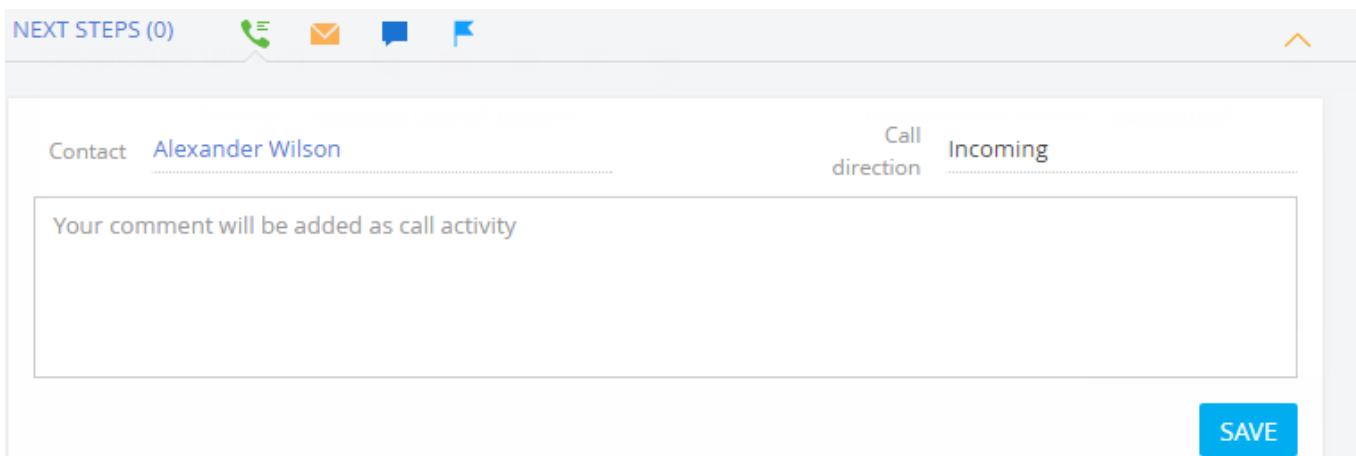
In the source code page, specify the schema name of the object that will run along with the page (in this case, *Activity*), implement the logic of message publication and override the *getServiceConfig* method, in which you must set the class name from the configuration.

```
//Sets the class that will work with this page.
getServiceConfig: function() {
    return {
        className: "Terrasoft.Configuration.CallsMessagePublisher"
    };
}
```

Implementation of message publication logic contains big number of methods, attributes and properties. The full source code of the *UsrCallsMessagePublisherPage* schema can be found in the [sdkActionsDashboardNewChannel](#) package. The source code shows the implementation of the *CallMessagePublisher* channel that is used for logging incoming and outgoing calls.

As a result you will get the new channel in the *SectionActionsDashboard* (Fig. 5).

Fig. 5. An example of a custom CallsMessagePublisher channel in the SectionActionsDashboard of the [Contacts] section.



Displaying contact's time zone

Beginner Easy Medium **Advanced**

General information

The ability to work with different timezones was introduced in version 7.8. Contact pages now display information about the contact's local timezone. Timezone values, such as time difference between the user's timezone and the contact's timezone, are calculated automatically. For more information about timezone calculation please see the "[How to determine the current local time for a contact](#)". The information is displayed in the element generated by the view generator. This element cannot be added to a page with the section wizard.

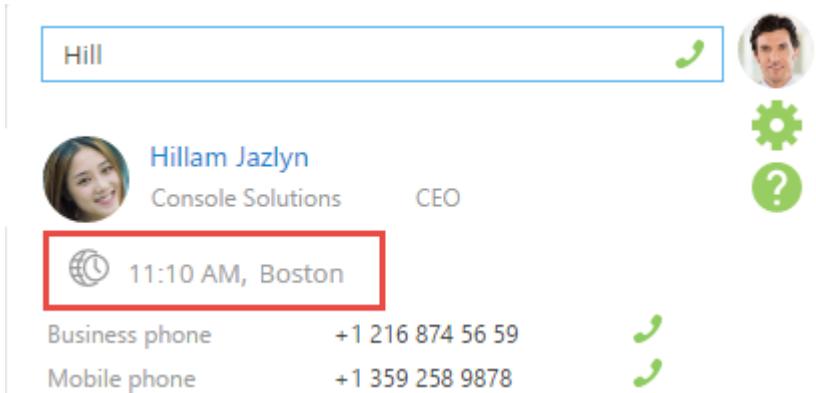
To add a contact timezone display element to a custom page:

1. Create a replacing page schema module.
2. Add timezone display element.
3. Connect timezone search.
4. Set up display styles.

Case description

Add a contact timezone display element to the call panel. The contact's current local time must be displayed when searching for phone numbers to ensure that subscribers in different timezones are contacted during business hours only.

Fig. 1. Displaying subscriber's current local time



Source code

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

Case implementation algorithm

1. Create a replacing page schema module

To modify this schema, add a replacing client module in the custom package (Fig. 2) and specify *SubscriberSearchResultItem* as the parent object and custom package (Fig. 3). Creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 2. Creating a replacing module

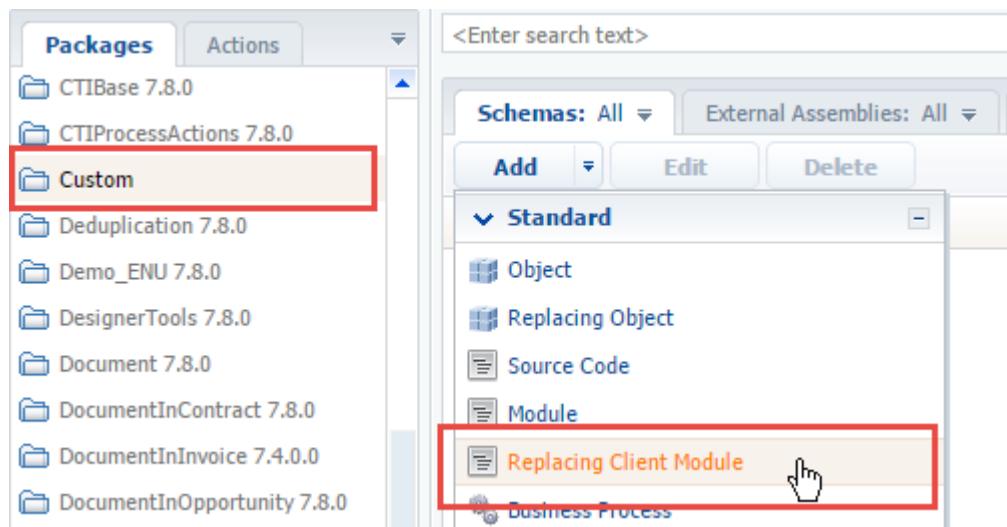
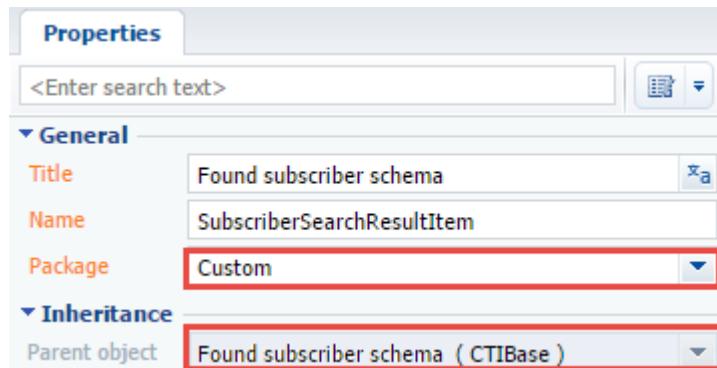


Fig. 3 Replacing module properties



2. Add timezone display element

Add modules according to the schema dependency:

- *TimezoneGenerator* – module that helps to create contact's timezone display item.
- *TimezoneMixin* – mixin is used to search for contact's timezone.

Add configuration object for displaying a contact's timezone item to the *diff* array.

3. Connect timezone search

To run the contact timezone search, pass the contact's unique Id to the *init* method of the *TimezoneMixin*. As a result of execution the following attributes will be set:

- *TimeZoneCaption* – contact's timezone name and current local time.
- *TimeZoneCity* – name of the city for which the timezone is set.

Source code of the schema:

```
// Declaring schema.
```

```
define("SubscriberSearchResultItem",
    // Dependency from TimezoneGenerator, TimezoneMixin.
    ["TimezoneGenerator", "TimezoneMixin",
     // Dependency from the module with style.
     "css!UsrSubscriberSearchResultItemCss"],
    function() {
        return {
            // Block for creating attributes.
            attributes: {
                // Name of the attribute that controls timexone element display
                status.

                "IsShowTimeZone": {
                    // Attribute data type.
                    "dataValueType": Terrasoft.DataValueType.BOOLEAN,
                    // Attribute type in model.
                    "type": Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                    // Default value.
                    "value": true
                }
            },
            // Mixin connection block.
            mixins: {
                // Connecting mixin.
                TimezoneMixin: "Terrasoft.TimezoneMixin"
            },
            // Method definition block.
            methods: {
                // Class constructor.
                constructor: function() {
                    // Selecting base constructor.
                    this.callParent(arguments);
                    // Indicates if the subscriber is a contact.
                    var isContact = this.isContactType();
                    // The element is displayed if the subscriber is a contact.
                    this.set("IsShowTimeZone", isContact);
                    // If the subscriber is a contact.
                    if (isContact) {
                        // Contact Id.
                        var contactId = this.get("Id");
                        // Searching for contact's timezone.
                        this.mixins.TimezoneMixin.init.call(this, contactId);
                    }
                },
                // Gets an indicator that the subscriber is a contact.
                isContactType: function() {
                    // Subscriber type.
                    var type = this.get("Type");
                    // Gets comparison result.
                    return type === "Contact";
                }
            },
            // Array of modifications.
            diff: [
                {
                    // Adding a new element.
                    "operation": "insert",
                    // Parent element is SubscriberSearchResultItemContainer.
                    "parentName": "SubscriberSearchResultItemContainer",
                    // New element is added to the collection of elements of the
                    parent.
                    "propertyName": "items",
                    // Element name.
                }
            ]
        }
    }
);
```

```
        "name": "TimezoneContact",
        // Element properties.
        "values": {
            // Element type.
            "itemType": Terrasoft.ViewItemType.CONTAINER,
            // Generator method is called for generating view
configuration.
            "generator": "TimezoneGenerator.generateTimeZone",
            // Container visibility is bound to an attribute.
            "visible": {"bindTo": "IsShowTimeZone"},
            // Element style.
            "wrapClass": ["subscriber-data", "timezone"],
            // Binding title to attribute.
            "timeZoneCaption": {"bindTo": "TimeZoneCaption"},
            // Binding cisty to attribute.
            "timeZoneCity": {"bindTo": "TimeZoneCity"}
        },
        // Element position in the parent container.
        "index": 2
    }
}
};
```

As a result, the contact's current local time and city are displayed.

4. Display style setup

During the previous steps, the configuration object placed in the `diff` array already has preliminary display settings.

Use the `index` property to adjust element positioning. By default, the elements are placed one after another in the `SubscriberSearchResultItemContainer`:

- the first element is subscriber photo with index "0",
 - then subscriber information with index "1",
 - then subscriber phone numbers with index "2".

If you set the index value to "2", the element will be displayed between subscriber information and the list of phone numbers.

Use the `subscriber-data` CSS-class to set styles for text elements in the schema. The element generator provides the `wrapClass` property to manage styles.

Create the `UsrSubscriberSearchResultItemCss` module in the custom package for positioning of the element and its visual highlighting.

In the LESS tab of the created module, add CSS-selectors for classes that will determine the required styles.

```
/* Display style setup for the added element.*/
.ctiPanelMain .search-result-items-list-container .timezone {
    /* Top padding.*/
    padding-top: 13px;
    /* Bottom margin.*/
    margin-bottom: -10px;
}
/* Setting styles to display contact time.*/
.ctiPanelMain .search-result-items-list-container .timezone-caption {
    /* Left padding.*/
    padding-left: 10px;
    /* Text color.*/
    color: rgb(255, 174, 0);
    /* Text font - bold.*/
    font-weight: bold;
}
```

```
/* Display style setup for contact city.*/
.ctiPanelMain .search-result-items-list-container .timezone-city {
    /* Left padding.*/
    padding-left: 10px;
}
```

To load module with styles, add the following code to the module and save the schema.

```
define("UsrSubscriberSearchResultItemCss", [],  
    function() {  
        return {};  
    }) ;
```

Add the *UsrSubscriberSearchResultItemCss* module to the *SubscriberSearchResultItem* dependencies.

After saving the schema and refreshing the application page, the subscriber search results will be displayed as shown on Fig. 1.

How to display the difference between dates on edit page fields

Beginner

Easy

Medium

Advanced

Creatio uses the capabilities of the standard *Date* JavaScript-object to work with dates on the client's side of the application. For example, the [*Date.prototype.getDate\(\)*](#) method is used to display the day of the month for a specific date in accordance with the local time, and the [*Date.prototype.setDate\(\)*](#) method is used to set the day of the month relative to the current month. All properties and methods of the *Date* object may be found in the [documentation](#).

For example, when creating a new contract, the [End Date] field should display a date that is 5 days longer than the [Start Day] field. To do this:

1. Create a replacing *ContactPageV2* edit page schema of the [Contracts] section. The procedure for creating a replacing client schema is covered in the “**Creating a custom client module schema**”.

2. Add the following code to the created module:

```
define("ContractPageV2", [], function() {  
    return {  
        entitySchemaName: "Contract",  
        methods: {  
            // The date is set after the object is initialized.  
            onEntityInitialized: function() {  
                // Checking the mode of the new record.  
                if ((this.isAddMode() && this.Ext.isEmpty(this.get("EndDate")))) {  
                    // Calling an auxiliary method.  
                    this.setEndDate(this.get("StartDate"), 5);  
                }  
                // Calling the base functionality.  
                this.callParent(arguments);  
            },  
            // Auxiliary method for setting the date.  
            setEndDate: function(date, dateOffsetInDays) {  
                var offsetDate = new Date();  
                offsetDate.setDate(date.getDate() + dateOffsetInDays);  
                this.set("EndDate", offsetDate);  
            }  
        }  
    };  
});
```

3. Save the changes.

4. Refresh browser page.

As a result, while adding a new contract, its end date will be 5 days ahead of its start date.

Fig. 1. Case result

The screenshot shows a Creatio application interface for managing contracts. On the left is a sidebar with navigation links: Dashboards, Feed, Leads, Accounts, Contacts, Activities, Opportunities, and Orders. The main area displays a form for a contract with the following details:

- Number:** 22
- Type:** Contract
- Start date:** 8/30/2017
- End date:** 9/4/2017
- Owner:** Supervisor
- Status:** Draft

The form has several tabs at the top: GENERAL INFORMATION (selected), CONTRACT DETAILS, HISTORY, APPROVALS, ATTACHMENTS AND NOTES, and FEED. Below the tabs, there are sections for Account, Account's banking details, Contact, and Amount. A sidebar on the right contains icons for user profile, settings, help, phone, email, and notifications (with a red '4' badge).

In order for the date in the [End Date] field to be recalculated automatically when the user changes the [Start Day] field, it is necessary to use the functionality of the computed fields. Please refer to the “[Adding calculated fields](#)” article for more details.

How to block fields of the edit page

Beginner

Easy

Medium

Advanced

Introduction

During the development of the Creatio custom functions you may need to block all fields and details on the page when specific condition is met. Mechanism of blocking of the edit page fields can simplify the process without creating a number of business rules.

More information about blocking of the page fields can be found in the “[Locking edit page fields](#)”.

Blocking mechanism is implemented in the Creatio version 7.11.1 or higher.

Case description

Block all the fields on the invoice edit page if the invoice is on the [Paid] stage. The [Payment status] field and the [Activities] detail should stay editable.

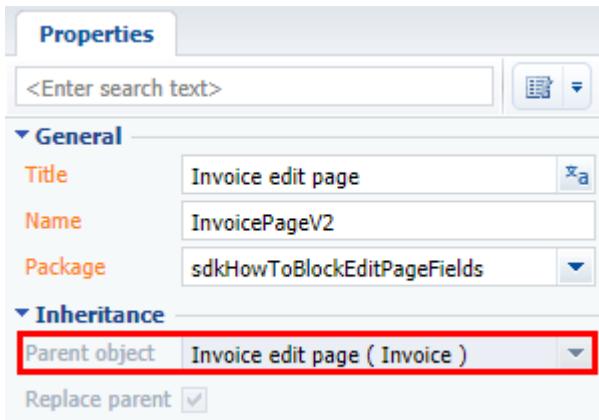
If the field has binding for the `enabled` property in the `diff` array element or in the BINDPARAMETER business rule, the mechanism will not block this field.

Case implementation algorithm

1. Create a replacing schema of the invoice edit page

Create a replacing client module and specify the [Invoice edit page] schema as parent (Fig. 1). The procedure for creating a replacing client schema is covered in the “[Creating a custom client module schema](#)” article.

Fig. 1. The properties of the [Invoice edit page] schema



2. Add schema source code

Add the source code of implementation of the [Invoice edit page] replacing schema on the [Source code] panel of schema designer. The source code is available below:

```
define("InvoicePageV2", ["InvoiceConfigurationConstants"],
function(InvoiceConfigurationConstants) {
    return {
        entitySchemaName: "Invoice",
        attributes: {
            // Status of the blocking of fields.
            "IsModelItemsEnabled": {
                dataValueType: Terrasoft.DataValueType.BOOLEAN,
                value: true,
                dependencies: [
                    {
                        columns: ["PaymentStatus"],
                        methodName: "setCardLockoutStatus"
                    }
                ]
            },
            methods: {
                getDisableExclusionsColumnTags: function() {
                    // The [Payment status] field should not be blocked.
                    return ["PaymentStatus"];
                },
                getDisableExclusionsDetailSchemaNames: function() {
                    // Also, the "Activity" detail is not blocked.
                    return ["ActivityDetailV2"];
                },
                setCardLockoutStatus: function() {
                    // Get current invoice status.
                    var state = this.get("PaymentStatus");
                    // If the current account status is "paid", then block the fields.
                    if (state.value ===
InvoiceConfigurationConstants.Invoice.PaymentStatus.Paid) {
                        // Set a property that stores the field lock flag.
                        this.set("IsModelItemsEnabled", false);
                    } else {
                        // Otherwise, unlock the fields.
                        this.set("IsModelItemsEnabled", true);
                    }
                },
                onEntityInitialized: function() {
                    this.callParent(arguments);
                    // Set the status of the blocking of fields.
                    this.setCardLockoutStatus();
                }
            }
        }
    }
});
```

```

        }
    },
    diff: /**SCHEMA_DIFF*/ [
    {
        "operation": "merge",
        "name": "CardContentWrapper",
        "values": {
            "generator": "DisableControlsGenerator.generatePartial"
        }
    }
] /**SCHEMA_DIFF*/
);
}
);

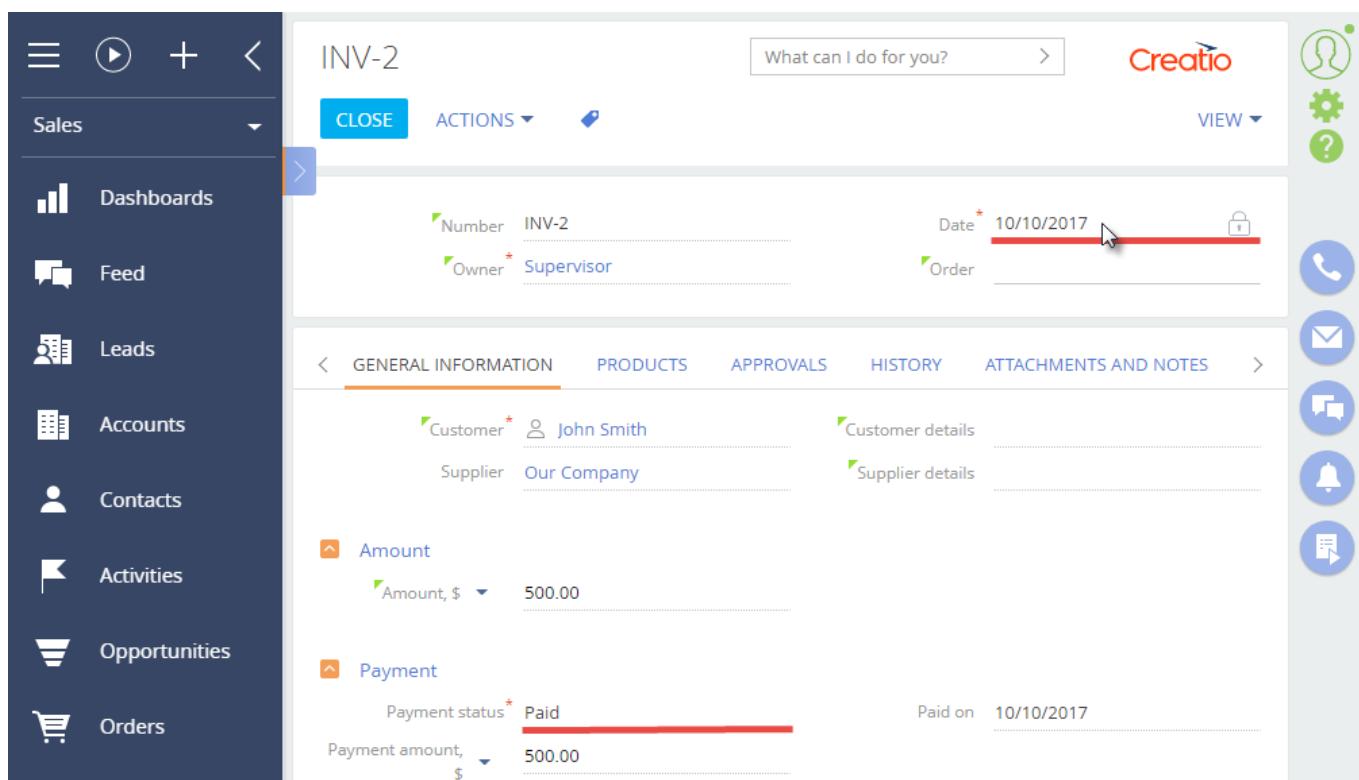
```

The `IsModelItemsEnabled` attribute will be defined and methods for blocking and unlocking the field of the invoice edit page will be implemented. The `CardContentWrapper` container of the edit page is used as the container of the blocking mechanism.

Save the schema after adding the source code. Then clear the browser cache.

As a result, the most of the invoice fields will be blocked after changing the status to the [Paid]. Fields and details specified in the exceptions for blocking will stay unlocked. The fields that have the `enabled` property explicitly set to `true` will stay unlocked

Fig. 2. Case result



Work with details

Contents

- **Introduction**
- **Creating a detail in wizards**
- **Adding an edit page detail**
- **Adding a detail with an editable list**
- **Creating a detail with selection from lookup**

- Adding multiple records to a detail
- Creating a custom detail with fields
- Advanced settings of a custom detail with fields
- Adding the [Attachments] detail
- Displaying additional columns on the [Attachments] tab
- How to hide menu commands of the detail with list
- Deleting a detail

Adding details

Beginner

Easy

Medium

Advanced

Overview

Details are the elements of the section record edit page that display supplemental data for a primary section object. The details displayed on the tabs of section edit page in the tabs container.

There are four main types of details:

1. A detail with adding page is a standard Creatio detail. It can be created manually or added to the section via the [detail wizard](#). More information about detail creation can be found in the “**Adding an edit page detail**” and “**Creating a detail in wizards**” articles.
2. A detail with editable list different from the standard detail. The data can be added, deleted and modified directly in the detail. More information about creation of the detail with editable list can be found in the “**Adding a detail with an editable list**” article.
3. A detail with selection from lookup – data are selected from a lookup displayed in the modal window. More information about detail creation can be found in the “**Creating a detail with selection from lookup**” article.
4. A detail with edit fields – data are filled in and edited in the detail data fields. More information about detail creation can be found in the “**Creating a custom detail with fields**” and “**Advanced settings of a custom detail with fields**” article.

More information about the details of each type is described in the “**Details**” article of the “**Application interface and structure (on-line documentation)**” section.

Main schemas, classes, methods and properties of detail functions are described in the “**Details**” article of the “**Controls (on-line documentation)**” section.

Contents

- **Adding an edit page detail**
- **Adding a detail with an editable list**
- **Creating a detail with selection from lookup**
- **Adding multiple records to a detail**
- **Creating a custom detail with fields**
- **Advanced settings of a custom detail with fields**
- **Creating a detail in wizards**
- **Adding the [Attachments] detail**
- **Displaying additional columns on the [Attachments] tab**
- **How to hide menu commands of the detail with list**
- **Deleting a detail**

Creating a detail in wizards

Beginner

Easy

Medium

Advanced

Introduction

Detail is an element of the section record edit page, designed to display additional data related to the main section object. Details are displayed on edit page tabs. The difference between details and section records is that details are stored in a separate object, and the records in this database object are usually associated with the main section record entity with the "many-to-one" ratio. Please refer to the "**Details**" article in the "**Application interface and structure (on-line documentation)**" section for more information.

Details are standard Creatio elements and can be added to the section by using a [detail wizard](#) or a [section wizard](#).

The general outline for adding a detail with an "add" page to an existing system section:

1. Create a detail object schema
2. Create a schema of a client detail list module and the schema of a detail edit page
3. Implement the detail on the section record edit page using the section wizard.

Case description

Create a custom [Contact's ID] detail in the [Contacts] section. The detail must display the contact's ID number and the document number. One contact may have several ID's.

Case implementation algorithm

1. Creating a detail object schema

Learn more about adding object schema columns in the "**Creating the entity schema**" article.

Create an object schema with the following parameters (Fig. 1):

- [Title] – "Contact Identity Card".
- [Name] – "UsrContactIdentityCard".
- [Package] – "Custom" (or a different custom package).
- [Parent object] – "Base object", implemented in the *Base* package.

Add three columns in the object structure. Column properties are listed in Table 1.

Table 1. – Column properties of the UsrRegDocument detail object schema

Title	Name	Data Type	Lookup
Series	UsrSeries	Text (50 characters)	–
Number	UsrNumber	Text (50 characters)	–
Contact	UsrContact	Lookup	Contact

Publish the schema to apply changes.

Add columns in the detail wizard.

2. Creating a schema of the detail list client module and a schema of the detail edit page.

If the development needs to be carried out in a custom package, it needs to be specified in the [Current package] system setting. Otherwise, the detail wizard will not be able to save the changes to the package used for development.

To create a new detail in a wizard, go to the [Detail wizard] section in the [System setup] group of the system designer.

On the [DETAIL] step, specify the title of the detail and select the main detail object (Fig. 1).

Fig. 1. The [DETAIL] step in the detail wizard

Contact Identity Card: Detail

The screenshot shows the 'Contact Identity Card: Detail' step of a detail wizard. At the top, there are 'SAVE' and 'CANCEL' buttons. To the right of these are navigation arrows and tabs labeled 'DETAIL', 'PAGE', and 'BUSINESS RULES'. Below the tabs, there are two input fields: 'Title * Contact Identity Card' and 'Object * Contact Identity Card'. The 'PAGE' tab is currently selected.

Arrange the required detail columns on the [PAGE] step.

Fig. 2. The [PAGE] step in the detail wizard

Contact Identity Card: Page

The screenshot shows the 'Contact Identity Card: Page' step of a detail wizard. At the top, there are 'SAVE' and 'CANCEL' buttons. To the right of these are navigation arrows and tabs labeled 'DETAIL', 'PAGE', and 'BUSINESS RULES'. A sidebar on the left contains a button 'Add existing column' and a list of available columns: 'Contact', 'Created by', 'Created on', 'Modified by', 'Modified on', 'Number', and 'Series'. The 'PAGE' tab is currently selected. The main area displays a grid with two columns: 'Series' and 'Number'. The 'Series' column has 10 empty rows, and the 'Number' column has 10 empty rows. Below the grid is a toolbar with icons for adding, editing, and deleting columns.

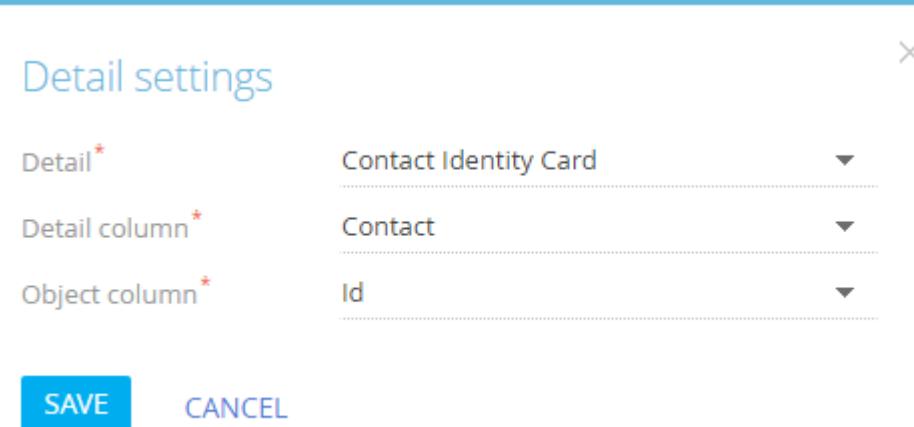
Save the detail when the setup is done.

As a result, the custom package will have a schema of the detail list client module and a schema of the detail edit page.

3. Implement the detail on the section record edit page using the detail wizard

Open the detail wizard in the [Contacts] section, and select [NEW DETAIL] on the [PAGE] step. In the opened window, select the [Contact's ID] detail and configure the connection between detail object columns and the section object (Fig. 3).

Fig. 3. Detail properties setup



The detail will be displayed in the section record page constructor (Fig. 4).

Fig. 4. A detail in the section record page constructor

- Detail: Contact communication options
- Detail: Contact addresses
- Detail: Noteworthy events of contact
- Detail: Contact relationships
- Detail: Contact Identity Card

NEW FIELDS GROUP

NEW DETAIL

Save the changes when the section record page setup is done.

As a result, the custom package will have a replacing client module of a section page and a schema of the section record edit page.

Upon refreshing, the detail will be displayed on a record edit page.

Fig. 5. Case result

Andrew Baker (sample)

Birthday: 1/20/1986

Connected to: No data

Contact Identity Card:

Series: AA-BB	Number: 1111111
---------------	-----------------

It is necessary to configure the columns in a detail menu, and add a few records to see the result.

Fig. 6. Adding a record to a detail

Andrew Baker (sample)...

What can I do for you? > Creatio

SAVE CANCEL ACTIONS ▾

Series: BB-CC Number: 2222222

Adding an edit page detail

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

General provisions

Details are special elements of section edit pages, which display additional data that is not stored in the fields of the section primary object. The details are displayed on the edit page tabs in the tab container.

There are four basic types of details:

- details with a record edit page;
- details with edit fields;
- details with editable list;
- details with lookup selection.

For more information on details, please see the "**Details**" article in the "**Application interface and structure (on-line documentation)**" section.

An edit page detail is a standard Creatio detail that can be added to sections using the [Detail Wizard](#). Below is an example of adding an edit page detail without the use of the Detail Wizard.

General procedure for adding an edit page detail to an existing section

To add a custom edit page detail:

1. Create a detail object schema.
2. Create a detail schema.
3. Create a detail edit page schema.
4. Set up the detail in a replacing section edit page schema.
5. Register links between object, detail and detail page schemas, using special SQL queries to system tables.
6. Set up the detail fields.

To bind custom schemas, make changes to the SysModuleEdit, SysModulenty and SysDetail system tables in the Creatio database, using SQL queries.

Be extremely careful when composing and executing SQL queries to the database. Executing an invalid query may damage existing data and disrupt system operation.

Case description

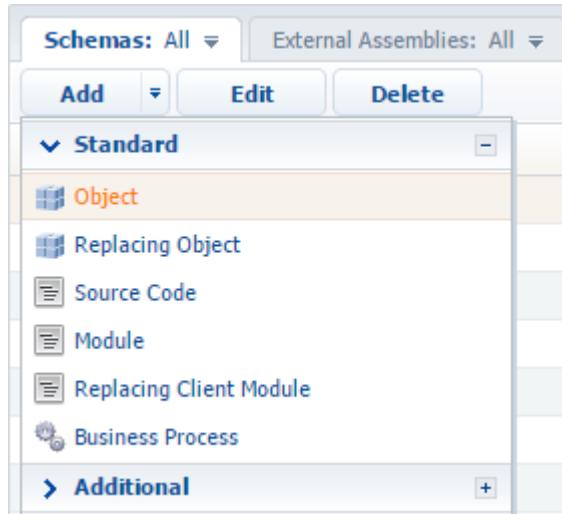
Create a custom [Couriers] detail in the [Orders] section. The detail must contain the list of couriers for the current order.

Case implementation algorithm

1. Create detail object schema

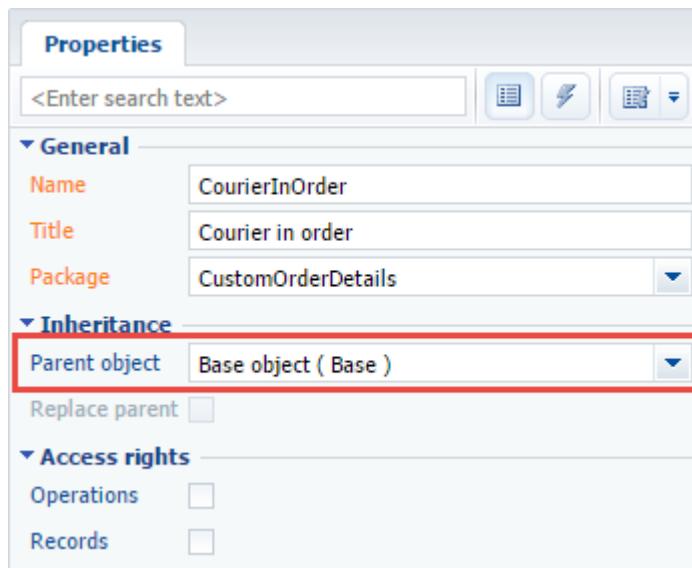
In the settings mode, go to the [Configuration] section, open the [Schemas] tab and execute the [Add] > [Object] menu command (Fig. 1).

Fig. 1. Adding a detail object schema



In the opened **Object Designer**, fill out the properties of the detail object schema, as shown on Fig. 2.

Fig. 2. Setup of detail object schema properties



Add a lookup column [Order], which will connect the detail object with the section object, and a lookup [Contact] column where the courier's contact will be stored. Both columns must be required. Column properties setup is shown on Fig. 3 and Fig. 4.

Fig. 3. Specifying the properties of the [Order] column

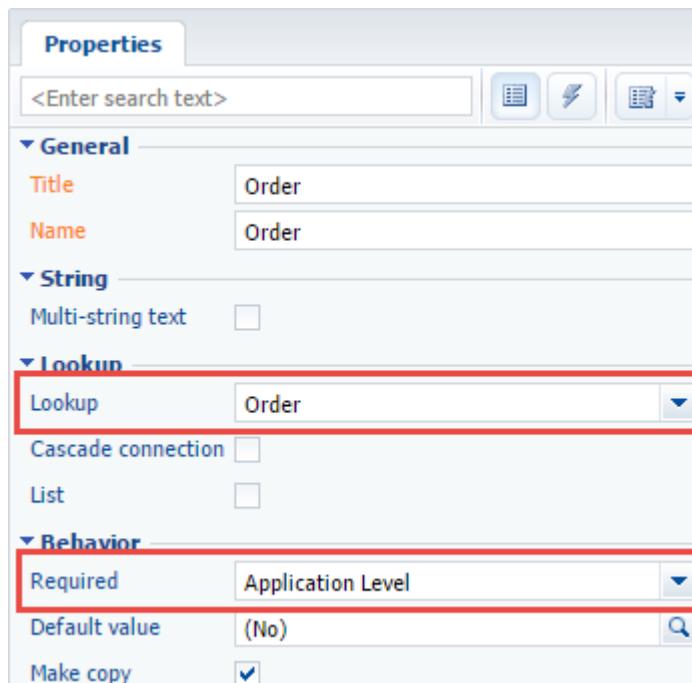
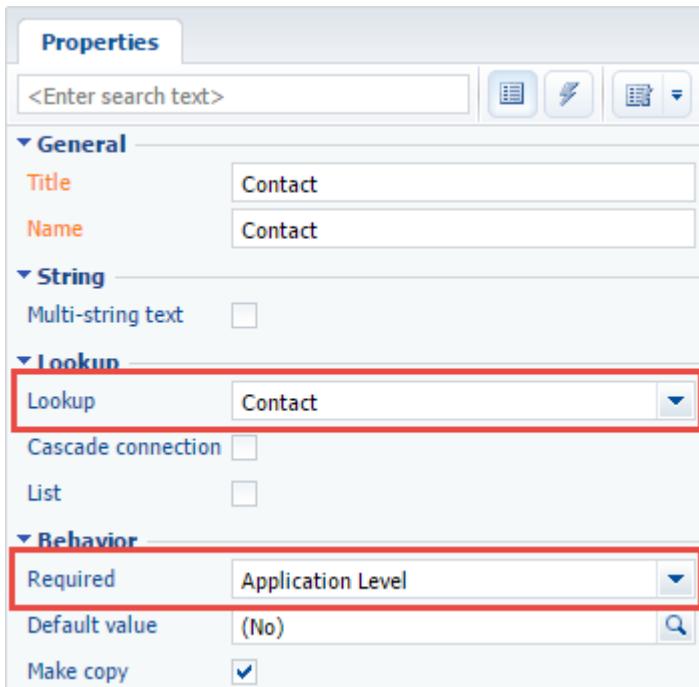


Fig. 4. Specifying the properties of the [Contact] column



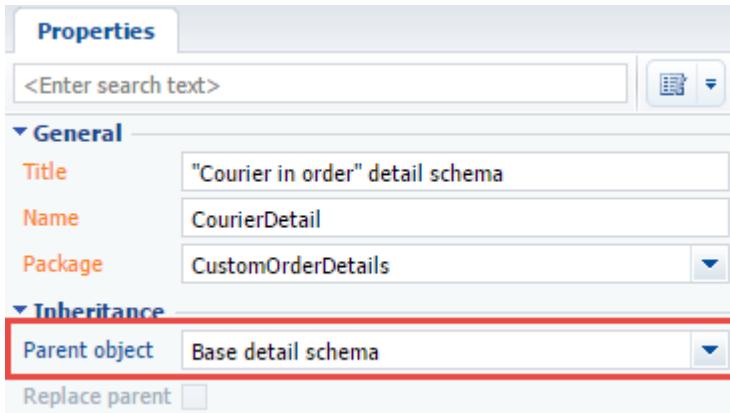
Object schema must be saved and published.

2. Create detail schema

In the settings mode, go to the [Configuration] section, open the [Schemas] tab and execute the [Add] > [Module] menu command (Fig. 1).

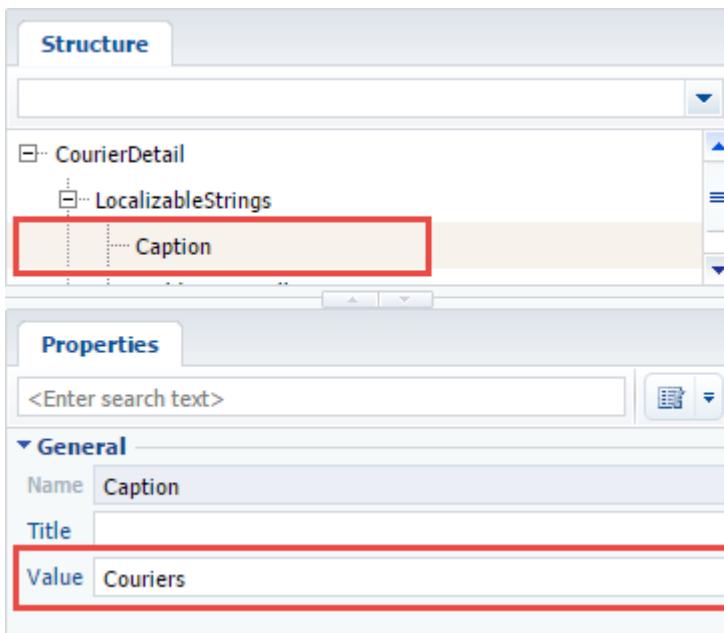
The new module must inherit the functionality of the base detail schema with list *BaseGridDetailV2*, which is available in the *NUI* package. To do this, specify this schema as the parent for the created detail schema (Fig. 5). The rest of the properties must be set up as shown on Fig. 5. In the [Package] property, the system will specify the current package name.

Fig. 5. Detail schema properties



Set the *Couriers* value for the [Caption] localizable string of the detail schema (Fig. 6). The [Caption] string contains detail title, shown on the edit page.

Fig. 6. Setting the value of the [Caption] string



Below is the detail schema source code, which must be added to the [Source code] section of the client module designer. The code defines the schema name and its connection to the object schema.

```
define("CourierDetail", [], function() {
    return {
        // Detail object schema name.
        entitySchemaName: "UsrCourierInOrder",
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        methods: {},
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    };
});
```

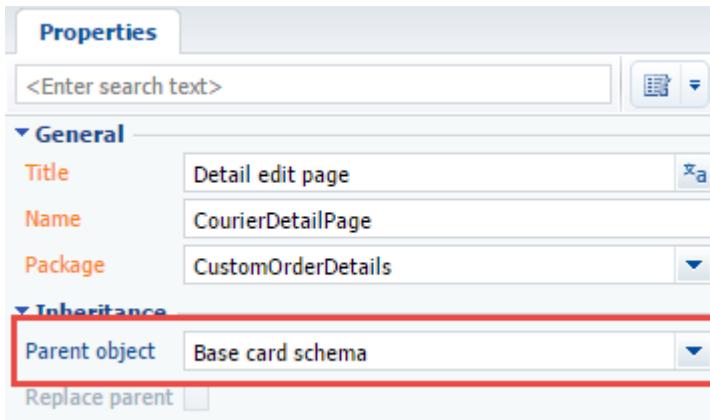
Save the detail schema to apply the changes.

3. Create detail edit page schema

In the settings mode, go to the [Configuration] section, open the [Schemas] tab and execute the [Add] > [Module] menu command (Fig. 1).

The created detail edit page schema must inherit the functionality of base page schema *BasePageV2*, which is available in the *NUI* package. To do this, specify this schema as parent (Fig. 7). The rest of the properties must be set up as shown on Fig. 7. In the [Package] property, the system will specify the current package name.

Fig. 7. Detail edit page schema properties



To set up fields displayed on the detail edit page, add the following code to the [Source code] section of the client

module designer. In this code, in the diff array, configuration objects of metadata for adding, setting location and binding the order and courier fields are specified.

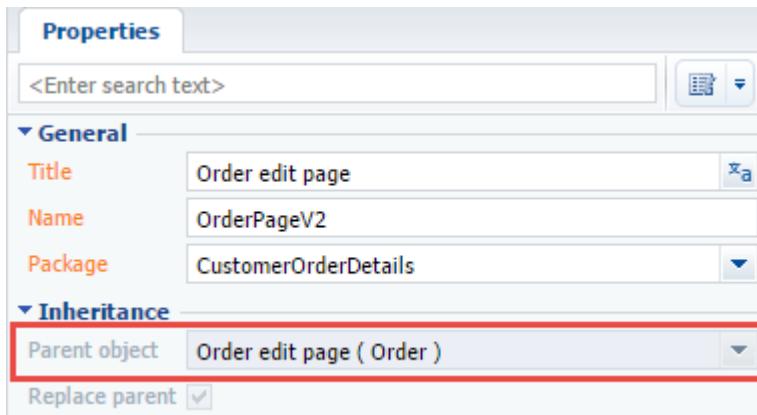
```
define("CourierDetailPage", [], function() {
    return {
        // Detail object schema name.
        entitySchemaName: "UsrCourierInOrder",
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        // Array with modifications.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding the [Order] field.
            {
                "operation": "insert",
                //Field name.
                "name": "Order",
                "values": {
                    // Field position setup on the edit page.
                    "layout": {
                        "colSpan": 12,
                        "rowSpan": 1,
                        "column": 0,
                        "row": 0,
                        "layoutName": "Header"
                    },
                    // Binding to the [Order] column of the object schema.
                    "bindTo": "UsrOrder"
                },
                "parentName": "Header",
                "propertyName": "items",
                "index": 0
            },
            // Metadata for adding the [Contact] field.
            {
                "operation": "insert",
                //Field name.
                "name": "Contact",
                "values": {
                    // Field position setup on the edit page.
                    "layout": {
                        "colSpan": 12,
                        "rowSpan": 1,
                        "column": 12,
                        "row": 0,
                        "layoutName": "Header"
                    },
                    // Binding to the [Contact] column of the object schema.
                    "bindTo": "UsrContact"
                },
                "parentName": "Header",
                "propertyName": "items",
                "index": 1
            }
        ]/**SCHEMA_DIFF*/,
        methods: {},
        rules: {}
    };
}) ;
```

Save the detail edit page schema to apply the changes.

4. Set up detail in replacing section edit page schema

To display the detail on the order edit page, first create a replacing client module, and specify the order edit page *OrderPageV2* (located in the Order package) as the parent (Fig. 8). For more information on creating replacing schemas, please see the "**Creating a custom client module schema**" article.

Fig. 8. Order edit page replacing schema properties



To display the [Couriers] detail on the [Delivery] tab of the order edit page, add the following source code. In the *details* section, a new *CourierDetail* model will be defined, and its location on the edit page will be specified in the modification section of the *diff* array.

```
define("OrderPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Order",
        // List of edit page details being added.
        details: /**SCHEMA_DETAILS*/{
            // [Couriers] detail setup.
            "CourierDetail": {
                // Detail schema name.
                "schemaName": "UsrCourierDetail",
                // Detail object schema name.
                "entitySchemaName": "UsrCourierInOrder",
                // Filtering contacts for current order only.
                "filter": {
                    // Detail object schema column.
                    "detailColumn": "UsrOrder",
                    // Section object schema column.
                    "masterColumn": "Id"
                }
            }
        }/**SCHEMA_DETAILS*/,
        // Array of modifications.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding the [Couriers] detail.
            {
                "operation": "insert",
                // Detail name.
                "name": "CourierDetail",
                "values": {
                    "itemType": 2,
                    "markerValue": "added-detail"
                },
                // Parent container ([Delivery] tab).
                "parentName": "OrderDeliveryTab",
                // Container where detail is located.
                "propertyName": "items",
                // Index in the list of added elements.
                "index": 3
            }
        ]
    }
})
```

```

        }
    ] /**SCHEMA_DIFF*/,
    methods: {},
    rules: {}
);
);
}) ;
}
);
});
```

To apply the changes, the replacing page schema must be saved.

After this, the detail will be displayed on the edit page of the [Orders] section. New records cannot be added to the detail until the connections between the detail schemas are registered.

5. Register connections between the schemas using SQL queries to system tables

To register connection between a detail object schema and detail schema, execute the following SQL query.

```

DECLARE
-- Schema name of the created detail.
@DetailSchemaName NCHAR(100) = 'CourierDetail',
-- Detail title.
@DetailCaption NCHAR(100) = 'Couriers',
-- Schema name of the object, to which the detail is bound.
@EntitySchemaName NCHAR(100) = 'CourierInOrder'

INSERT INTO SysDetail(
    ProcessListeners,
    Caption,
    DetailSchemaUID,
    EntitySchemaUID
)
VALUES (
    0,
    @DetailCaption,
    (SELECT TOP 1 UIid
    FROM SysSchema
    WHERE name = @DetailSchemaName),
    (SELECT TOP 1 UIid
    FROM SysSchema
    WHERE name = @EntitySchemaName)
)
```

To register connection between the detail object schema and edit page schema, execute the following SQL query.

```

DECLARE
-- Schema name of the detail page
@CardSchemaName NCHAR(100) = 'CourierDetailPage',
-- Object schema.
@EntitySchemaName NCHAR(100) = 'CourierInOrder',
-- Detail page caption.
@PageCaption NCHAR(100) = 'Page of detail "Courier In Order"',
-- Empty string.
@Blank NCHAR(100) = ''

INSERT INTO SysModuleEntity(
    ProcessListeners,
    SysEntitySchemaUID
)
VALUES (
    0,
    (SELECT TOP 1 UIid
    FROM SysSchema
    WHERE Name = @EntitySchemaName
```

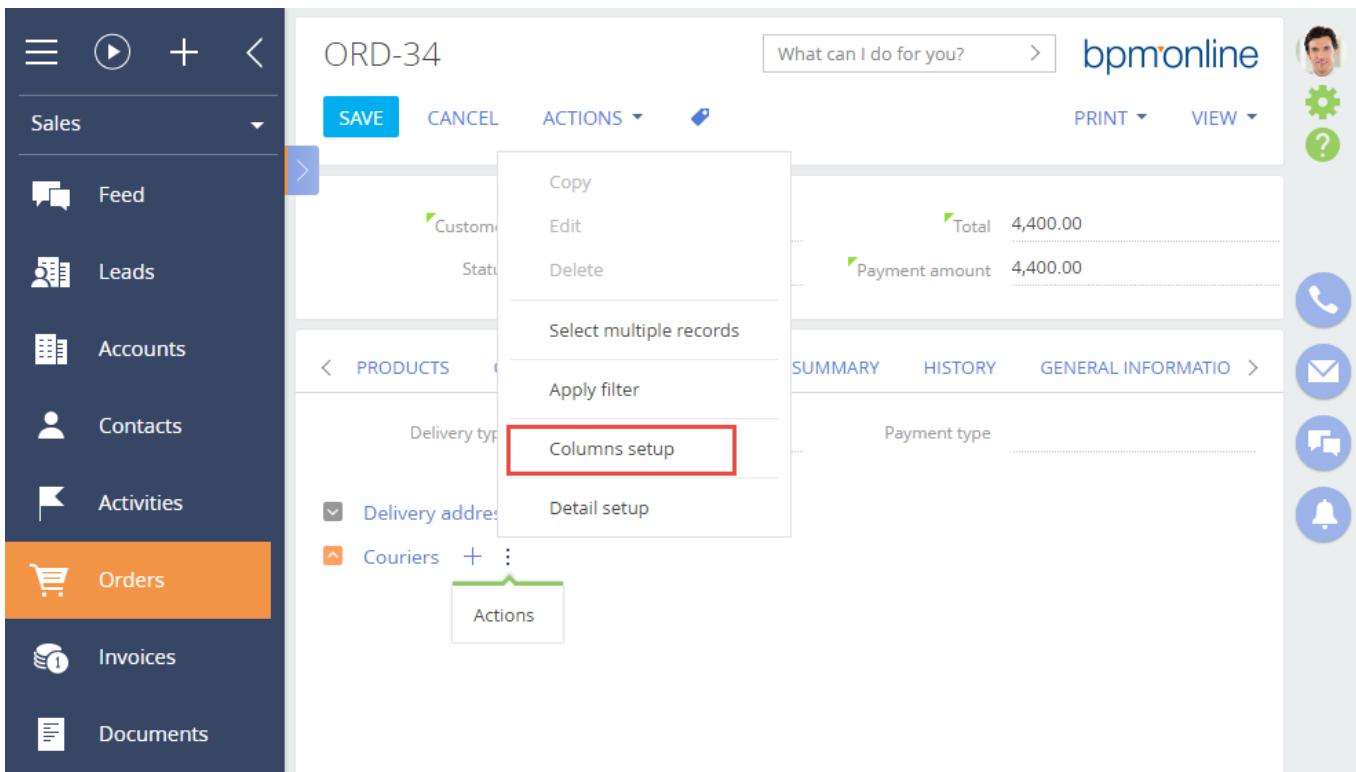
```
)  
)  
  
INSERT INTO SysModuleEdit(  
    SysModuleEntityId,  
    UseModuleDetails,  
    Position,  
    HelpContextId,  
    ProcessListeners,  
    CardSchemaUid,  
    ActionKindCaption,  
    ActionKindName,  
    PageCaption  
)  
VALUES (  
    (SELECT TOP 1 Id  
    FROM SysModuleEntity  
    WHERE SysEntitySchemaUid = (  
        SELECT TOP 1 UId  
        FROM SysSchema  
        WHERE Name = @EntitySchemaName  
    ))  
,  
    1,  
    0,  
    @Blank,  
    0,  
    (SELECT TOP 1 UId  
    FROM SysSchema  
    WHERE name = @CardSchemaName  
)  
,  
    @Blank,  
    @Blank,  
    @PageCaption  
)
```

To apply these changes on the application level, restart the application site in IIS, or compile the application in the [Configuration] section.

6. Set up detail list columns

At this stage, the detail is completely operational, however contacts are not displayed on the detail, because the detail list does not have displayed columns specified for it. Go to the detail menu and set up columns to display (Fig. 9).

Fig. 9. Detail actions menu



See also:

- [Creating a detail in wizards](#)
- [Adding a detail with an editable list](#)
- [Creating a detail with selection from lookup](#)
- [Creating a custom detail with fields](#)

Adding a detail with an editable list

Beginner Easy Medium **Advanced**

Introduction

Details are elements of section edit pages. Details display records bound to the current section record by a lookup field. These can be, for example, records from other sections, whose primary objects contain lookup columns linked to the primary object of the current section. Details are displayed on tabs of section edit pages.

More information about details is available in the “**Details**” article of the “**Application interface and structure (on-line documentation)**” section.

A detail with an editable list is not a standard Creatio detail and can not be added to sections using the [Detail Wizard](#).

To add a custom detail with editable list to an existing section:

1. Create a detail object schema.
2. Create ad configure the detail schema.
3. Set up the detail in the section edit page replacing schema.
4. Set up detail fields.

Case description

Create a custom [Courier services] detail in the [Orders] section. The detail should display a list of courier services for the current order.

Source code

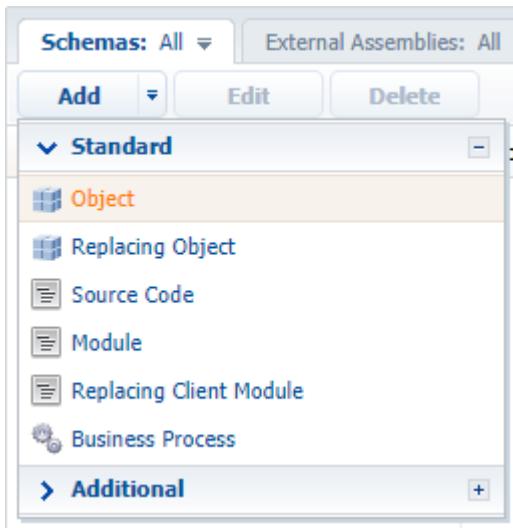
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Creating a detail object schema.

Perform the [Add] – [Object] action on the [Schemas] tab of the [Configuration] section.

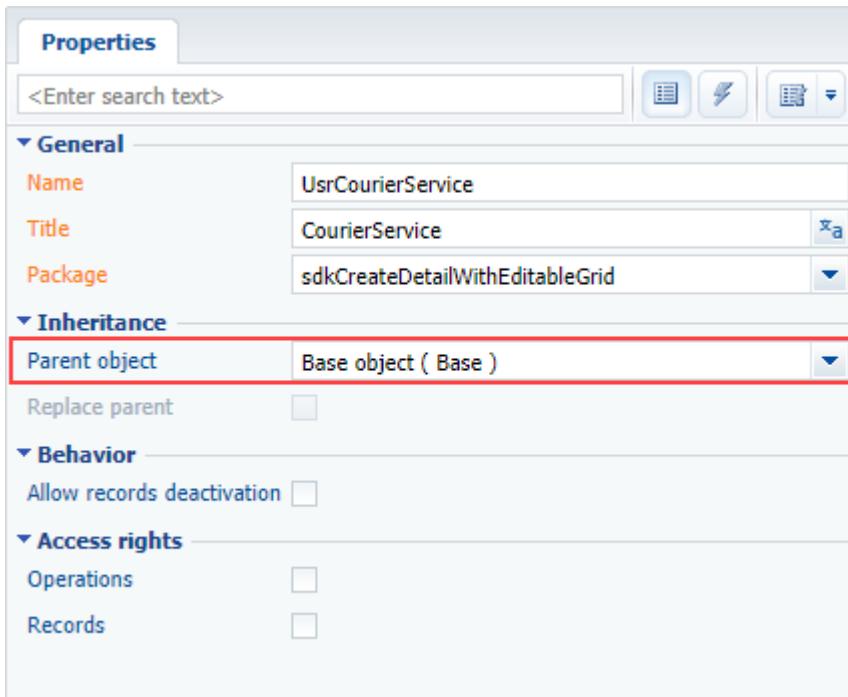
Fig. 1. Adding a detail object schema



For the created object schema (Fig. 2) specify following:

- [Name] – "UsrCourierService"
- [Title] – "CourierService"
- [Parent object] – [Base object].

Fig. 2. Setup of detail object schema properties



Add the [Order] lookup column to the object schema establishing connection with the [Orders] section and the [Account] lookup column containing the account carrying out delivery for this order. Mark both columns as

“required” to avoid adding empty records. Column setup is shown in fig. 3 and fig. 4.

Fig. 3. The [Order] column properties setup

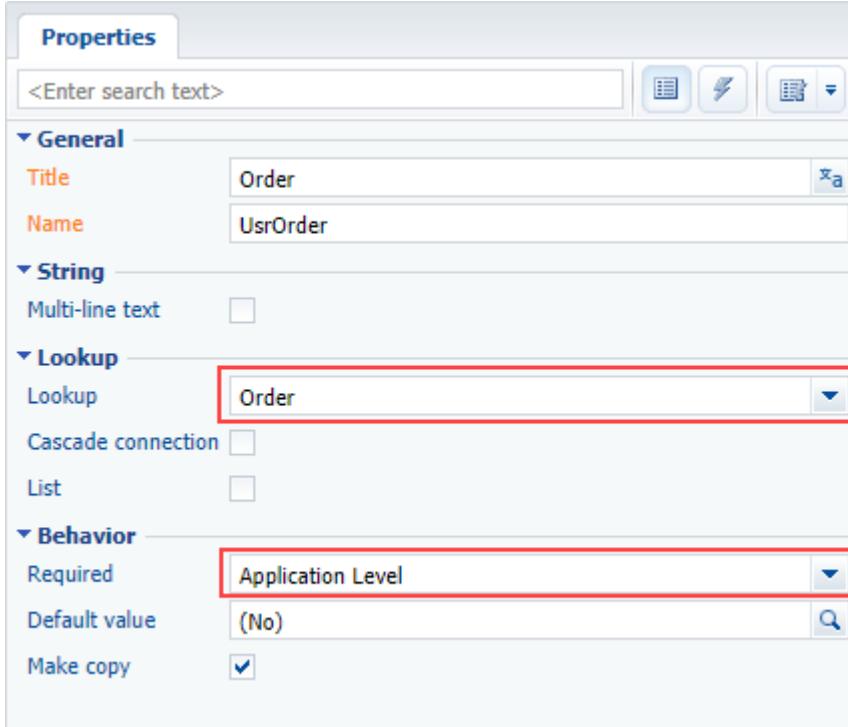
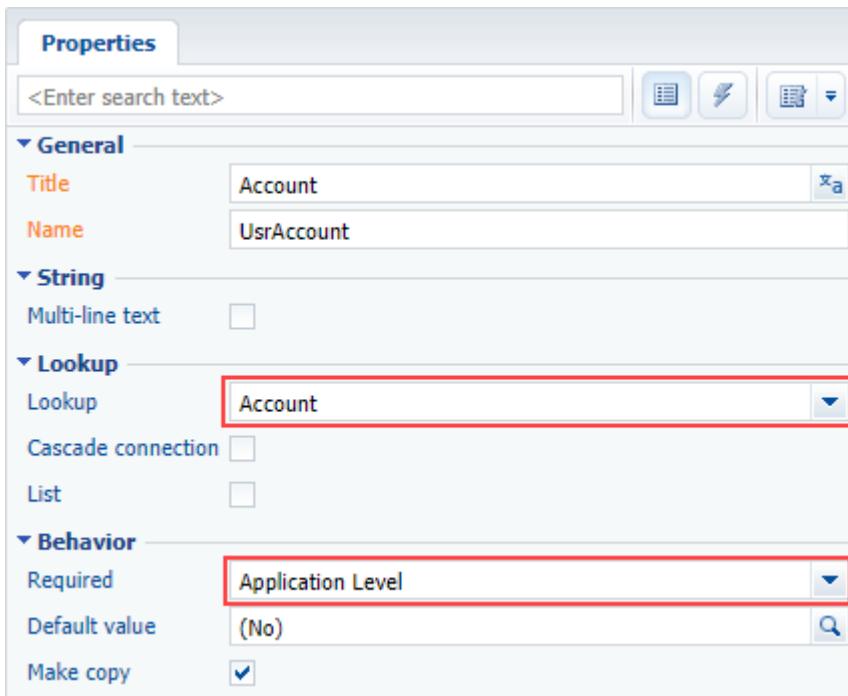


Fig. 4. The [Account] column properties setup



Save and publish the object schema.

2. Creating a detail schema.

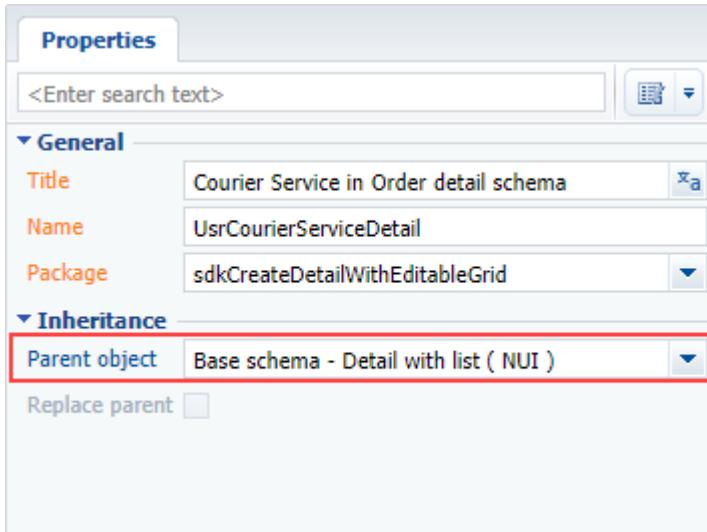
Run the [Add] – [Module] menu command on the [Schemas] tab of the [Configuration] section (fig. 1).

The created module should inherit the *BaseGridDetailV2* base detail list schema functions (specify it as the parent schema) defined in the *NUI* package.

Specify other properties (Fig. 5):

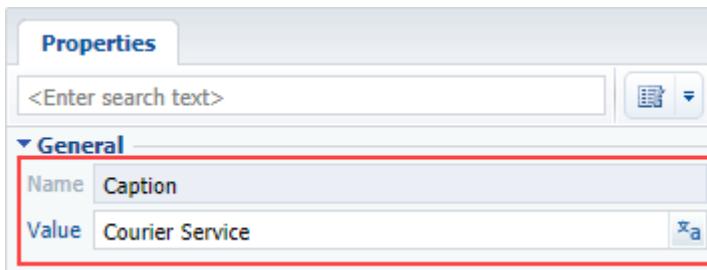
- [Name] – “UsrCourierServiceDetail”
- [Title] – “Courier Service in Order detail schema”.

Fig. 5. Detail schema properties



Set the [Courier Service] value for the [Caption] localizable string of the detail list schema (fig. 6). The [Caption] localizable string stores the detail caption, displayed on the edit page.

Fig. 6. Setting the [Caption] localizable string value



To make the list of the detail editable, make the following changes to the detail schema:

1. Add dependencies from the *ConfigurationGrid*, *ConfigurationGridGenerator*, *ConfigurationGridUtilities* modules.
2. Connect the *ConfigurationGridUtilities* mixin.
3. Set the *IsEditable* attribute value to *true*.
4. In the **diff array of modifications**, add the configuration object in which the properties are set and handler methods for detail list events are bound.

Below is the detail schema source code with comments:

```
// Defining schema and setting its dependencies from other modules.
define("UsrCourierServiceDetail", ["ConfigurationGrid", "ConfigurationGridGenerator",
    "ConfigurationGridUtilities"], function() {
    return {
        // Detail object schema name.
        entitySchemaName: "UsrCourierService",
        // Schema attribute list.
        attributes: {
            // Determines whether the editing is enabled.
            "IsEditable": {
                // Data type - logic.
                dataValueType: Terrasoft.DataValueType.BOOLEAN,
                // Attribute type - virtual column of the view model.
                type: Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
            }
        }
    }
})
```

```
// Set value.
    value: true
}
},
// Used mixins.
mixins: {
    ConfigurationGridUtilities: "Terrasoft.ConfigurationGridUtilities"
},
// Array with view model modifications.
diff: /**SCHEMA_DIFF*/[
{
    // Operation type - merging.
    "operation": "merge",
    // Name of the schema element, with which the action is performed.
    "name": "DataGrid",
    // Object, whose properties will be joined with the schema element
properties.
    "values": {
        // Class name
        "className": "Terrasoft.ConfigurationGrid",
        // View generator must generate only part of view.
        "generator": "ConfigurationGridGenerator.generatePartial",
        // Binding the edit elements configuration obtaining event
        // of the active page to handler method.
        "generateControlsConfig": {"bindTo":
"generateActiveRowControlsConfig",
            // Binding the active record changing event to handler method.
            "changeRow": {"bindTo": "changeRow"},
            // Binding the record selection cancellation event to handler
method.
            "unSelectRow": {"bindTo": "unSelectRow"},
            // Binding of the list click event to handler method.
            "onGridClick": {"bindTo": "onGridClick"},
            // Actions performed with active record.
            "activeRowActions": [
                // [Save] action setup.
                {
                    // Class name of the control element, with which the
action is connected.
                    "className": "Terrasoft.Button",
                    // Display style - transparent button.
                    "style":
this.Terrasoft.controls.ButtonEnums.style.TRANSPARENT,
                    // Tag.
                    "tag": "save",
                    // Marker value.
                    "markerValue": "save",
                    // Binding button image.
                    "imageConfig": {"bindTo": "Resources.Images.SaveIcon"}
                },
                // [Cancel] action setup.
                {
                    "className": "Terrasoft.Button",
                    "style":
this.Terrasoft.controls.ButtonEnums.style.TRANSPARENT,
                    "tag": "cancel",
                    "markerValue": "cancel",
                    "imageConfig": {"bindTo": "Resources.Images.CancelIcon"}
                },
                // [Delete] action setup.
                {
                    "className": "Terrasoft.Button",

```

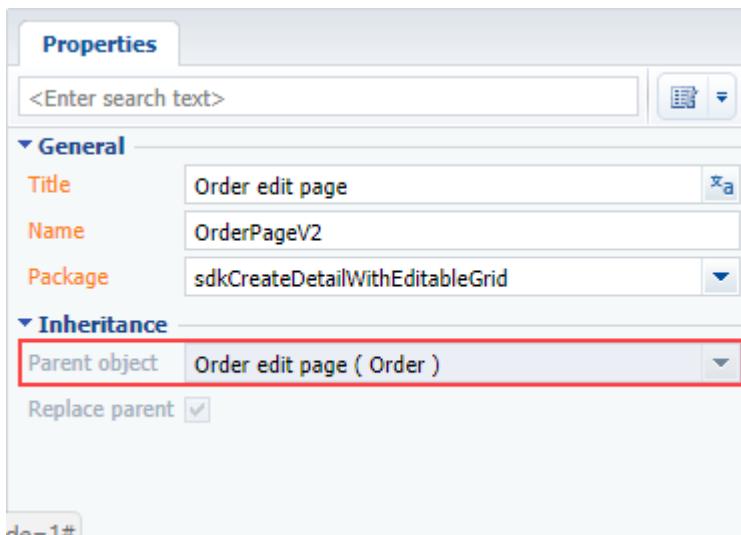
```
        "style":  
this.Terrasoft.controls.ButtonEnums.style.TRANSPARENT,  
        "tag": "remove",  
        "markerValue": "remove",  
        "imageConfig": {"bindTo": "Resources.Images.RemoveIcon"}  
    }  
],  
// Binding to method that initializes subscription to events  
// of clicking buttons in the active row.  
"initActiveRowKeyMap": {"bindTo": "initActiveRowKeyMap"},  
// Binding the active record action completion event to handler  
method.  
"activeRowAction": {"bindTo": "onActiveRowAction"},  
// Identifies whether multiple records can be selected.  
"multiSelect": {"bindTo": "MultiSelect"}  
}  
}  
]  
} /**SCHEMA_DIFF*/  
};  
});
```

Save the created detail list schema to apply the changes.

3. Setting up detail in the section edit page replacing schema.

To display the detail on the order edit page, create a client replacing module and indicate the [Order edit page] as the parent object, *OrderPageV2*) (fig. 7). Creating a replacing page is covered in the “[Creating a custom client module schema](#)” article.

Fig. 7. Properties of the order edit page replacing schema



To display the [Courier Service] detail on the [Delivery] tab of the order edit page, add the following source code. It defines a new *UsrCourierServiceDetail* detail in the *details* section and its location on the order edit page is specified in the *diff* modification array section.

```
define("OrderPageV2", [], function() {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Order",
        // List of edit page details being added.
        details: /**SCHEMA_DETAILS**/{
            // [Courier services] detail setup.
            "UsrCourierServiceDetail": {
                // Detail schema name.
                "schemaName": "UsrCourierServiceDetail",
                // ...
            }
        }
    }
})
```

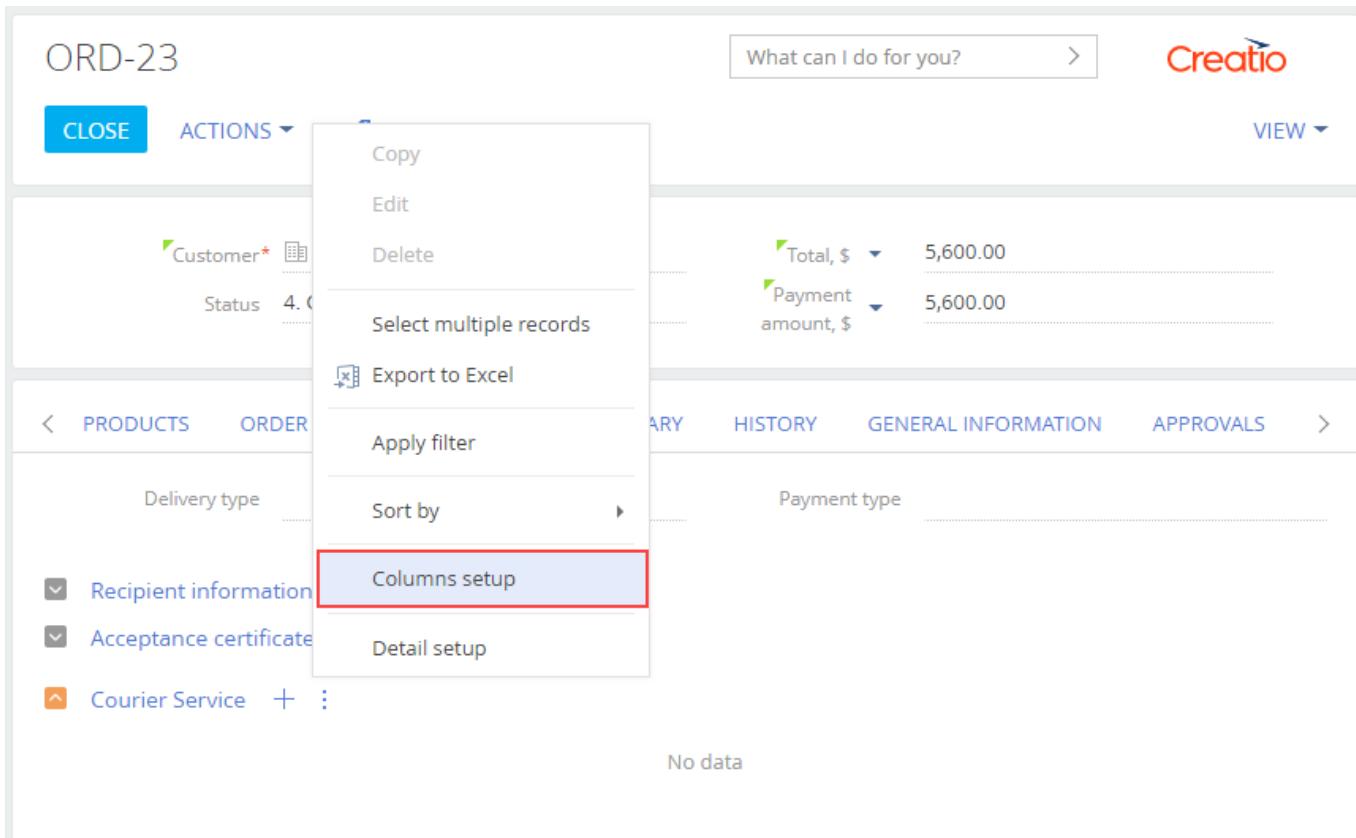
```
// Detail object schema name.
"entitySchemaName": "UsrCourierService",
// Filtering displayed contacts for current order only.
"filter": {
    // Detail object schema column.
    "detailColumn": "UsrOrder",
    // Section object schema column.
    "masterColumn": "Id"
}
}
}/**SCHEMA_DETAILS*/,
// Array with modifications.
diff: /**SCHEMA_DIFF*/[
    // Metadata for adding the [Courier services] detail.
    {
        "operation": "insert",
        // Detail name.
        "name": "UsrCourierServiceDetail",
        "values": {
            "itemType": Terrasoft.core.enums.ViewItemType.DETAIL,
            "markerValue": "added-detail"
        },
        // Containers, where the detail is located.
        // Detail is placed on the [Delivery] tab.
        "parentName": "OrderDeliveryTab",
        "propertyName": "items",
        // Index in the list of added elements.
        "index": 3
    }
]/**SCHEMA_DIFF*/
},
);
});
```

Save a replacing schema of the edit page

4. Setup of detail list columns.

At this stage, the detail is completely operational, however accounts are not displayed on the detail, because the detail list does not have displayed columns specified for it. Go to the detail menu and set up the columns to be displayed (Fig. 8)

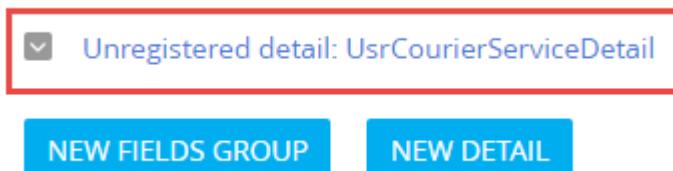
Fig. 8. Detail action menu



Detail wizard, Section wizard and details with editable lists

A detail with an editable list is not a standard Creatio detail and can not be added to sections using the [Detail Wizard](#). If you try perform the [Detail setup] action (Fig. 8), Creatio will display the “detail is not registered” message. The [Section wizard](#) displays a similar message (Fig. 9).

Fig. 9. Unregistered detail in the Section wizard



Usually details with editable lists do not require registration. However, if you still need to register such a detail for some reason, use the following SQL script:

```

DECLARE
    -- Name of the created pop-up summary view schema.
    @ClientUnitSchemaName NVARCHAR(100) = 'UsrCourierServiceDetail',
    -- Name of the object schema, to which the pop-up summary is bound.
    @EntitySchemaName NVARCHAR(100) = 'UsrCourierService',
    -- Detail name.
    @DetailCaption NVARCHAR(100) = 'Courier service'

INSERT INTO SysDetail(Caption, DetailSchemaUID, EntitySchemaUID)
VALUES (@DetailCaption,
        (SELECT TOP 1 UID
         from SysSchema
         WHERE Name = @ClientUnitSchemaName),
        (SELECT TOP 1 UId
         from SysSchema
         WHERE Name = @EntitySchemaName))

```

```
WHERE Name = @EntitySchemaName)
```

To register a custom detail, make changes to the *SysDetails* table in the Creatio database using the SQL query.

Pay high attention to creating and executing the SQL query. Executing an incorrect SQL query can damage the existing data and disrupt the system.

The detail becomes available in the corresponding lookup and displayed as registered in the section wizard after you update the page. After this you can use the Section wizard to add this detail to other tabs of the [Orders] edit page.

Fig. 9. Registered detail is displayed in the Section wizard



See also

- [Creating a detail in wizards](#)
- [Adding a detail with an editable list](#)
- [Creating a detail with selection from lookup](#)
- [Creating a custom detail with fields](#)

Creating a detail with selection from lookup

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

Introduction

Details are elements of section edit pages. Details display records bound to the current section record by a lookup field. These can be, for example, records from other sections, whose primary objects contain lookup columns linked to the primary object of the current section. Details are displayed on tabs of section edit pages.

More information about details is available in the “**Details**” article of the “**Application interface and structure (on-line documentation)**” section.

Since a detail with selection from lookup is not a standard Creatio detail, it is not enough to use [section wizard](#) to add it to the section. Below we will describe a way of adding such a detail to a section edit page.

To add a custom detail with selection from lookup to an existing section:

1. Create a detail object schema.
2. Create a detail via detail wizard and add it to the section.
3. Configure the detail schema.
4. Set up detail fields.

Case description

Create the [Acceptance certificates] custom detail on the [Delivery] tab of the [Orders] section. The detail should display the list of documents – acceptance certificates for the current order.

Source code

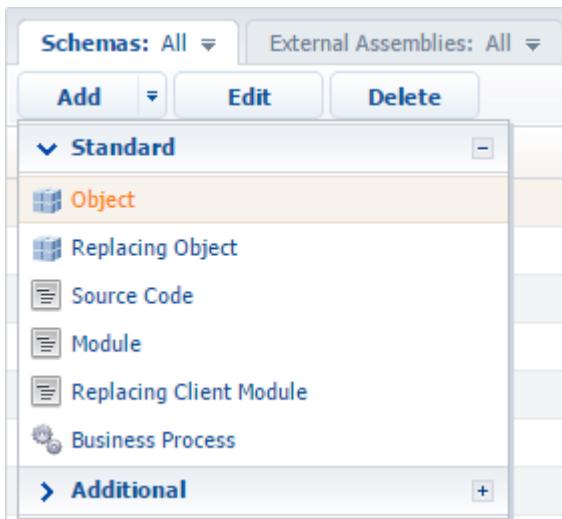
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Creating a detail object schema

Run the [Add] – [Object] menu command on the [Schemas] tab of the [Configuration] section (Fig. 1).

Fig. 1. Adding a detail object schema



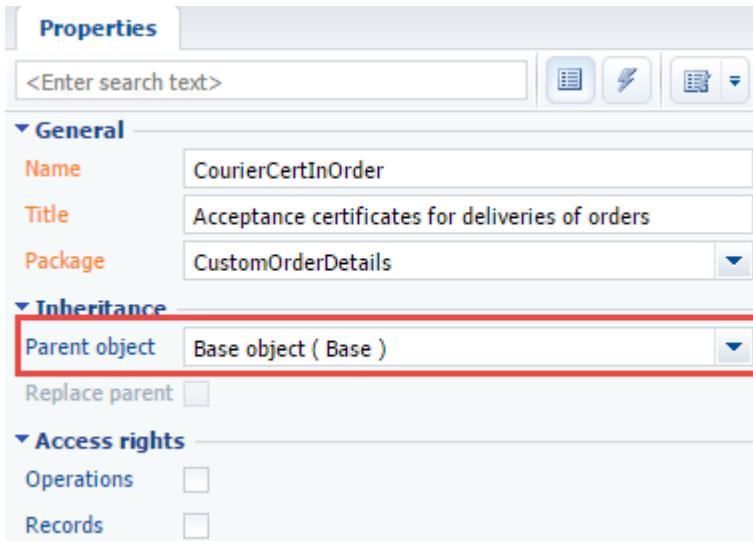
Select the “Base object” as the parent object (Fig. 2).

Object properties:

- [Name] – “UsrCourierCertInOrder”
- [Title] – “Acceptance certificates for deliveries of orders”

Find more information about object schema property setup in object designer in the **“Workspace of the Object Designer”** article.

Fig. 2. Setup of detail object schema properties



Add the [Order] lookup column to the object schema establishing connection with the [Orders] section and the [Document] lookup column containing the acceptance certificate. Mark both columns as “required” to avoid adding empty records. Column setup is shown in fig. 3 and fig. 4.

Fig. 3. The [Order] column property setup

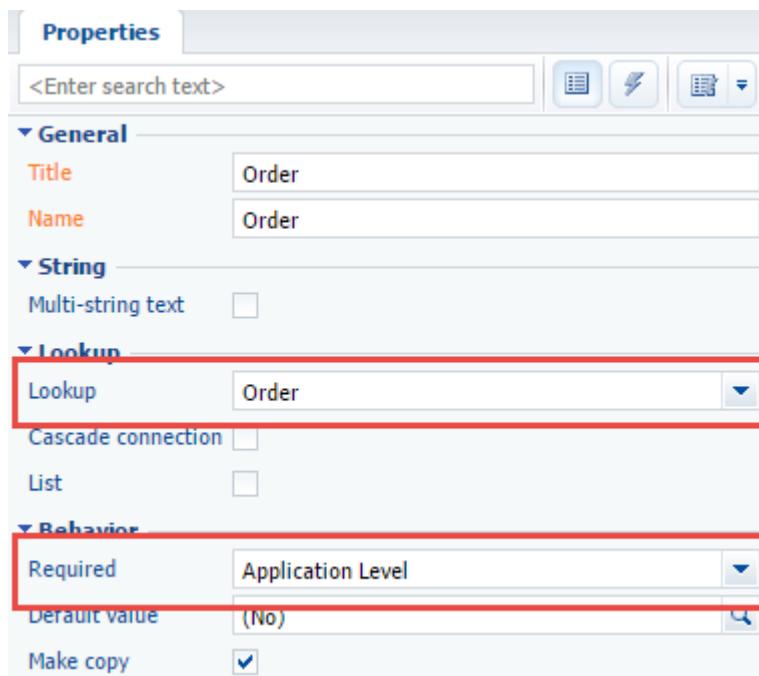
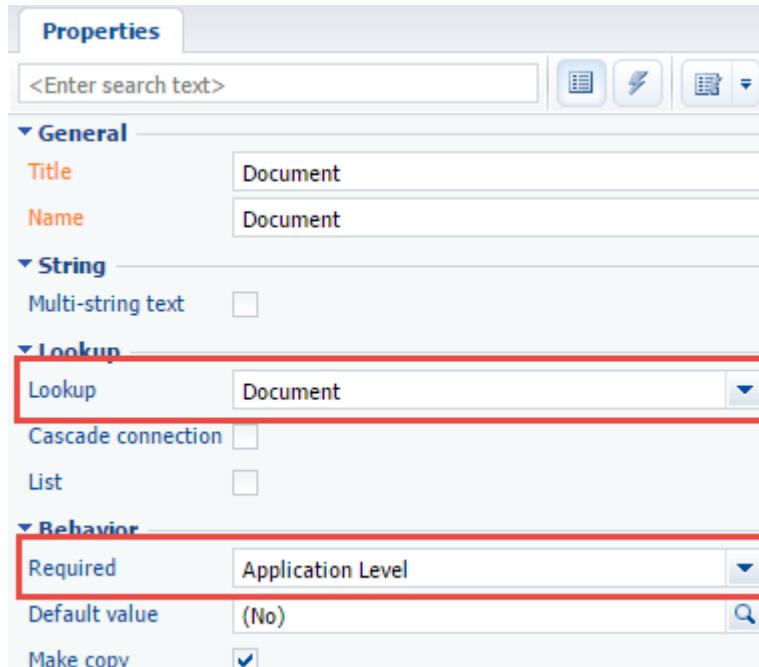


Fig. 4. The [Document] column property setup



Save and publish the object schema.

2. Creating a detail and adding it to the section

Via [detail wizard](#) create the [Acceptance certificates] detail (Fig. 5). Add the detail to the [Delivery] tab of the [Orders] section page via section wizard (Fig. 6).

Fig. 5. Creating the detail

Select properties for detail:

Title* Acceptance certificates

Object* Acceptance certificates for deliveries of orders

Acceptance certificates for deliveries of orders

Fig. 6. Adding the detail to the section

Detail settings

Detail* Acceptance certificates

Caption on the page* Acceptance certificates

Detail column* Order

Object column* Id

SAVE CANCEL

As a result, the *UsrSchemaDetail* detail schema and the *OrderPageV2* section replacing schema will be created in the development package.

We recommend changing the name of the detail schema to a more unique name, since the auto generated name can be duplicated in a different development package. We replaced the name for *UsrCourierCertDetail* in the current case.

Change the detail schema name for a new one in the *OrderPageV2* section replacing schema.

3. Configuring the detail

Change the source code of the created detail schema:

1. Add dependency from the *ConfigurationEnums* module.
2. Add the *onDocumentInsert()*, *onCardSaved()* event handler-methods, the *openDocumentLookup()* method of calling the modal lookup window and additional data control methods.
3. Add the necessary configuration objects to the *diff* modification array.

The Terrasoft.EntitySchemaQuery class is used for executing database queries in the current case. You can learn more about using EntitySchemaQuery in the **“The use of EntitySchemaQuery for creation of queries in database (on-line documentation)**” article. To simplify the case, we blocked the detail menu options that enable editing and copying detail list records.

Source code of the detail schema:

```
// Defining the shcema and setting up its dependencies from other modules.
define("UsrCourierCertDetail", ["ConfigurationEnums"],
    function(configurationEnums) {
        return {
            // Name of the detail object schema.
            entitySchemaName: "UsrCourierCertInOrder",
            // Detail schema methods.
            methods: {
                //Returns columns selected by query.
                getGridDataColumns: function() {
                    return [
                        "Id": {path: "Id"},
                        "Document": {path: "UsrDocument"},
                        "Document.Number": {path: "UsrDocument.Number"}
                    ];
                },
                //Configures and displays modal lookup window.
                openDocumentLookup: function() {
                    //Configuration object
                    var config = {
                        // Name of the object schema whose records will be displayed
in the lookup.
                        entitySchemaName: "Document",
                        // Multiple selection option.
                        multiSelect: true,
                        // Columns used in the lookup, e.g., for sorting.
                        columns: ["Number", "Date", "Type"]
                    };
                    var OrderId = this.get("MasterRecordId");
                    if (this.Ext.isEmpty(OrderId)) {
                        return;
                    }
                    // The [EntitySchemaQuery] class instance.
                    var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
                        // Setting up the root schema.
                        rootSchemaName: this.entitySchemaName
                    });
                    // Adding the [Id] column.
                    esq.addColumn("Id");
                    // Adding the [Id] column for the [Document] schema.
                    esq.addColumn("Document.Id", "DocumentId");
                    // Creating and adding filters to query collection.
                    esq.filters.add("filterOrder",
this.Terrasoft.createColumnFilterWithParameter(
                        this.Terrasoft.ComparisonType.EQUAL, "UsrOrder", OrderId));
                    // Receiving the whole record collection and its display in the
modal lookup window.
                    esq.getEntityCollection(function(result) {
                        var existsDocumentsCollection = [];
                        if (result.success) {
                            result.collection.each(function(item) {
                                existsDocumentsCollection.push(item.get("DocumentId"));
                            });
                        }
                    });
                }
            }
        }
    }
);
```

```
        }
        // Adding filter to the configuration object.
        if (existsDocumentsCollection.length > 0) {
            var existsFilter =
this.Terrasoft.createColumnInFilterWithParameters("Id",
                                                existsDocumentsCollection);
            existsFilter.comparisonType =
this.Terrasoft.ComparisonType.NOT_EQUAL;
            existsFilter.Name = "existsFilter";
            config.filters = existsFilter;
        }
        // Call of the modal lookup window
        this.openLookup(config, this.addCallBack, this);
    },
},
// Event handler of saving the edit page.
onCardSaved: function() {
    this.openDocumentLookup();
},
//Opens the document lookup if the order edit page has been saved.
addRecord: function() {
    var masterCardState = this.sandbox.publish("GetCardState", null,
[this.sandbox.id]);
    var isNewRecord = (masterCardState.state ===
configurationEnums.CardStateV2.ADD ||
    masterCardState.state === configurationEnums.CardStateV2.COPY);
    if (isNewRecord === true) {
        var args = {
            isSilent: true,
            messageTags: [this.sandbox.id]
        };
        this.sandbox.publish("SaveRecord", args, [this.sandbox.id]);
        return;
    }
    this.openDocumentLookup();
},
// Adding the selected products.
addCallBack: function(args) {
    // Class instance of the BatchQuery package query.
    var bq = this.Ext.create("Terrasoft.BatchQuery");
    var OrderId = this.get("MasterRecordId");
    // Collection of the selected documents from the lookup.
    this.selectedRows = args.selectedRows.getItems();
    // Collection passed over to query.
    this.selectedItems = [];
    // Copying the necessary data.
    this.selectedRows.forEach(function(item) {
        item.OrderId = OrderId;
        item.DocumentId = item.value;
        bq.add(this.getDocumentInsertQuery(item));
        this.selectedItems.push(item.value);
    }, this);
    // Executing the package query if it is not empty.
    if (bq.queries.length) {
        this.showBodyMask.call(this);
        bq.execute(this.onDocumentInsert, this);
    }
},
```

```
//Returns query for adding the current object.
getDocumentInsertQuery: function(item) {
    var insert = Ext.create("Terrasoft.InsertQuery", {
        rootSchemaName: this.entitySchemaName
    });
    insert.setParameterValue("UsrOrder", item.OrderId,
this.Terrasoft.DataValueType.GUID);
    insert.setParameterValue("UsrDocument", item.DocumentId,
this.Terrasoft.DataValueType.GUID);
    return insert;
},

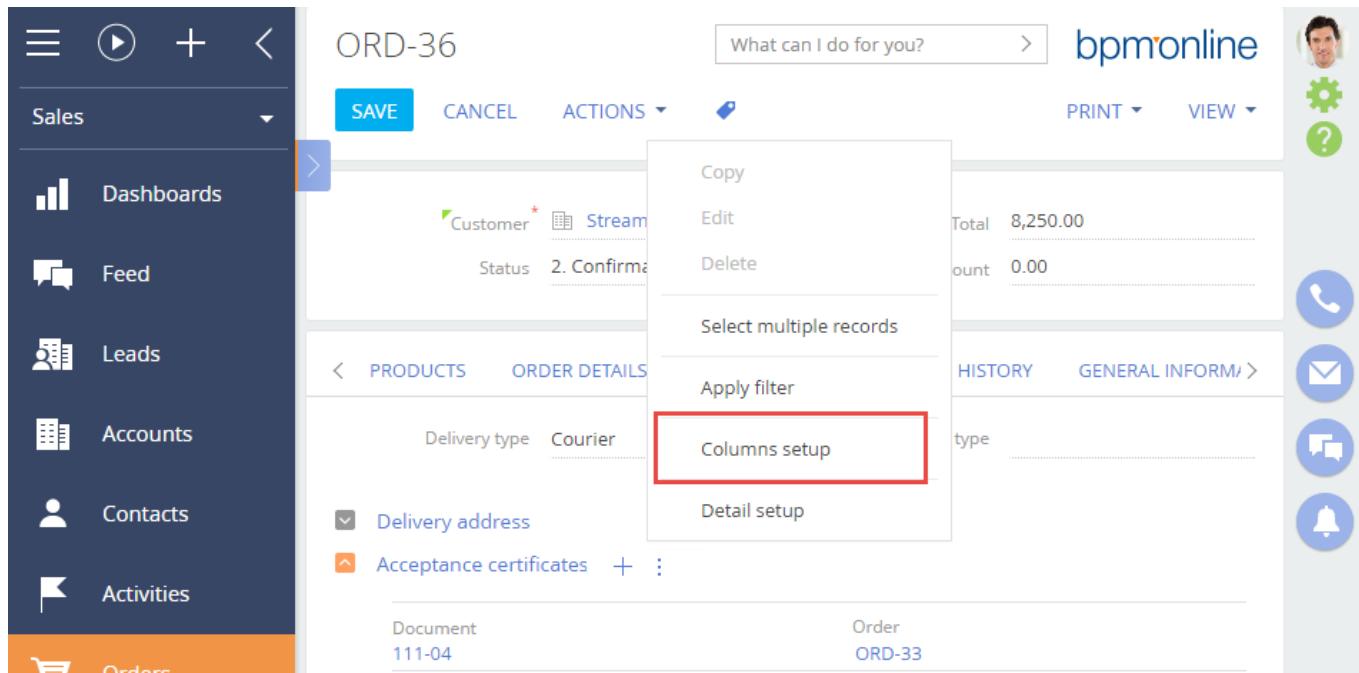
//Method called when adding records to the detail record list.
onDocumentInsert: function(response) {
    this.hideBodyMask.call(this);
    this.beforeLoadGridData();
    var filterCollection = [];
    response.queryResults.forEach(function(item) {
        filterCollection.push(item.id);
    });
    var esq = Ext.create("Terrasoft.EntitySchemaQuery", {
        rootSchemaName: this.entitySchemaName
    });
    this.initQueryColumns(esq);
    esq.filters.add("recordId",
Terrasoft.createColumnInFilterWithParameters("Id", filterCollection));
    esq.getEntityCollection(function(response) {
        this.afterLoadGridData();
        if (response.success) {
            var responseCollection = response.collection;
            this.prepareResponseCollection(responseCollection);
            this.getGridData().loadAll(responseCollection);
        }
    }, this);
},
// Method called when deleting records from the detail record list.
deleteRecords: function() {
    var selectedRows = this.getSelectedItems();
    if (selectedRows.length > 0) {
        this.set("SelectedRows", selectedRows);
        this.callParent(arguments);
    }
},
// Hide the [Copy] menu option.
getCopyRecordMenuItem: Terrasoft.emptyFn,
// Hide the [Edit] menu option.
getEditRecordMenuItem: Terrasoft.emptyFn,
// Returns the default filter column name.
getFilterDefaultColumnName: function() {
    return "UsrDocument";
}
},
// Modification array.
diff: /**SCHEMA_DIFF*/[
{
    // Operation type - merging.
    "operation": "merge",
    // Name of the schema element under operation.
    "name": "DataGrid",
    // The object, whose properties will be combined with the schema
}
```

```
element properties.  
        "values": {  
            "rowDataItemMarkerColumnName": "UsrDocument"  
        }  
    },  
    {  
        // Operation type - merging.  
        "operation": "merge",  
        // Name of the schema element under operation.  
        "name": "AddRecordButton",  
        // The object, whose properties will be combined with the schema  
        element properties.  
        "values": {  
            "visible": {"bindTo": "getToolsVisible"}  
        }  
    }  
} /*SCHEMA_DIFF*/  
};  
}  
};
```

4. Setting up detail list columns

At this stage the detail is completely operable but contact records are not displayed on the detail as the display columns are not specified. Call the detail action menu and set up the column display (fig. 7).

Fig. 7. Detail action menu



As a result, the new detail will enable adding records from the document lookup via the modal window (Fig. 8).

Fig. 8. Case result

Number	Type	Date	Company
085-02	Regulation	9/5/2016 3:46 PM	Milestone Consulting
088-01	Minutes	9/5/2016 3:46 PM	Novelty
087-01	Minutes	9/5/2016 3:46 PM	Apex Solutions
103-02	Regulation	9/5/2016 3:46 PM	Axiom
094-01	Minutes	9/5/2016 3:46 PM	Axiom
094-02	Regulation	9/5/2016 3:46 PM	Axiom
103-02	Incoming document	9/5/2016 3:46 PM	Alpha Business
099-02	Incoming document	9/5/2016 3:46 PM	Infocom
106-01	Incoming document	9/5/2016 3:46 PM	Durable Industries
In.203-22 (sample)	Incoming document	9/15/2016 8:41 PM	Accom (sample)
120-01	Incoming document	9/5/2016 3:46 PM	Clearsoft
119-01	Regulation	9/5/2016 3:46 PM	Apex Solutions

See also

- [Creating a detail in wizards](#)
- [Adding a detail with an editable list](#)
- [Adding an edit page detail](#)
- [Creating a custom detail with fields](#)

Adding multiple records to a detail

Beginner Easy Medium **Advanced**

Introduction

Normally, a detail allows you to only add one record. Adding multiple entries to a detail is done through the `LookupMultiAddMixin` mixin.

A mixin is a class designed to extend the functions of other classes. Learn more about mixins in the “[Mixins. The “mixins” property](#)”.

The `LookupMultiAddMixin` is designed to extend the detail schemas. It provides an opportunity to add multiple lookup records to the detail at the same time.

The sequence of adding the multiple records selection functionality to a detail:

1. Create a replacing schema of the detail.
2. Use mixin methods instead of base detail methods.

Case description

Implement the possibility of multiple records selection in the [Contacts] detail on the [Sales] section records edit page.

Source code

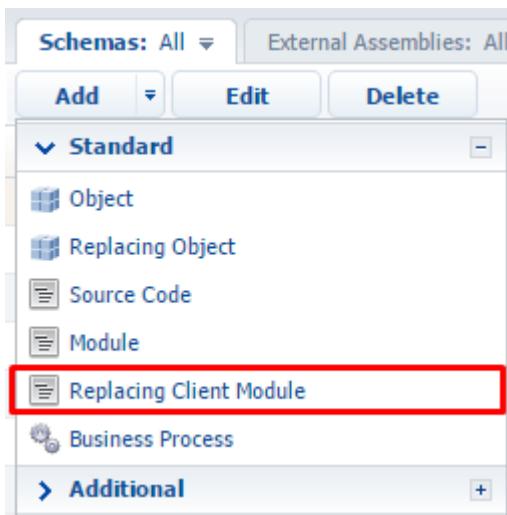
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Create a replacing view model schema of a detail

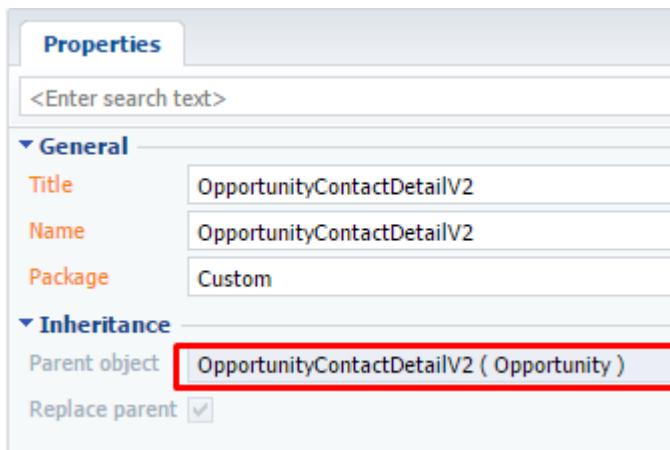
Create a detail replacing schema in the development package (Fig. 1). The procedure for creating a replacing schema is covered in the “**Creating a custom client module schema**”.

Fig. 1. Adding replacing client module



Select the *OpportunityContactDetailV2* schema as the parent object (Fig. 2).

Fig. 2. Properties of the replacing client module



2. Use mixin methods instead of base detail methods.

To use the *LookupMultiAddMixin* mixin in a schema, add it to the *mixins* property and initialize it in the pre-defined *init()* schema method. Learn more about pre-defining the *init()* method in the “**Modular development principles in Creatio**” article.

To implement the required functionality, you need to pre-define the “add” button displaying methods (*getAddRecordButtonVisible()*), saving detail page methods (*onCardSaved()*), and adding a detail record methods (*addRecord()*).

Saving a detail page and adding record methods include the method of calling the help window for multiple selection `openLookupWithMultiSelect()` and the associated method `getMultiSelectLookupConfig()` which configures the help window, is used. The description and parameters of these methods are given in Table 1.

Table 1. Methods for calling and configuring the help window

Methods.	Description
<code>openLookupWithMultiSelect(isNeedCheckOfNew)</code>	Opens a lookup window with a multiple selection option. The <code>isNeedCheckOfNew {bool}</code> parameter indicates the need to check if the record is new.
<code>getMultiSelectLookupConfig()</code>	Returns the configuration object for the help window. Object properties: <code>rootEntitySchemaName</code> – root object schema; <code>rootColumnName</code> – a connected column that indicates the root schema record; <code>relatedEntitySchemaName</code> – connected schema; <code>relatedColumnName</code> – a column that indicates the connected schema record.

In this case the help window will pull data from the *OpportunityContact* table using the *Opportunity* and *Contact* columns.

Source code of the detail schema:

```
define("OpportunityContactDetailV2", ["LookupMultiAddMixin"], function() {
    return {
        mixins: {
            // Connecting the mixin to the schema.
            LookupMultiAddMixin: "Terrasoft.LookupMultiAddMixin"
        },
        methods: {
            // Overriding the base method for initializing the schema.
            init: function() {
                this.callParent(arguments);
                // Initializing the mixin.
                this.mixins.LookupMultiAddMixin.init.call(this);
            },
            // Overriding the base method for displaying the "Add" button.
            getAddRecordButtonVisible: function() {
                // Displaying the "add" button if the detail is maximized, even if the
                detail edit page is not implemented.
                return this.getToolsVisible();
            },
            // Overriding the base method.
            // The save event handler for the detail edit page.
            onCardSaved: function() {
                // Opens the window for multiple record selection.
                this.openLookupWithMultiSelect();
            },
            // Overriding the base method of adding a detail record.
            addRecord: function() {
                // Opens the window for multiple records selection.
                this.openLookupWithMultiSelect(true);
            },
            // A method that returns a window configuration object.
            getMultiSelectLookupConfig: function() {
                return {
                    // Root schema - [Opportunities].
                    rootEntitySchemaName: "Opportunity",
                    ...
```

```
// Root schema column.
rootColumnName: "Opportunity",
// Connected schema - [Contact].
relatedEntitySchemaName: "Contact",
// Root schema column.
relatedColumnName: "Contact"
};

}

};

});
```

After saving the schema and reloading the application page, the user will be able to select multiple records from the lookup (Fig. 4) by clicking the “add” detail record button (Fig. 3). All selected records will be added to the [Contacts] detail of the [Opportunities] section record edit page (Fig. 5).

Fig. 3. Adding multiple records to a detail



Fig. 4. Selecting necessary records from a lookup

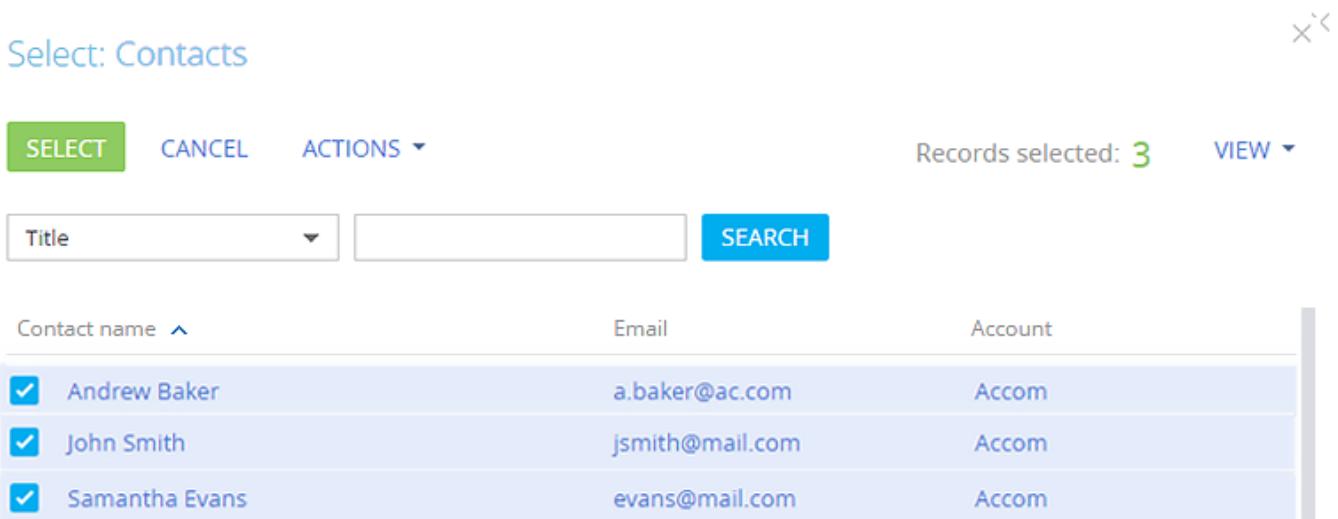
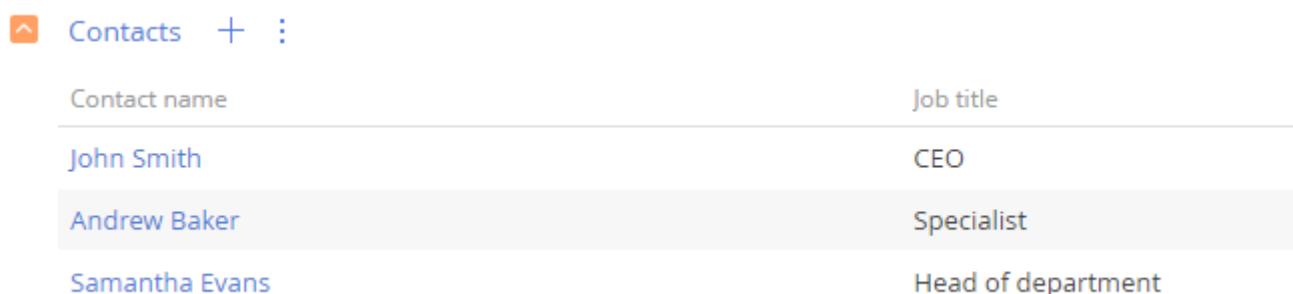


Fig. 5. Case result: All selected records are added to a detail



Creating a custom detail with fields

[Beginner](#)

[Easy](#)

[Medium](#)

[Advanced](#)

Introduction

A detail with fields can include multiple field groups. The base detail with fields is implemented in the *BaseFieldsDetail* schema of the *BaseFinance* package, which is available in Creatio bank customer journey, bank sales and lending. The detail record view model is implemented in the *BaseFieldRowViewModel* schema.

Base detail with fields enables you to:

- Add detail records without saving a page.
- Work with a detail like you would with an edit page.
- Use the base field validation with the ability to add a custom one.
- Add a virtual record.
- Expand record behavior logic.

A custom detail with fields can be **created** with the help of the base detail (a custom detail schema should be inherited from the base detail schema).

Case description

Implement a custom detail with fields for document registration. The detail should be populated with records that include the document's [Number] and [Series] fields. The detail should be located on the [History] tab of the contact edit page.

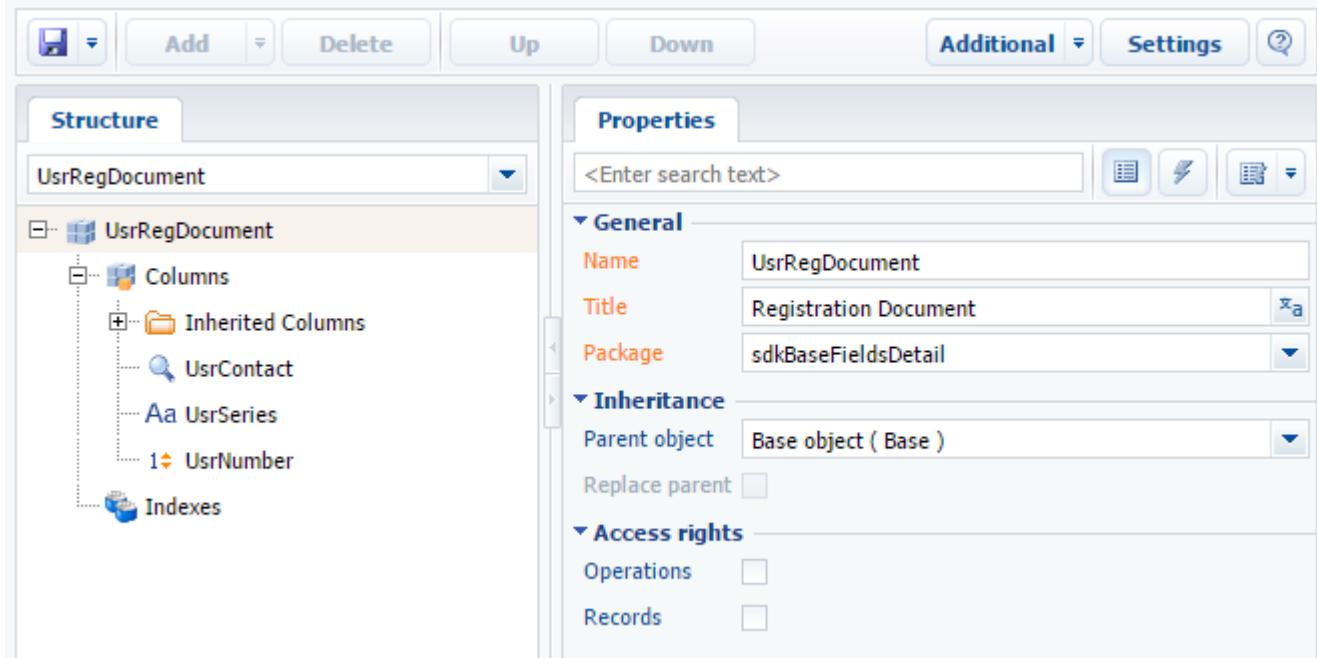
Case implementation algorithm

1. Create a detail object schema

Create a new object schema in a custom package with the following property values:

- [Title] – “Registration document”.
- [Name] – “UsrRegDocument”.
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down list.
- [Parent object] – “Base object”, implemented in the Base package.

Fig. 1. Object schema properties



Add three columns in the object structure. Column properties are listed in Table 1. Learn more about adding object schema columns in the “[Creating the entity schema](#)” article.

Table 1. — Column properties of the UsrRegDocument detail object schema

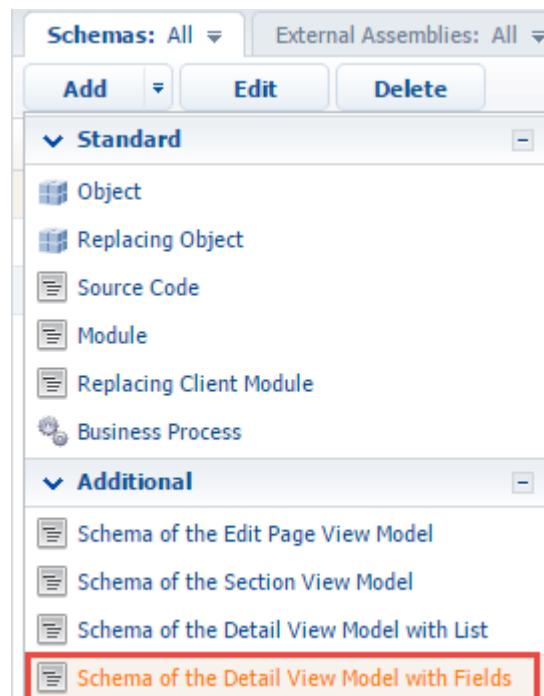
Title	Name	Data Type
Contact	UsrContact	Lookup
Series	UsrSeries	Text (50 characters)
Number	UsrNumber	Integer

Publish the schema to apply changes.

2. Create a view model schema for the custom detail with fields.

Add a custom schema([Schema of the Detail View Model with Fields]) in a custom package (Fig. 2).

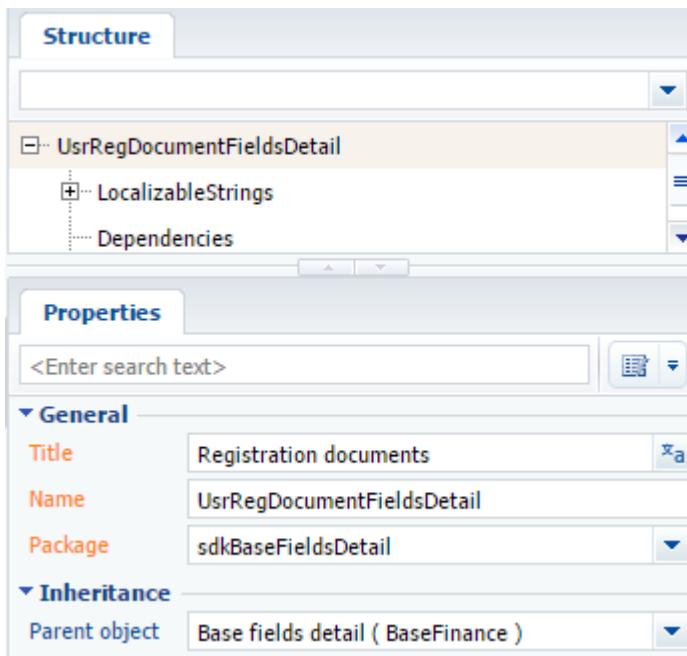
Fig. 2. Adding a custom view model schema for the custom detail with fields



Property values for the created schema (Fig. 3):

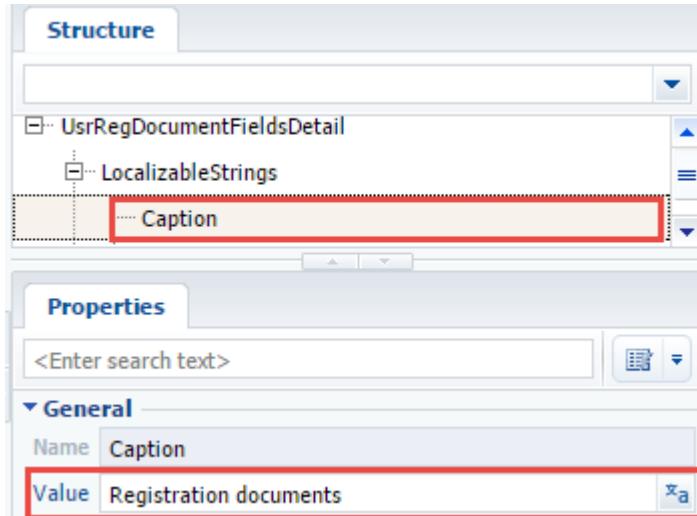
- [Title] – “Registration documents”.
- [Name] – “UsrRegDocumentFieldsDetail”.
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down list.
- [Parent object] – "Base fields detail", implemented in the BaseFinance package.

Fig. 3. UsrRegDocumentFieldsDetail client schema properties



Assign the “Registration documents” value to the localizable [Caption] string of the [Value] property.

Fig. 4. Localizable string properties



Create a module description and redefine the base `getDisplayColumns()` method, which returns column names that are displayed as detail fields. By default, this method returns all required columns, as well as the column with the set [Displayed value] checkbox in the object schema.

Schema source code:

```
define("UsrRegDocumentFieldsDetail", [],
  function() {
    return {
      entitySchemaName: "UsrRegDocument",
      diff: /**SCHEMA_DIFF*/ [], /**SCHEMA_DIFF*/
      methods: {
        getDisplayColumns: function() {
          return ["UsrSeries", "UsrNumber"];
        }
      }
    };
  });
});
```

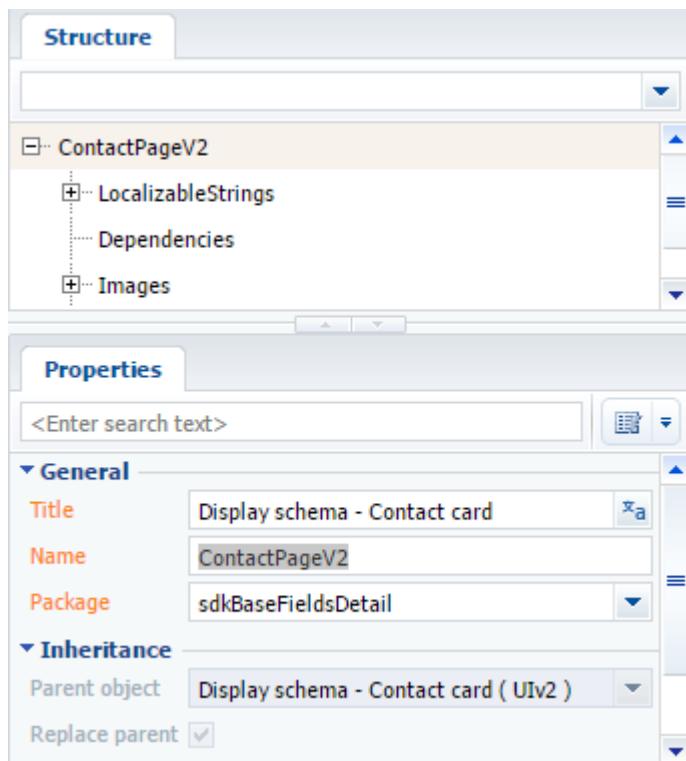
Save the schema to apply changes.

3. Create a replacing schema for the edit page

To do this, create a **replacing schema** of the contact edit page (*ContactPageV2*). Main replacing schema properties (Fig. 5):

- [Title] – “Display schema - Contact card”.
- [Name] – “ContactPageV2”.
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down list.
- [Parent object] – “Display schema - Contact card”, implemented in the *UIv2* package.

Fig. 5. The ContactPageV2 replacing schema properties



Make the following adjustments to the source code:

- **Create a module description**
- Add a detail in the **“details” property**
- Add a configuration object of the detail's view model to the **“diff” modification array**.

Schema source code:

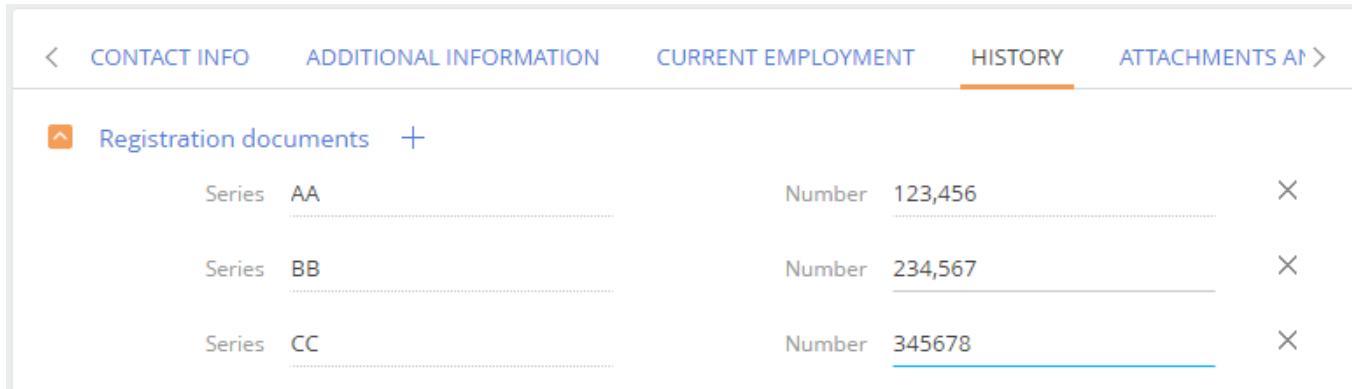
```
define("ContactPageV2", [], function() {
    return {
        entitySchemaName: "Contact",
        details: /**SCHEMA_DETAILS*/ {
            // Adding a field with details.
            "UsrRegDocumentFieldsDetail": {
                // Name of the custom detail schema.
                "schemaName": "UsrRegDocumentFieldsDetail",
                // Filtering current contact's record details.
                "filter": {
                    // Detail object column.
                    "detailColumn": "UsrContact",
                    // Contact's Id column.
                }
            }
        }
    }
});
```

```
        "masterColumn": "Id"
    }
}
} /**SCHEMA_DETAILS*/,
diff: /**SCHEMA_DIFF*/ [
// Adding a new element.
"operation": "insert",
// Element name.
"name": "UsrRegDocumentFieldsDetail",
// Value configuration object.
"values": {
    // Element type.
    "itemType": Terrasoft.ViewItemType.DETAIL
},
// Container element name.
"parentName": "HistoryTab",
// Property name of the container element with the collection of nested
elements.
"propertyName": "items",
// The index of the element added to the collection.
"index": 0
}] /**SCHEMA_DIFF*/
};
});
```

Save the schema to apply changes.

The [History] tab of the contact edit page should now have the custom detail with the [Registration documents] fields (Fig. 6).

Fig. 6. Case result



Register the detail with fields (similarly to the **detail with the editable grid**) in the system to make it visible in detail and section wizards.

Advanced settings of a custom detail with fields

Beginner Easy Medium **Advanced**

Introduction

A detail with fields can include multiple field groups. The base detail with fields is implemented in the *BaseFieldsDetail* schema of the *BaseFinance* package, which is available in Creatio bank customer journey, bank sales and lending. The detail record view model is implemented in the *BaseFieldRowViewModel* schema.

The process of creating a custom detail with fields is described in a separate **Creating a custom detail with fields**.

Adding custom styles

Case description

Redefine the field signature style for a detail implemented in the “[Creating a custom detail with fields](#)” article. The field signatures should be displayed in blue.

You can override the basic CSS style classes for displaying detail records by using the `getLeftRowContainerWrapClass()` and `getRightRowContainerWrapClass()` methods.

Case implementation algorithm

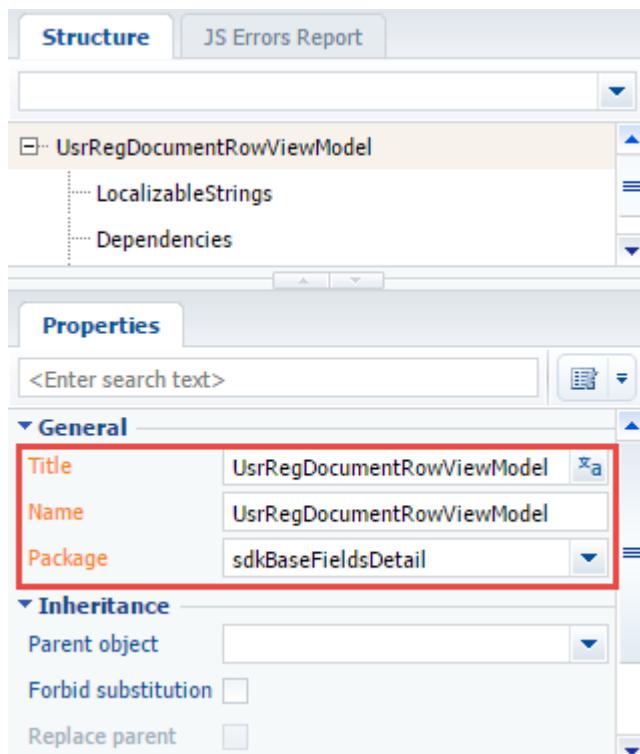
1. Create a module schema and define record view styles

You can not set the styles in a view model schema of the edit page. It is necessary to create a new module schema, define the styles and add the created module to module dependencies of a detail.

Create a new module schema in a custom package with the following property values:

- [Title] – “UsrRegDocumentRowViewModel”.
- [Name] – “UsrRegDocumentRowViewModel”.
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down list.

Fig. 1. UsrRegDocumentFieldsDetail custom schema properties



Create a module description, and define the `Terrasoft.configuration.UsrRegDocumentRowViewModel` class which is inherited from `Terrasoft.configuration.BaseFieldRowViewModel`.

The source code of the schema:

```
define("UsrRegDocumentRowViewModel", ["BaseFieldRowViewModel"], function() {
    Ext.define("Terrasoft.configuration.UsrRegDocumentRowViewModel", {
        extend: "Terrasoft.BaseFieldRowViewModel",
        alternateClassName: "Terrasoft.UsrRegDocumentRowViewModel"
    });
    return Terrasoft.UsrRegDocumentRowViewModel;
});
```

Define the CSS view classes for the correct display of detail records. To do this, add the following CSS classes to the LESS tab of the module designer:

```
.reg-document-left-row-container {  
    .t-label {  
        color: blue;  
    }  
}  
.field-detail-row {  
    width: 100%;  
    display: inline-flex;  
    margin-bottom: 10px;  
  
    .field-detail-row-left {  
        display: flex;  
        flex-wrap: wrap;  
        width: 100%;  
  
        .control-width-15 {  
            min-width: 300px;  
            width: 50%;  
            margin-bottom: 5px;  
        }  
        .control-width-15:only-child {  
            width: 100% !important;  
        }  
    }  
    .field-detail-row-left.singlecolumn {  
        width: 50%;  
    }  
}
```

Save the schema to apply changes.

2. Modifying a replacing view model schema of a detail

To use the created module and its styles in a detail schema, add it to the dependency of the module defined in the detail schema.

Additionally, add the following methods to a detail schema module:

- `getRowViewModelClassName()` – returns the name of the record view model class to the detail.
- `getLeftRowContainerWrapClass()` – returns the string array with CSS class names, used to generate the views of record signature field containers.

The source code of the modified schema:

```
define("UsrRegDocumentFieldsDetail", ["UsrRegDocumentRowViewModel",  
"css!UsrRegDocumentRowViewModel"],  
function() {  
    return {  
        entitySchemaName: "UsrRegDocument",  
        diff: /**SCHEMA_DIFF*/ [], /**SCHEMA_DIFF*/  
        methods: {  
            getDisplayColumns: function() {  
                return ["UsrSeries", "UsrNumber"];  
            },  
            getRowViewModelClassName: function() {  
                return "Terrasoft.UsrRegDocumentRowViewModel";  
            },  
            getLeftRowContainerWrapClass: function() {  
                return ["reg-document-left-row-container", "field-detail-row"];  
            }  
        }  
    };  
});
```

```
} );
```

Save the schema to apply changes.

On the [History] tab of the contact edit page, the detail names will be displayed in blue (Fig. 2).

Fig. 2. Case result

Registration documents		+
Series	AA	Number 123,456
Series	BB	Number 234,567
Series	CC	Number 345,678

Adding additional custom logic for detail records

Case description

Add the field validation to the [Number] field of the detail, implemented in the “[Creating a custom detail with fields](#)” article. The field value can not be negative.

Case implementation algorithm

1. Adding a localizable string with the error message

In the module designer, add the localizable string on the [Structure] tab of the opened *UsrReqDocumentRowViewModel* schema with the following property values

(Fig. 3):

- [Name] – “NumberMustBeGreaterThenZeroMessage”.
 - [Value] – “Number must be greater than zero!”.

Fig. 3. Case result

The screenshot shows the 'Structure' view in a software application. The tree view displays the following hierarchy:

- UserRegDocumentRowViewModel
 - LocalizableStrings
 - NumberMustBeGreaterThanZeroMessage

The 'NumberMustBeGreaterThanZeroMessage' node is highlighted with a yellow background, indicating it is selected. Below the tree view, there are standard navigation buttons for moving up and down the list.

A localized string is a schema resource. In order for its values to appear in the client part of the application, add the

UsrRegDocumentRowViewModelResources resource module to the dependencies of the *UsrRegDocumentRowViewModel* module.

Save the schema to apply changes.

2. Adding the validation program logic

Add the following methods to the *UsrRegDocumentRowViewModel* module to implement the validation program logic:

- *validateNumberMoreThenZero()* – contains the validation logic of the field value.
- *setValidationConfig()* – connects the [Number] column and the *validateNumberMoreThenZero()* validation method.
- *Init()* – an overridden base method that calls the base logic and the *setValidationConfig()* method.

Source code of the modified schema:

```
define("UsrRegDocumentRowViewModel", ["UsrRegDocumentRowViewModelResources",
"BaseFieldRowViewModel"],
function(resources) {
    Ext.define("Terrasoft.configuration.UsrRegDocumentRowViewModel", {
        extend: "Terrasoft.BaseFieldRowViewModel",
        alternateClassName: "Terrasoft.UsrRegDocumentRowViewModel",
        validateNumberMoreThenZero: function(columnValue) {
            var invalidMessage;
            if (columnValue < 0) {
                invalidMessage =
resources.localizableStrings.NumberMustBeGreaterThanZeroMessage;
            }
            return {
                fullInvalidMessage: invalidMessage,
                invalidMessage: invalidMessage
            };
        },
        setValidationConfig: function() {
            this.addColumnValidator("UsrNumber",
this.validateNumberMoreThenZero);
        },
        init: function() {
            this.callParent(arguments);
            this.setValidationConfig();
        }
    });
    return Terrasoft.UsrRegDocumentRowViewModel;
});
```

Save the schema to apply changes.

When a negative value is entered in the [Number] field on the [History] tab of the contact edit page, a warning message will be displayed (Fig. 4).

Fig. 4. Case result

Series	Number
AA	-1
BB	234,567
CC	345,678

Adding a virtual record

When a detail is loaded, adding virtual records enables you to display a field edit card immediately, without pressing the [Add] button.

That requires defining the `useVirtualRecord()` method (returns `true`) in the `UsrRegDocumentFieldsDetail` detail schema:

```
useVirtualRecord: function() {
    return true;
}
```

When opening a tab with a detail, a virtual record will be displayed (Fig. 5).

Fig. 5. Displaying a virtual record

Adding the [Attachments] detail

Beginner Easy Medium **Advanced**

Introduction

The [Attachments] detail is designed for storing files and links to web resources and knowledge base articles related to a section record. The detail is available in all Creatio sections (see: "[Attachments](#)"). The main functionality of the detail is implemented in the `FileDetailV2` schema of the `UIv2` package.

To add a detail to the edit page of a custom section record, do the following:

1. Create a regular detail in the detail wizard using the object schema of the `[Section object name]File` section (see: "[Creating a new section](#)".)
2. Change the parent for the detail list schema.
3. In the wizard, add the detail to the the edit page of a custom section record.
4. Perform additional detail configurations.

Case description

Add the [Attachments] details to the record edit page of the custom [Photos] section. All schemas of the [Photos]

section and the [Attachments] details should be stored in a custom package (for example, `sdkDetailAttachment`).

To create a section, use the section wizard (see: “[Creating a new section](#)” and “[Section wizard](#)”).

If the development needs to be carried out in a **custom package**, it needs to be specified in the [Current package] system setting. Otherwise, the wizard will save the changes to the [Custom] package.

Case implementation algorithm

1. Create a detail, using the [Section object name]File section object schema.

As a result (Fig. 1), a number of client modules and object schemas will be created in the custom package. The name of the schema of the main section object is *UsrPhotos* (Fig. 2). The schema name of the section object that you want to use for the details is *UsrPhotosFile*.

Fig. 1. The [Photos] section properties in the section wizard

Photos: General section properties

The screenshot shows the 'Photos: General section properties' dialog. At the top, there are buttons for 'SAVE' (green), 'CANCEL' (blue), and navigation arrows. Below the buttons, tabs for 'SECTION' (selected), 'PAGE', 'BUSINESS RULES', and 'CASES' are visible. The main area is titled 'Select basic properties for section:' and contains the following fields:

- Title**: * Photos
- Code**: * UsrPhotos
- Menu icon**: A blue square icon containing a white user profile symbol.

Below this, a section titled 'Page settings:' contains two radio button options:

- One page for all records
- Multiple pages

Fig. 2. Schemas created by the section wizard in the user package

Name ¹	Package	Title ²	Database Upda...
UsrPhotos	sdkDetailAttachment	Photos	<input type="checkbox"/>
UsrPhotos1Page	sdkDetailAttachment	Card schema: "Photos"	<input type="checkbox"/>
UsrPhotos1Section	sdkDetailAttachment	Section schema: "Photos"	<input type="checkbox"/>
UsrPhotosFile	sdkDetailAttachment	Photos attachment	<input type="checkbox"/>
UsrPhotosFolder	sdkDetailAttachment	Photos folder	<input type="checkbox"/>
UsrPhotosInFolder	sdkDetailAttachment	Photos in Folder	<input type="checkbox"/>
UsrPhotosInTag	sdkDetailAttachment	Photos section record tag	<input type="checkbox"/>
UsrPhotosTag	sdkDetailAttachment	Photos section tag	<input type="checkbox"/>

Creating details in a wizard is described in more detail in the “**Creating a detail in wizards**” article. Choose the detail title and the [Photos attachment] object as the default one in the detail wizard (Fig. 3). The transition to the second step in the wizard is optional.

Fig. 3. Detail properties in the wizard

New detail: Detail

SAVE CANCEL < DETAIL PAGE BUSINESS RULES >

Title * Photos attachment

Object * Photos attachment

As a result, the custom package will have a schema of the detail list client module and a schema of the detail edit page (Fig. 4).

Fig. 4. Schemas created by the section wizard in the user package

Schemas: All	External Assemblies: All	SQL Scripts: All	Data: All	Package Dependencies
Add	Edit	Delete		
Name	Package	Title	Database Update Required	
UsrPhotos	sdkDetailAttachment	Photos	<input type="checkbox"/>	
UsrPhotos1Page	sdkDetailAttachment	Card schema: "Photos"	<input type="checkbox"/>	
UsrPhotos1Section	sdkDetailAttachment	Section schema: "Photos"	<input type="checkbox"/>	
UsrPhotosFile	sdkDetailAttachment	Photos attachment	<input type="checkbox"/>	
UsrPhotosFolder	sdkDetailAttachment	Photos folder	<input type="checkbox"/>	
UsrPhotosInFolder	sdkDetailAttachment	Photos in Folder	<input type="checkbox"/>	
UsrPhotosInTag	sdkDetailAttachment	Photos section record tag	<input type="checkbox"/>	
UsrPhotosTag	sdkDetailAttachment	Photos section tag	<input type="checkbox"/>	
UsrSchema3Detail	sdkDetailAttachment	Detail schema: "Photos attachment"	<input type="checkbox"/>	
UsrUsrPhotosFile1Page	sdkDetailAttachment	Card schema: "Photos attachment"	<input type="checkbox"/>	

Schema names are generated automatically and may be different from those shown on Fig. 4.

2. Change the parent for the detail list schema.

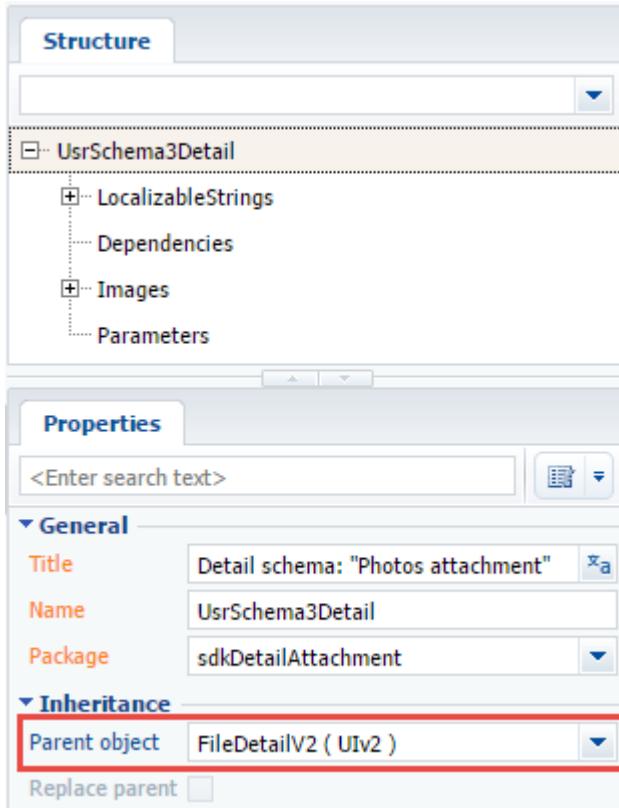
The detail with **an edit page** is created. The [Base schema - Detail with list] schema of the *NUI* package is the parent object of the client module schema of the *UsrSchema3Detail* detail list (Fig. 5).

Fig. 5. A default parent object

General	
Title	Detail schema: "Photos attachment"
Name	UsrSchema3Detail
Package	sdkDetailAttachment
Inheritance	
Parent object	Base schema - Detail with list (NUI)
Replace parent	<input type="checkbox"/>

To implement the [Files and Links] detail functionality in the custom detail, specify the *FileDialogV2* schema as the parent object of the *UsrSchema3Detail* schema (Fig. 6).

Fig. 6. The parent object – the FileDetailV2 schema



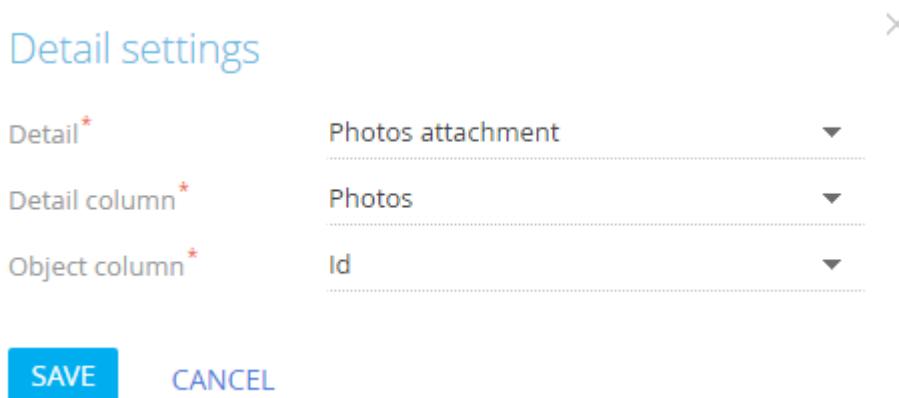
The procedure of specifying a parent object is covered in the “[Creating a custom client module schema](#)”.

Save the schema to apply changes.

3. In the wizard, add the detail to the the edit page of a custom section record.

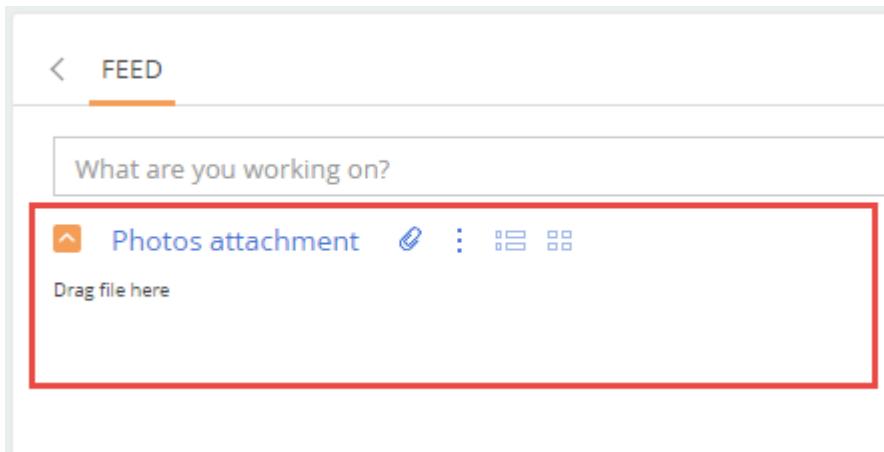
Adding a detail to the record edit page using the detail wizard is described in the “[Creating a detail in wizards](#)” article. When you add a detail in the wizard, configure the link between the detail columns and the main section object (Fig. 7).

Fig. 7. Configuring the link between the detail columns and the main section object



Upon refreshing, the detail will be displayed on a record edit page (Fig. 8).

Fig. 8. A detail on an edit page



You can add details to any edit page tab. Additionally, you can create a separate [[Attachments](#)] tab for the detail.

4. Perform additional detail configurations

After completing the previous steps, the detail is fully functional, but looks different from the [Attachments] detail of the base application sections. To make the custom detail look similar to the standard one, you need to define CSS styles for it.

The client module schema of the *UsrSchema3Detail* detail list is a schema of the view model. Defining styles in it is impossible. It is necessary to create a new module schema, define the styles and add the created module to module dependencies of a detail.

Create a new module schema in a custom package with the following property values:

- [Title] – “UsrSchema3DetailCSS”.
- [Name] – “UsrSchema3DetailCSS”.
- [Package] – “sdkDetailAttachment”.

Add the following CSS selectors to the LESS tab of the module designer:

```
div[id*="UsrSchema3Detail"] {  
    .grid-status-message-empty {  
        display: none;  
    }  
    .grid-empty > .grid-bottom-spinner-space {  
        height: 5px;  
    }  
    .dropzone {  
        height: 35px;  
        width: 100%;  
        border: 1px dashed #999999;  
        text-align: center;  
        line-height: 35px;  
    }  
    .dropzone-hover {  
        border: 1px dashed #4b7fc7;  
    }  
    .DragAndDropLabel {  
        font-size: 1.8em;  
        color: rgb(110, 110, 112);  
    }  
}  
  
div[data-item-marker*="added-detail"] {  
    div[data-item-marker*="tiled"], div[data-item-marker*="listed"] {  
        .entity-image-class {  
            width: 165px;  
        }  
    }  
}
```

```
.entity-image-container-class {
    float: right;
    width: 128px;
    height: 128px;
    text-align: center;
    line-height: 128px;
}
.entity-image-view-class {
    max-width: 128px;
    max-height: 128px;
    vertical-align: middle;
}
.images-list-class {
    min-height: 0.5em;
}
.images-list-class > .selectable {
    margin-right: 10px;
    display: inline-block;
}
.entity-label {
    display: block;
    max-width: 128px;
    margin-bottom: 10px;
    text-align: center;
}
.entity-link-container-class > a {
    font-size: 1.4em;
    line-height: 1.5em;
    display: block;
    max-width: 128px;
    margin-bottom: 10px;
    color: #444;
    text-decoration: none;
    text-overflow: ellipsis;
    overflow: hidden;
    white-space: nowrap;
}
.entity-link-container-class > a:hover {
    color: #0e84cf;
}
.entity-link-container-class {
    float: right;
    width: 128px;
    text-align: center;
}
.select-entity-container-class {
    float: left;
    width: 2em;
}
.listed-mode-button {
    border-top-right-radius: 1px;
    border-bottom-right-radius: 1px;
}
.tiled-mode-button {
    border-top-left-radius: 1px;
    border-bottom-left-radius: 1px;
}
.tiled-mode-button, .listed-mode-button {
    padding-left: 0.308em;
    padding-right: 0.462em;
}
```

```
.button-pressed {
    background: #fff;

    .t-btn-image {
        background-position: 0 16px !important;
    }
}

div[data-item-marker*="tiled"] {
    .tiled-mode-button {
        .button-pressed;
    }
}

div[data-item-marker*="listed"] {
    .listed-mode-button {
        .button-pressed;
    }
}

}
```

The styles defined in the source code above almost completely coincide with the styles defined in the schema of the *FileDetailCssModule* module in the *UIv2* package. They are intended for the *FileDetailV2* schema used as the parent object.

You can not use the *FileDetailCssModule* module directly, because the markers and identifiers of the HTML elements of the standard detail and the detail created by the wizard are different.

Save the schema to apply changes.

To use the created module and its styles in a detail *UsrSchema3Detail* detail schema, add it to the dependency of the module defined in the detail schema.

The source code of the modified schema:

```
div[id*="UsrSchema3Detail"] {
    .grid-status-message-empty {
        display: none;
    }

    .grid-empty > .grid-bottom-spinner-space {
        height: 5px;
    }

    .dropzone {
        height: 35px;
        width: 100%;
        border: 1px dashed #999999;
        text-align: center;
        line-height: 35px;
    }

    .dropzone-hover {
        border: 1px dashed #4b7fc7;
    }

    .DragAndDropLabel {
        font-size: 1.8em;
        color: rgb(110, 110, 112);
    }
}

div[data-item-marker*="added-detail"] {
    div[data-item-marker*="tiled"], div[data-item-marker*="listed"] {
        .entity-image-class {
            width: 165px;
        }

        .entity-image-container-class {
            float: right;
            width: 128px;
        }
    }
}
```

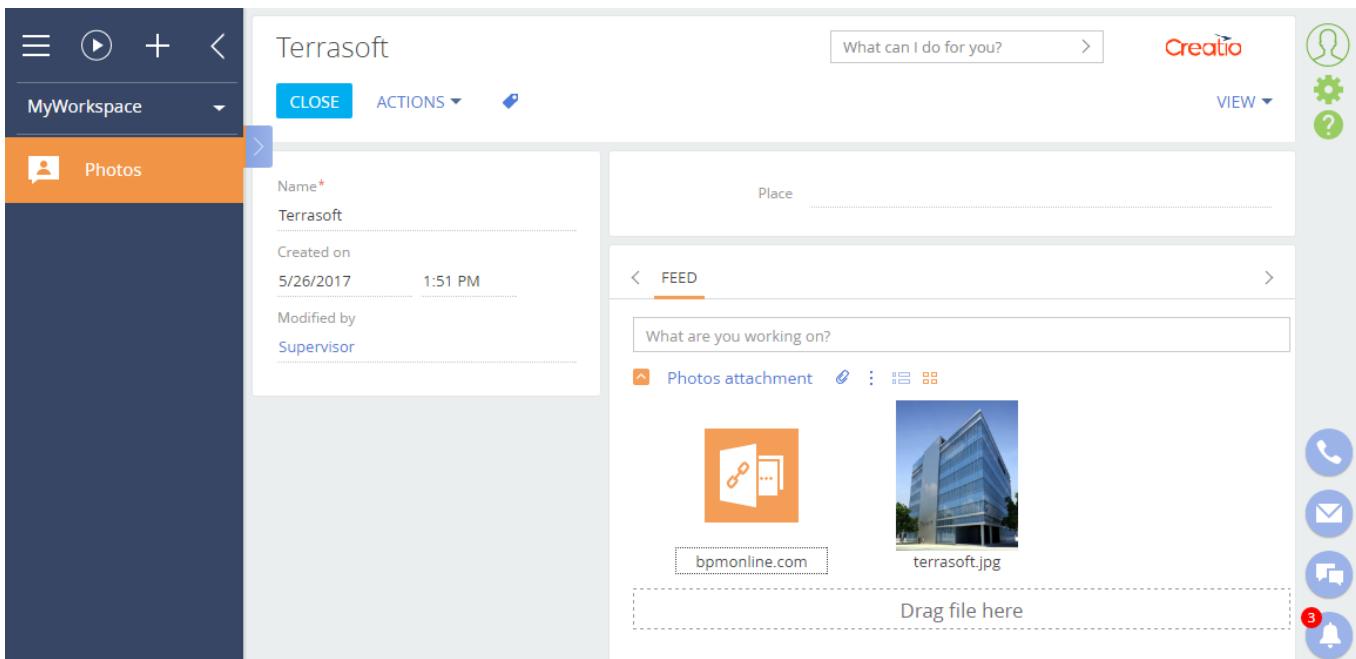
```
height: 128px;
text-align: center;
line-height: 128px;
}
.entity-image-view-class {
max-width: 128px;
max-height: 128px;
vertical-align: middle;
}
.images-list-class {
min-height: 0.5em;
}
.images-list-class > .selectable {
margin-right: 10px;
display: inline-block;
}
.entity-label {
display: block;
max-width: 128px;
margin-bottom: 10px;
text-align: center;
}
.entity-link-container-class > a {
font-size: 1.4em;
line-height: 1.5em;
display: block;
max-width: 128px;
margin-bottom: 10px;
color: #444;
text-decoration: none;
text-overflow: ellipsis;
overflow: hidden;
white-space: nowrap;
}
.entity-link-container-class > a:hover {
color: #0e84cf;
}
.entity-link-container-class {
float: right;
width: 128px;
text-align: center;
}
.select-entity-container-class {
float: left;
width: 2em;
}
.listed-mode-button {
border-top-right-radius: 1px;
border-bottom-right-radius: 1px;
}
.tiled-mode-button {
border-top-left-radius: 1px;
border-bottom-left-radius: 1px;
}
.tiled-mode-button, .listed-mode-button {
padding-left: 0.308em;
padding-right: 0.462em;
}
.button-pressed {
background: #fff;
```

```
.t-btn-image {  
    background-position: 0 16px !important;  
}  
}  
div[data-item-marker*="tiled"] {  
    .tiled-mode-button {  
        .button-pressed;  
    }  
}  
}  
div[data-item-marker*="listed"] {  
    .listed-mode-button {  
        .button-pressed;  
    }  
}  
}
```

Save the schema to apply changes.

As a result, the edit page of the custom [Photos] section will display an [Attachments] detail, almost identical to the base one (Fig. 9).

Fig. 9. Case result



Displaying additional columns on the [Attachments] tab

Beginner

Easy

Medium

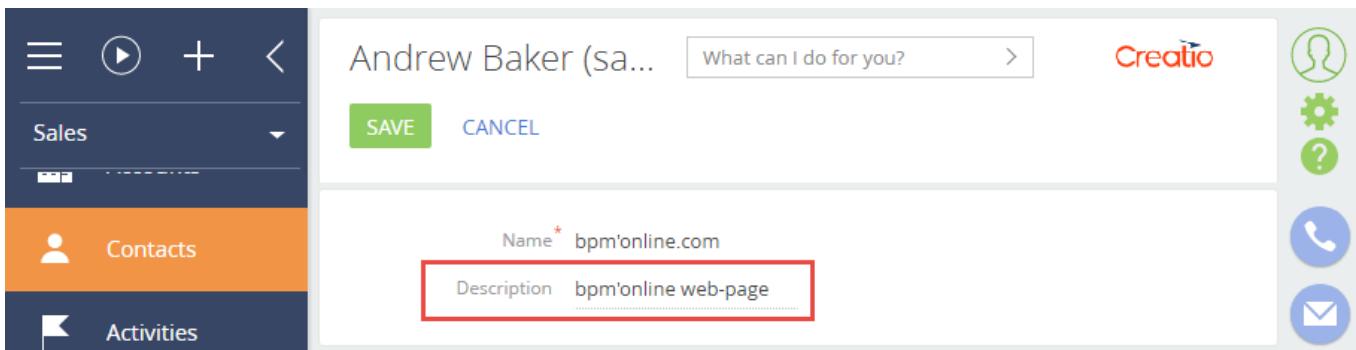
Advanced

Introduction

The [Attachments] detail is designed for storing files and links to web resources and knowledge base articles related to the section record. The detail is available in all Creatio section (see: “[How to work with attachments and notes](#)”). The main functionality of the detail is implemented in the *FileDetailV2* scheme of the *UIv2* package.

By default, the [Attachments] detail is set to have only the [Name] and [Version] columns in the list view. The [Description] column is available while adding a new link. However, it is not displayed in the list view.

Fig. 1. The [Description] field



The tile view of the [Attachments] detail only features the [Name] column and a file or a link.

Use the columns setup page to set up the detail's list columns (see: "[Setting up columns](#)"). However, the [Attachments] detail does not have this configuration option by default.

To add this configuration option, do the following:

- Create a replacing schema of the *FileDetailV2* detail.
- Call the method of opening the column configuration page in the replacing schema.

Case description

Add a new [Columns setup] command to the [Actions] menu for the [Attachments] detail.

Case implementation algorithm

1. Replace the *FileDetailV2* detail schema

The procedure for creating a replacing client schema is covered in the "[Creating a custom client module schema](#)". Create a replacement schema with the following properties in a custom package:

- [Title]— “FileDetailV2”.
- [Name] — “FileDetailV2”.
- [Package] – the schema will be placed in this package after publishing. By default, this property contains the name of the package selected prior to creating a schema. It can be populated with any value from the drop-down list.
- [Parent object] – “FileDetailV2”.

2. Call the method of opening the column configuration page in the replacing schema.

To do this, go to the the **module description** in the source code of the replacement schema, and redefine the base *getGridSettingsMenuItem()* method, which returns the detail menu item, associated with the call to the base *openGridSettings()* method defined in the *GridUtilities* mixin.

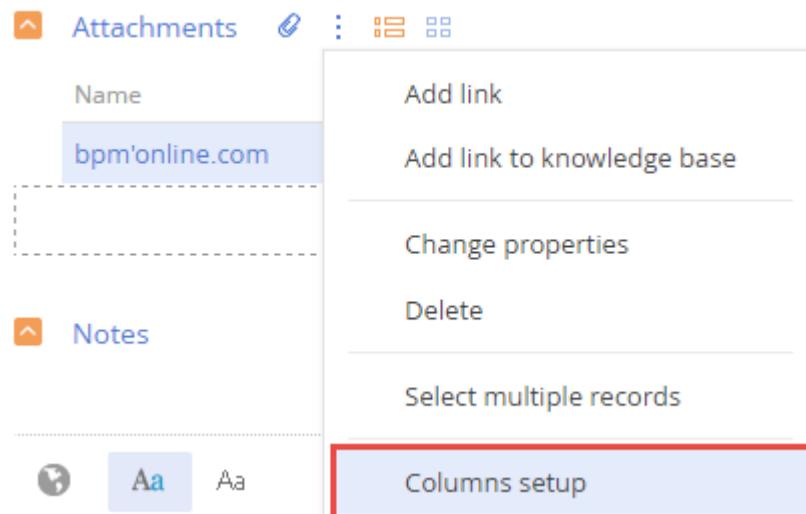
Schema source code:

```
define("FileDetailV2", [], function() {
    return {
        methods: {
            getGridSettingsMenuItem: function() {
                return this.getButtonMenuItem({
                    Caption: {"bindTo": "Resources.Strings.SetupGridMenuCaption"},
                    Click: {"bindTo": "openGridSettings"}
                });
            }
        }
    };
});
```

Save the schema to apply changes.

As a result, a new [Columns setup] command is displayed in the [Actions] menu (Fig. 2).

Fig. 2. The [Columns setup] command



This command enables the user to configure the list column view, and add the [Description] column to the detail.

Fig. 3. The column configuration page

The screenshot shows the 'List setup' configuration page. It has a 'Tile view' and 'List view' radio button, with 'List view' selected. Below is a preview of three columns: 'Name' (light blue), 'Description' (yellow), and 'Version' (light blue). At the bottom are buttons for 'SET', 'MOVE' (with left and right arrows), and 'DELETE'.

After saving the settings in the detail list view, the column will be displayed (Fig. 4).

Fig. 4. Case result

The screenshot shows the 'Attachments' detail view. It has tabs for 'CONTACT INFO', 'CURRENT EMPLOYMENT', 'HISTORY', and 'COMMUNICATION CHANNELS'. Under 'Attachments', there is a table with columns 'Name', 'Description', and 'Version'. One row shows 'bpm'online.com' with 'bpm'online web-page' in the 'Description' column and '1' in the 'Version' column. At the bottom is a dashed box placeholder for 'Drag file here'.

How to hide menu commands of the detail with list

Beginner

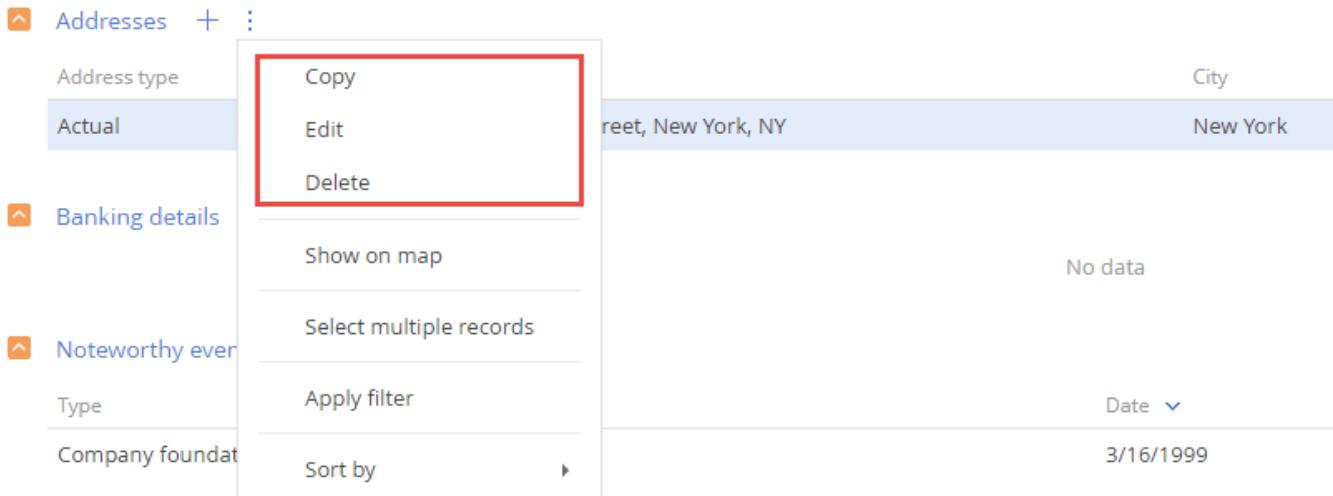
Easy

Medium

Advanced

The [Copy], [Edit], and [Delete] commands in a detail menu are used to manage records in the detail list (Fig. 1).

Fig. 1. The [Addresses] detail menu



To hide menu detail commands:

1. Create a replacing schema of the detail list. For example, for the [Addresses] detail of the account edit page it will be the [Account addresses detail]. The procedure for creating a replacing client schema is covered in the “**Creating a custom client module schema**” article.

2. Add the following source code to the schema:

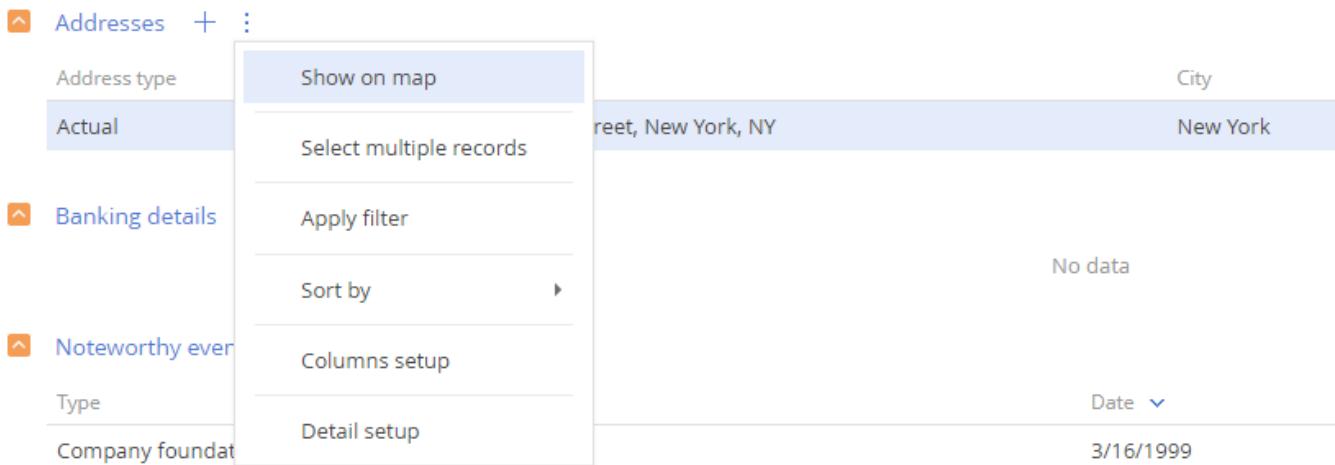
```
define("AccountAddressDetailV2", [], function() {
    return {
        entitySchemaName: "AccountAddress",
        methods: {
            // Disabling the [Copy] command
            getCopyRecordMenuItem: Terrasoft.emptyFn,
            // Disabling the [Edit] command
            getEditRecordMenuItem: Terrasoft.emptyFn,
            // Disabling the [Delete] command
            getDeleteRecordMenuItem: Terrasoft.emptyFn
        },
        diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
    };
});
```

3. Save the changes.

4. Refresh the browser page.

As a result, commands will be disabled in the detail menu (Fig.2).

Fig. 2. [Addresses] detail menu without menu commands



Deleting a detail

Beginner Easy **Medium** Advanced

Introduction

To delete a custom detail in Creatio, you need access to the system configuration tools and its database.

Before you delete a custom detail, unlock the corresponding detail in the SVN storage.

First, delete the database records. Use the following script to delete a detail:

```
DECLARE @Caption nvarchar(max);
SET @Caption = 'ToDelete';
DECLARE @UID UNIQUEIDENTIFIER;
select @UID = EntitySchemaUID from SysDetail
where Caption = @Caption
delete from SysDetail where EntitySchemaUID = @UID
```

Replace the “ToDelete” value with the detail schema name, which you can view in the [Advanced settings] of the System Designer. After deleting data from the database, delete the custom detail schema in the [Configuration] section. The [Advanced settings] in the System Designer allow deleting the object of the deleted detail.

Business processes

Contents

- **How to add auto-numbering to the edit page field**
- **Process launch from a client module**
- **Creating custom [User task] process element**
- **How to customize notifications for the [User task] process element**
- **How to run Creatio processes via web service**
- **How to save the record without closing the edit page which is opened by the business process**

How to add auto-numbering to the edit page field

Beginner Easy **Medium** Advanced

Introduction

You can add auto numbering for an object column. For instance, auto numbering is applied in the [Documents], [Invoices] and [Contracts] sections where a preformatted number is automatically generated when you add a new record.

There are two ways of implementing auto numbering:

- Client-side implementation.
- Server-side implementation.

To implement auto numbering **on the client side**, override the base virtual method *onEntityInitialized()* and call the *getIncrementCode()* method of the edit page base schema *BasePageV2*.

The *getIncrementCode()* method accepts two parameters:

- *callback* – the function that will run after receiving service response. The response must be passed to the corresponding column (attribute);
- *scope* – the context where the *callback* function will be run (optional parameter).

To implement auto numbering **on the server side**, add the event handler [Before record adding] to the object, whose column will be auto numbered. Set up number generating parameters in the business process, namely:

- Indicate the schema of the object for which generation will be performed.
- Call the [Generate ordinal number] action.
- Pass the generated value to the necessary object column.

This is not the only way to implement auto numbering on the server side. It can be implemented via custom means, for instance, by creating a [custom service](#).

Regardless of the chosen solution, add two system settings to use auto numbering:

- *[Entity]CodeMask* – object number mask.
- *[Entity]LastNumber* – current object number.

[Entity] – is the name of the object, whose column will be auto numbered. For example, *InvoiceCodeMask* (Invoice number mask) and *InvoiceLastNumber* (Current invoice number).

Case description

Set up auto numbering for the [Code] field in the [Products] section. The product code format must be as follows: ART_oooo1, ART_oooo2 and so on.

We covered two alternative ways of case implementation: client- and server-side.

Source code

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

The package does not contain the bound system settings *ProductCodeMask* and *ProductLastNumber*. You will need to add them manually.

Case implementation algorithm: client-side

1. Create two system settings

Create the [Product code mask] system setting with the following number mask: "ART_{o:ooooo}" (Fig.1) Populate the following fields:

- [Name] – "Product code mask".
- [Code] – "ProductCodeMask".
- [Type] – a string, whose length depends on the number of characters in the mask. In most cases 50 characters are enough. In this example, we use a string of unlimited length.
- [Default value] – "ART_{o:ooooo}".

Fig. 1. The [Product code mask] system setting

Product code mask

What can I do for you? >



SAVE

[CANCEL](#)

Name * Product code mask

Type * Unlimited length text

Default value ART_{0:00000}

Code * ProductCodeMask

Cached

Personal

Allow for portal users

Description

Create the [Product last number] system setting (Fig.2). Populate its properties:

- [Name] – “Product last number”.
- [Code] – “ProductLastNumber”.
- [Type] – “Integer”.

Fig. 2. The [Product last number] system setting

Product last number

What can I do for you? >

Creatio
7.15.1.1293

SAVE

[CANCEL](#)

Name * Product last number

Type * Integer

Default value

Description

Code * ProductLastNumber

Cached

Save value for current user

2. Create a replacing schema in the custom package

Create a replacing client module and specify the *ProductPageV2* schema as parent object (Fig. 3). The procedure for creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 3. Properties of the product edit page replacing schema

Properties

<Enter search text>

General

Title	Edit page - Product	x_a
Name	ProductPageV2	
Package	sdkCreateAutoIncrment	

Inheritance

Parent object	Edit page - Product (ProductBase)	
Replace parent	<input checked="" type="checkbox"/>	

3. Override the `onEntityInitialized()` method

In the collection of edit page view model methods, override the `onEntityInitialized()` method. In `onEntityInitialized()` method, call the `getIncrementCode()` method and fill in the generated number in the [Code] column of its callback function. The replacing schema source code is as follows:

```
define("ProductPageV2", [], function() {
    return {
        // The name of edit page object schema.
        entitySchemaName: "Product",
        // The collection of edit page view model methods.
        methods: {
            // Overriding Terrasoft.BasePageV2.onEntityInitialized base method, that
            // is triggered when the initialization of the edit page object schema is
            finished.
            onEntityInitialized: function() {
                // onEntityInitialized method parent realization is called.
                this.callParent(arguments);
                // The code is generated only in case we create a new element or a
                copy of the existing element.
                if (this.isAddMode() || this.isCopyMode()) {
                    // Call of the Terrasoft.BasePageV2.getIncrementCode base
                    method, that generates the number
                    // according to the previously set mask.
                    this.getIncrementCode(function(response) {
                        // The generated number is stored in [Code] column.
                        this.set("Code", response);
                    });
                }
            }
        }
    };
});
```

After saving the schema, clearing the browser cache and updating the application page, the automatically generated product code will be displayed when you add a new product (Fig.4).

Fig. 4. The result of case implementation on the client side

The screenshot shows a 'New record' form for a 'Product' object. The form has a header with 'New record', a search bar 'What can I do for you?', and a 'Creatio' logo. Below the header are buttons for 'SAVE', 'CANCEL', 'ACTIONS', and 'VIEW'. The main area contains fields for 'Name' (with a red asterisk), 'Code' (containing 'ART_00001'), 'Owner', 'Supervisor', and 'Inactive' (with an unchecked checkbox). To the left of the form is a placeholder icon for a user profile picture.

Case implementation algorithm: server-side

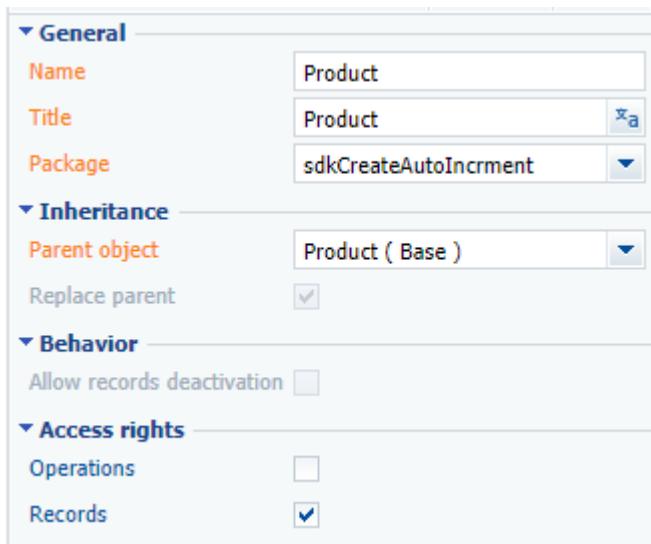
1. Create two system settings

This step is absolutely identical to the first step of case implementation algorithm on the client side.

2. Create a replacing schema of the [Product] object

Select a custom package and execute the [Add] – [Replacing object] menu command on the [Schemas] tab. Specify the [Product] object as the parent object in the new object properties (Fig. 5).

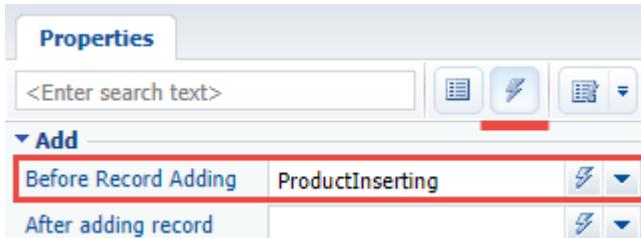
Fig. 5. Properties of the product replacing schema



3. Add the [Before record adding] event handler to the object schema

Add a new event handler in the object properties displayed in the **object designer**. To do this, go to the event tab and double-click the [Before Record Adding] field or click the event icon in this field (Fig.6).

Fig. 6. The [Before record adding] product event handler

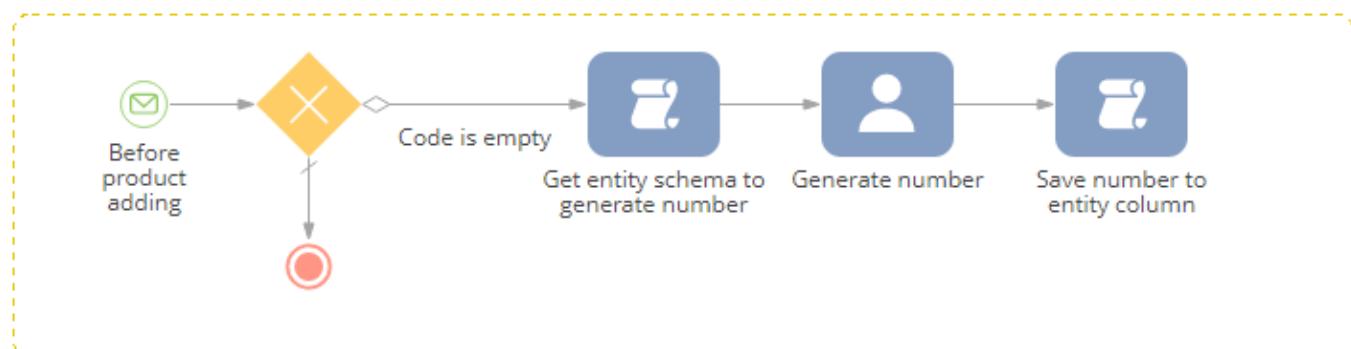


The object's **process designer** will open.

4. Add an event sub-process

To implement the [Before Record Adding] event handler, add [event sub-process](#) to the working area of the object process designer. Set up a business process for number generation (Fig.7).

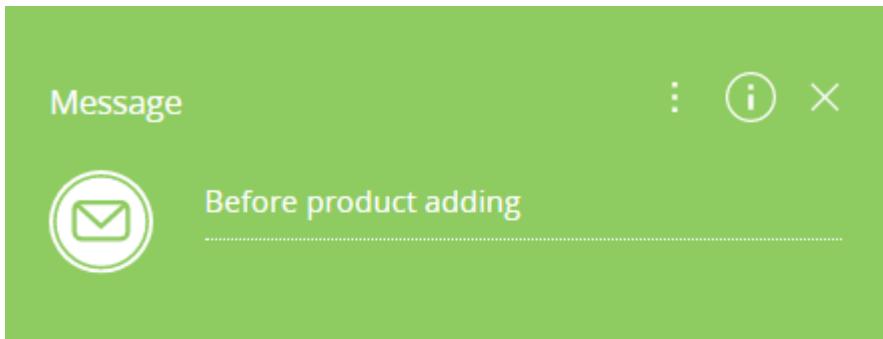
Fig. 7. The sub-process for the [Before Record Adding] event handler



Event sub-process elements

1. [Initial message](#) [Before product adding] (Fig.8) – the sub-process will be run upon receiving the *ProductInserting* message added at step3.

Fig. 8. The [Before product adding] initial message properties



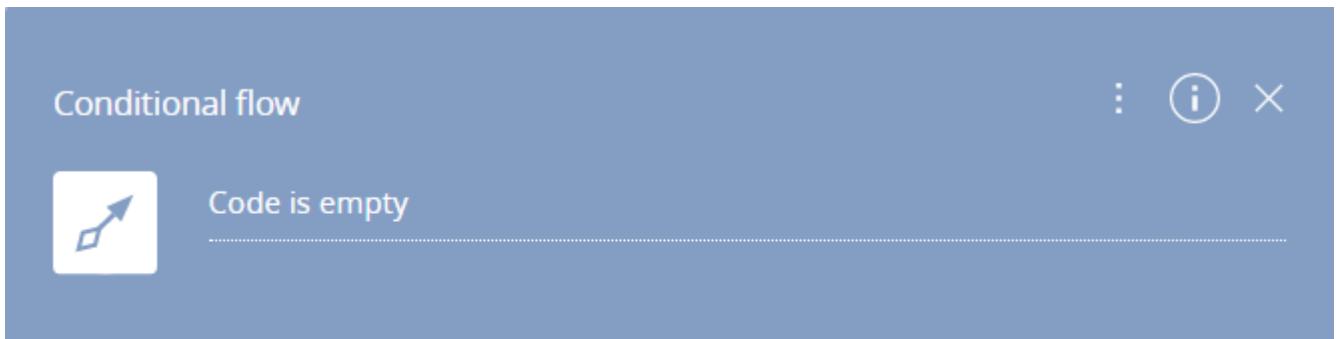
Which message event should start the process?

ProductInserting

2. [\[Exclusive gateway \(OR\)\]](#), which branches the process into two flows:

- [Default flow](#) – the transition down this flow will occur if the condition flow cannot be implemented. This branch finishes with the [\[Terminate\] event](#).
- [Condition flow](#) [Code is empty] – checks whether the [Code] column is populated (Fig.9). The further execution of the sub-process can only be possible if the column is not populated.

Fig. 9. The [Code is empty] condition flow properties



Condition to move down the flow

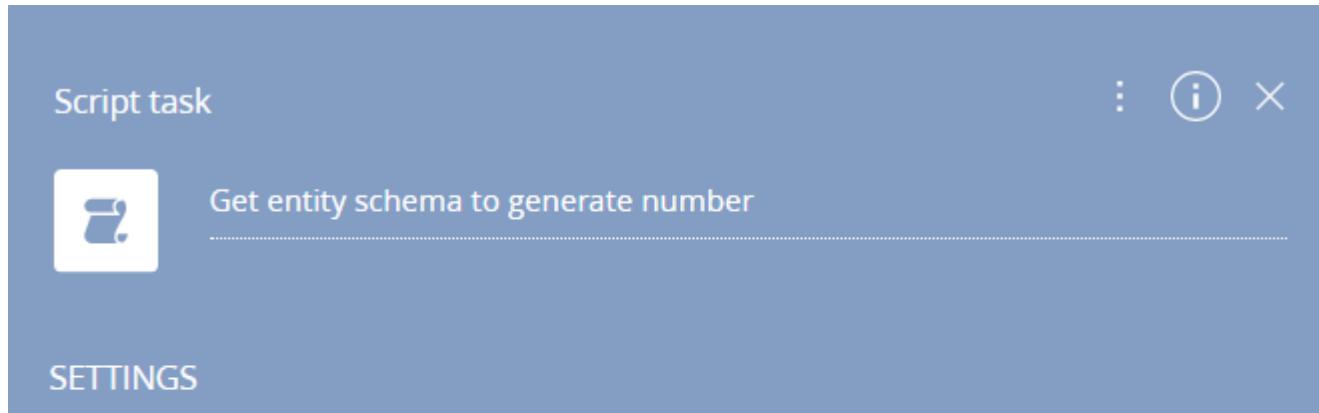
`string.IsNullOrEmpty(Entity.GetTypedColumnValue<string>("Code"))`

Add the following code to the [Condition] field of the condition flow:

```
string.IsNullOrEmpty(Entity.GetTypedColumnValue<string>("Code"))
```

3. [Script task](#) [Get entity schema to generate number] (Fig.10).

Fig. 10. [Get entity schema to generate number] script task properties



Code*

ScriptTask1

Program code C# script is executed in this element. To add it double-click the element. Add the following source code in the opened window:

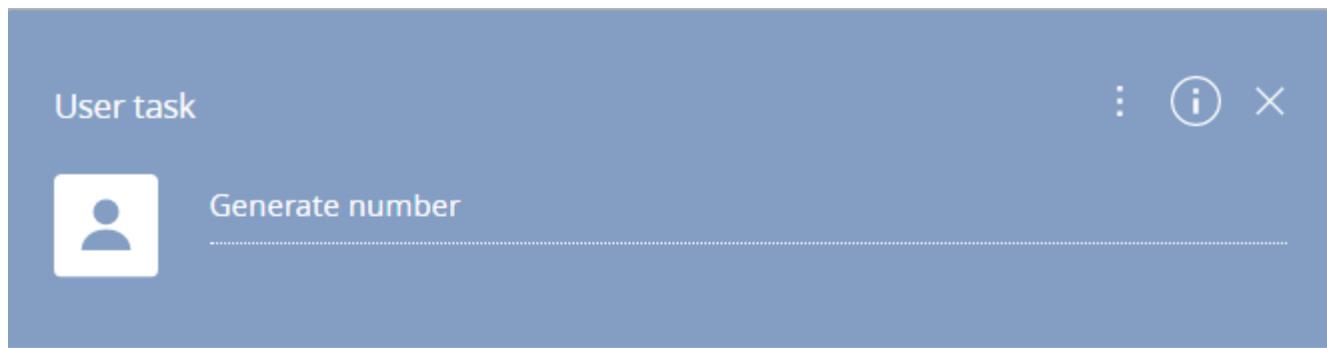
```
//Setting the schema for number generation.  
UserTask1.EntitySchema = Entity.Schema;  
return true;
```

Note that “UserTask1” here is the name of the [Generate number] user action.

Save the script after adding the source code. To do this, select the [Save] menu action.

4. [User task](#) [Generate number] (Fig.11).

Fig. 11. The [Generate number] user task properties



Which user task to perform?

Generate ordinal number



Process element parameters

EntitySchema*

ResultCode

This element performs the [Generate ordinal number] system action. It is the [Generate ordinal number] system action, which generates the current ordinal number in accordance with the *ProductCodeMask* mask set in the system settings.

5. [Script task](#) [Save number to entity column] (Fig.12).

Fig. 12. The [Save number to entity column] script task properties



Program code C# script is executed in this element. The value generated by the UserTask1 user action is stored in the [Code] column of the [Product] created object. The source code of the schema is available below:

```
Entity.SetColumnValue("Code", UserTask1.ResultCode);
return true;
```

Save and close the default process designer and publish the [Product] object schema. As a result, after saving the new product, the [Code] field will be automatically populated on the product page (Fig.13, Fig.14).

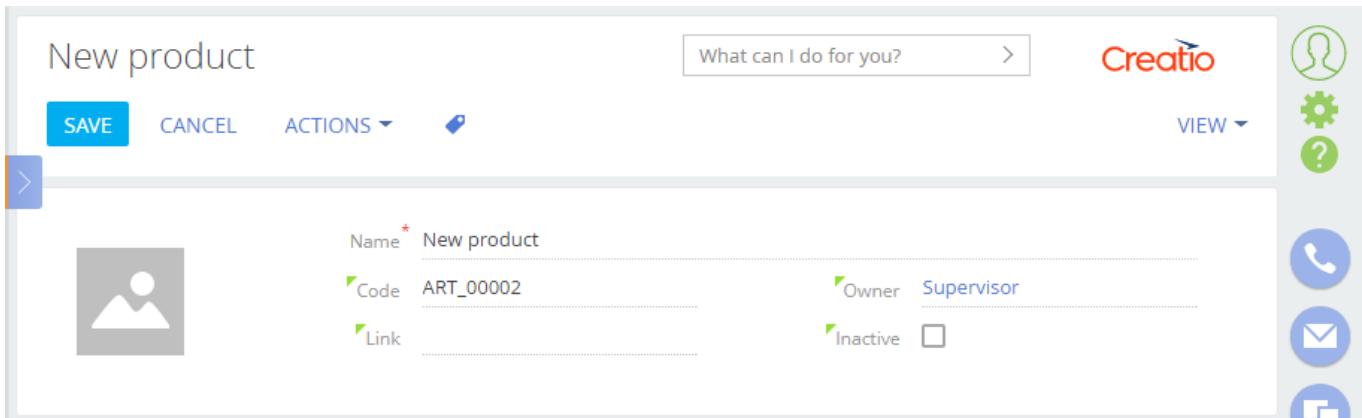
Since code auto generation and saving in the column is performed on the server side when the [Before saving record] event occurs, it is impossible to view the code value on the product page immediately. This is because the [Before saving record] event occurs on the server side after sending the request to add a record from the application client part.

Fig. 13. The code is not displayed when creating a product

The screenshot shows the 'New record' dialog for a product. The top bar includes 'New record', 'SAVE', 'CANCEL', 'ACTIONS ▾', and a search bar. On the right, there are 'VIEW ▾' and a vertical toolbar with icons for user profile, settings, help, phone, email, and file. The main form contains the following fields:

Name *	New product
Code	(Redacted)
Link	
Owner	Supervisor
Inactive	<input type="checkbox"/>

Fig. 14. The code is displayed in the saved product



Process launch from a client module

Beginner Easy **Medium** Advanced

Introduction

To launch a process from the JavaScript code client schema:

1. Add the *ProcessModuleUtilities* module as a dependency to the module of the page that was used for calling the service. This module provides a convenient interface for executing queries to the *ProcessEngineService.svc* service.
2. Call the *executeProcess(args)* method of the *ProcessModuleUtilities* module by passing the *args* object over to it as a parameter with the following properties (table 1):

Table 1. Properties of the args object

Property	Details
<i>sysProcessName</i>	The name of the called process (not required if the <i>sysProcessId</i> property is defined).
<i>sysProcessId</i>	The unique identifier of the called process (not required if the <i>sysProcessName</i> property is defined).
<i>parameters</i>	The object whose properties are the same as the properties of the called process incoming parameters.

Case description

Add an action that will launch the “Conducting a meeting” business process to the account edit page. Pass the primary contact of the account as a parameter to the business process.

Source code

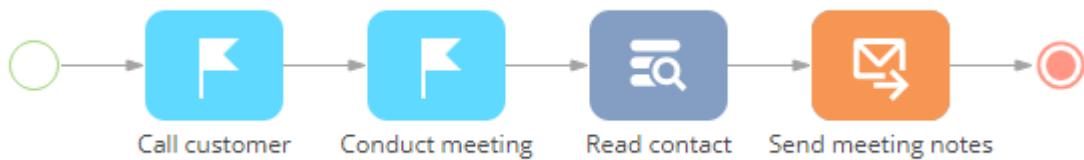
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Creating the “Conducting a meeting” custom business process

The case uses the “Conduct a meeting” business process described in the “[Designing a linear process](#)” and “[How to work with emails](#)” sections (fig. 1).

Fig. 1. Creating the “Conducting a meeting” source business process



After you create the business process, add the *ProcessSchemaContactParameter* incoming parameter to it. Specify “Unique identifier” in the [Data type] field of the parameter properties (fig. 2).

Fig. 2. Adding the incoming parameter

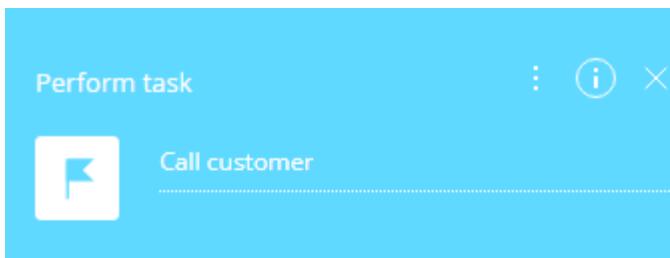
The screenshot shows the 'ADD PARAMETER' dialog box. At the top, there is a 'Process' header with a help icon and a close button. Below the header, the process name 'Holding a meeting' is displayed. The dialog has three tabs at the bottom: 'SETTINGS' (selected), 'PARAMETERS', and 'METHODS'. A large blue button labeled 'ADD PARAMETER ▾' is positioned above the input fields. The input fields are as follows:

- Title***: Meeting contact
- Code***: ProcessSchemaContactParameter
- Data type***: Unique identifier
- Value**: Select value

At the bottom right of the dialog are two buttons: 'SAVE' and 'CANCEL'.

In the properties of the [Call customer] action (which is the first business process action), populate the [Contact] field with the business process incoming parameter (fig. 3).

Fig. 3. Passing the parameter to the process element



Who performs the task?

[#System variable.Current user contact#]

Hint for user

Remind in

10 minutes

Connected to +

Contact

[#Meeting contact#]

2. Creating the replacing edit page of the account and adding the action.

Adding action to the edit page is covered in the “[Adding an action to the edit page](#)” article.

Add the *CallProcessCaption* localizable string with an action caption (for example, “Schedule a meeting”) to the replacing module schema of the edit page and the account section schema.

Add the *ProcessModuleUtilities* module as a dependency to declaring the edit page module.

The source codes of the section schema and the section edit page are below.

3. Adding the necessary methods to schemas

Use the *executeProcess()* method of the *ProcessModuleUtilities* module to launch the process. As a parameter, pass the object with the following properties: the created business process name, the object with initialized incoming process parameters.

In the below source code, it is implemented in the *callCustomProcess()* method. The *isAccountPrimaryContactSet()* method of verifying the availability of the primary contact and the *getActions()* method of adding action menu options are also implemented.

The source code of the edit page replacing module:

```
define("AccountPageV2", ["ProcessModuleUtilities"], function(ProcessModuleUtilities)
{
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Account",
        // Methods of the edit page view model.
        methods: {
            // Verifies if the [Primary contact] page field is populated.
            isAccountPrimaryContactSet: function() {
                return this.get("PrimaryContact") ? true : false;
            }
        }
    };
});
```

```

        },
        // Overriding the base virtual method that returns edit page action
collection.    getActions: function() {
    // Parent method implementation is called to receive
    // the collection of initialized actions of the base page.
    var actionMenuItems = this.callParent(arguments);
    // Adding a separator line.
    actionMenuItems.addItem(this.getActionsMenuItem({
        Type: "Terrasoft.MenuSeparator",
        Caption: ""
    }));
    // Adding the [Conducting a meeting] menu option to the edit page
action list.
    actionMenuItems.addItem(this.getActionsMenuItem({
        // Binding the caption of the menu option to localizable string
of the schema.
        "Caption": { bindTo: "Resources.Strings.CallProcessCaption" },
        // Binding the action handler method.
        "Tag": "callCustomProcess",
        // Binding the visibility property of the menu option to the
value, which returns the isAccountPrimaryContactSet() method.
        "Visible": { bindTo: "isAccountPrimaryContactSet" }
    }));
    return actionMenuItems;
},
// Action handler method.
callCustomProcess: function() {
    // Receiving the identifier of the account primary contact.
    var contactParameter = this.get("PrimaryContact");
    // The object that will be transferred to the executeProcess() method
as an argument.
    var args = {
        // The name of the process that needs to be launched.
        sysProcessName: "UsrCustomProcess",
        // The object with the ContactParameter incoming parameter value
for the CustomProcess process.
        parameters: {
            ProcessSchemaContactParameter: contactParameter.value
        }
    };
    // Launch of the custom business process.
    ProcessModuleUtilities.executeProcess(args);
}
}
);
}

```

Add the `isAccountPrimaryContactSet()` method implementation to the section schema for the correct action display in the menu when displaying the page with the vertical list in the combined mode.

The source code of the section schema replacing module:

```
define("AccountSectionV2", [], function() {
    return {
        // Name of the section schema.
        entitySchemaName: "Account",
        methods: {
            // Verifies if the [Primary contact] field of the selected record is
populated.
            isAccountPrimaryContactSet: function() {
                // Defining the active record.
```

```
        var activeRowId = this.get("ActiveRow");
        if (!activeRowId) {
            return false;
        }
        // Receiving the data collection of the section record list view.
        // Receiving the model of the selected account by the set value of
the primary column.
        var selectedAccount = this.get("GridData").get(activeRowId);
        if (selectedAccount) {
            // Receiving the model property – availability of the primary
contact.
            var selectedPrimaryContact =
selectedAccount.get("PrimaryContact");
            // The method returns true if the primary contact is established.
Otherwise, it returns false.
            return selectedPrimaryContact ? true : false;
        }
        return false;
    }
};

});
```

After you save the schemas and update the application page with clearing the cache, the new [Schedule a meeting] action will appear in the account page action menu (fig. 4). This action is available only if there exists a primary contact for the active list record. When executing the action, the “Conducting a meeting” custom business process will be launched. The primary contact of the account will be passed to the business process parameter (fig. 5)

Fig. 4. Launch of the business process by action on the edit page

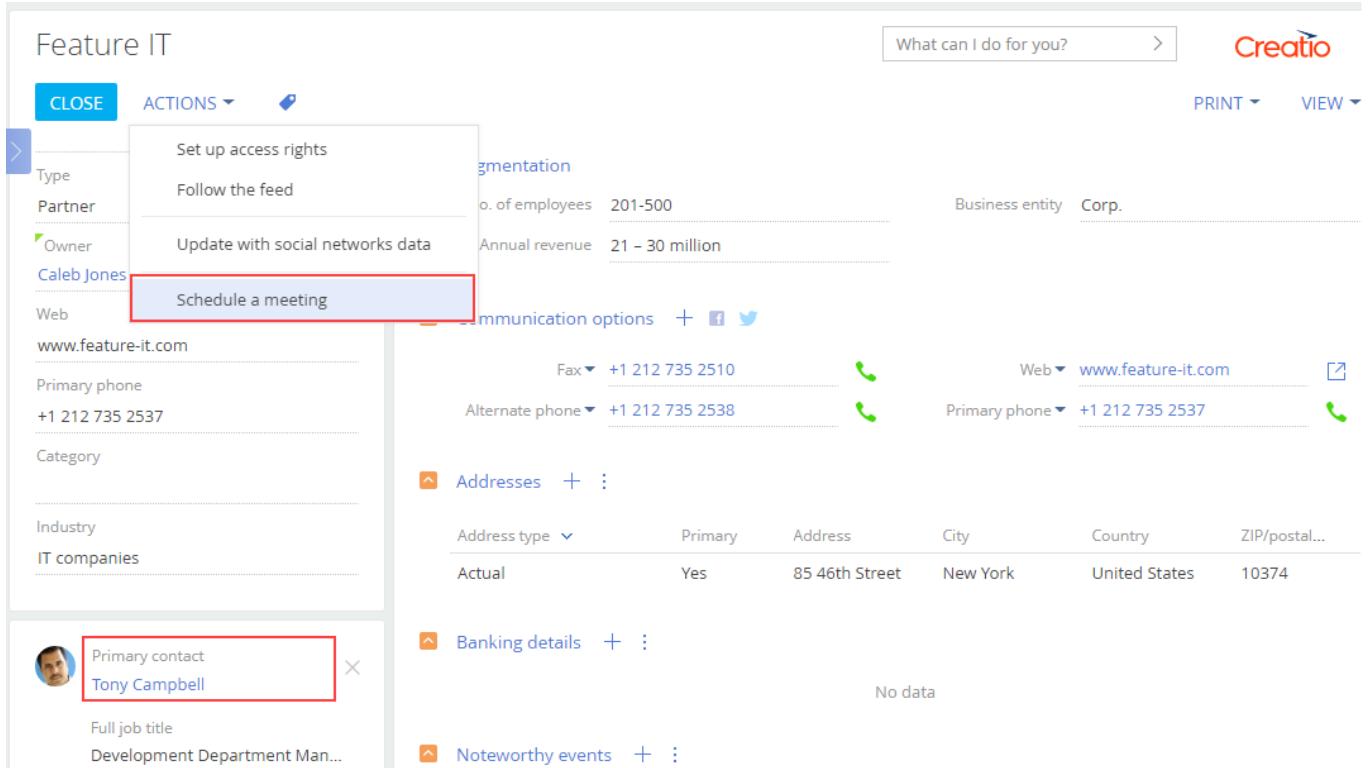
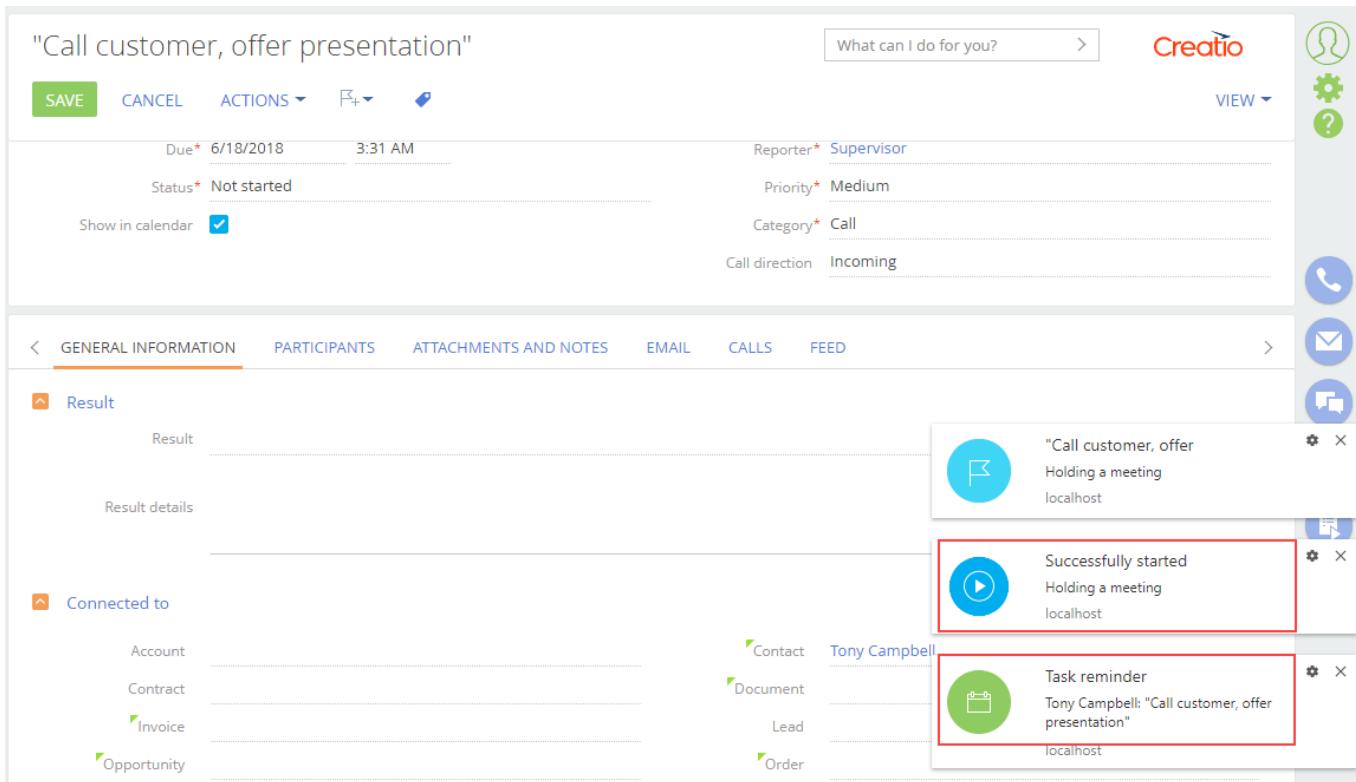


Fig. 5. Result of the business process launch. Passing the parameter from the account edit page to the business process



Creating custom [User task] process element

Introduction

It is often necessary to perform similar operations repeatedly while working with business processes in Creatio. The [User task] process element is best suited for these operations. Learn more about the [User task] element in the "[\[User task\] process element](#)" article.

By default, a number of user tasks is already available in the system. You can add new user tasks if needed.

The “User task” configuration schema type is used to create new user tasks. The process task partially replicates the logic of the [Script task] process element. However, a user task can be reused in different processes. Any changes to the task will be immediately applied to all processes that contain the mentioned task.

Case description

Create a simple process user task that would calculate the sum of two numbers. Use two numbers (specified as task parameters) to calculate the sum.

Source code

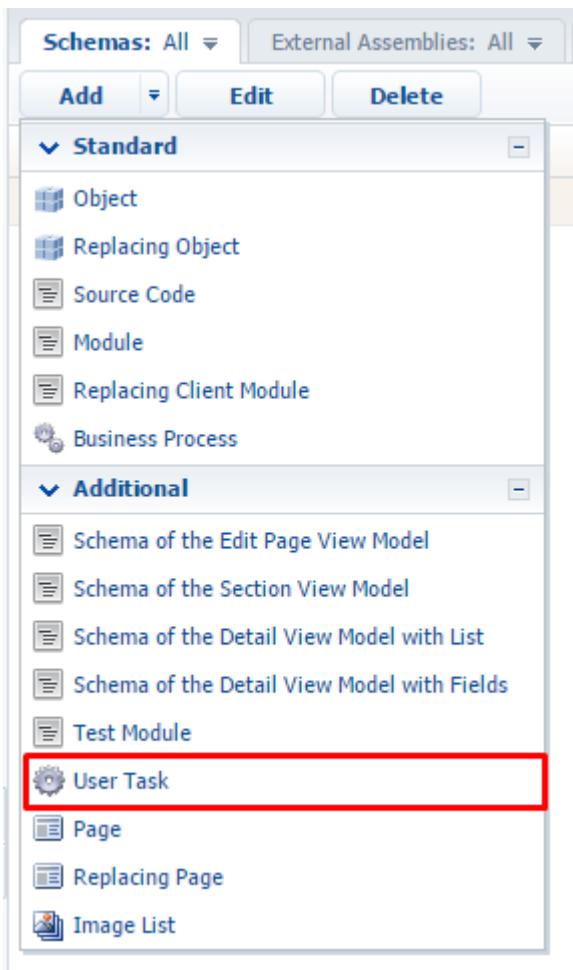
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Creating a user task schema

Go to the [Configuration] section of the system designer, select a custom package and execute the menu command [Add] - [User Task] on the [Schemas] tab (Fig. 1).

Fig. 1. Creating a user task schema



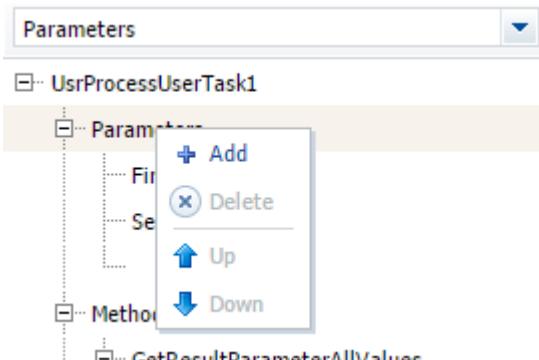
A user task designer window will open.

The default values for [Name] and [Title] are "UsrProcessUserTask1" and "User Task 1", respectively.

2. Adding task parameters

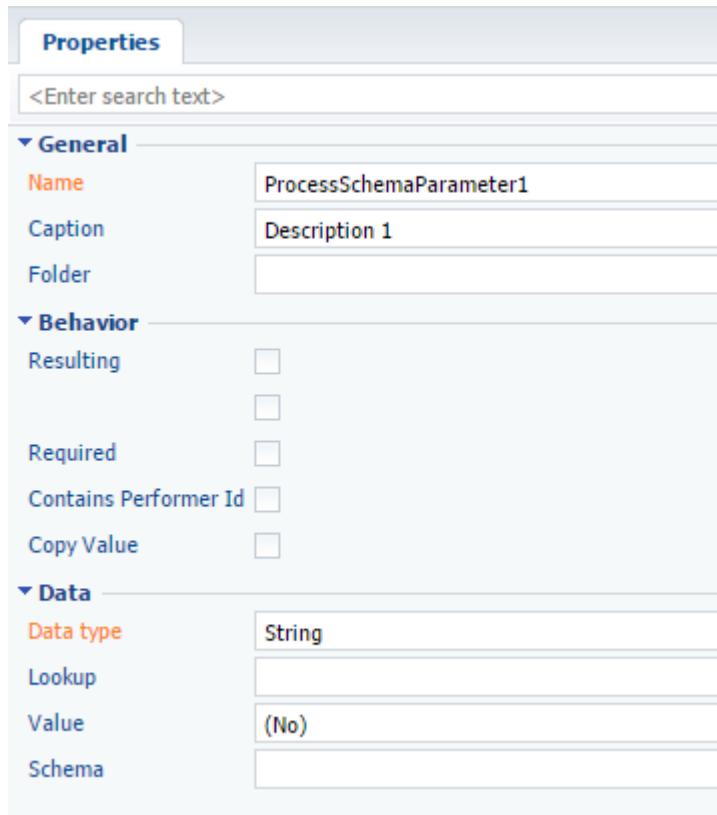
On the [Activity] tab of the process task designer, open the context menu on the [Parameters] element to add task result parameters. Execute the [Add] context menu command (Fig. 2).

Fig. 2. Creating a process task parameter



A new parameter will be added as a result, its main properties are displayed in Fig. 3.

Fig. 3. Default properties of a user task parameter



Add three parameters to create the process task (main properties are shown in table 1).

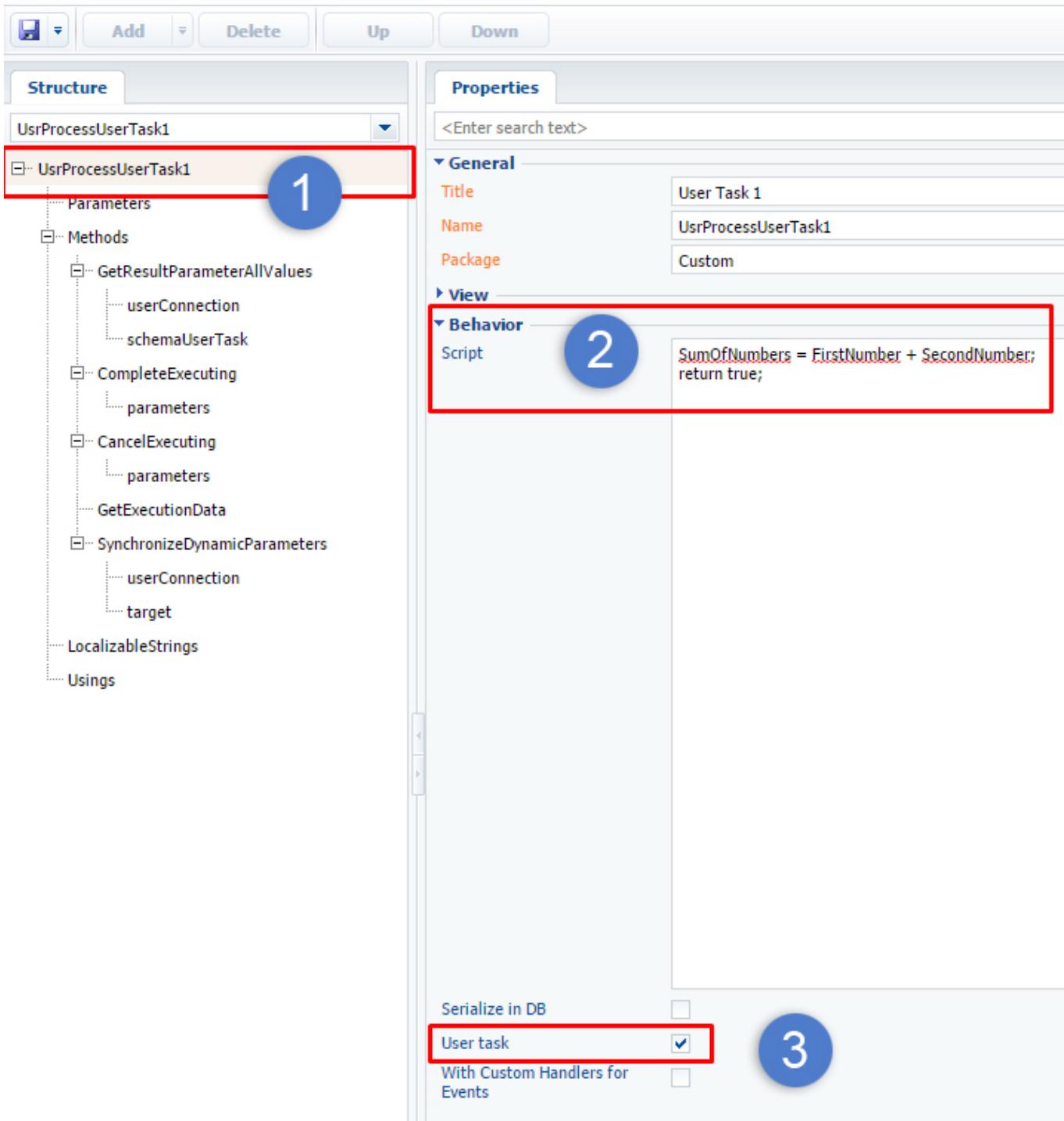
Table 1. Main parameters of the created process task

Name	Type	Description
FirstNumber	Integer	First number
SecondNumber	Integer	Second number
SumOfNumbers	Integer	Sum of numbers

3. Adding task logic

The task logic is set via a script. The task script is a process task parameter which contains the C# program code used to implement the necessary task logic.

Fig. 4. Adding user task logic



To add the task script program code, select the root element of the structure (Fig. 4, 1) and add the program code to the text field of the [Script] property (Fig. 4, 2):

```
// Performing operations with task parameters
SumOfNumbers = FirstNumber + SecondNumber;
// Indicates that the task script execution was successful.
return true;
```

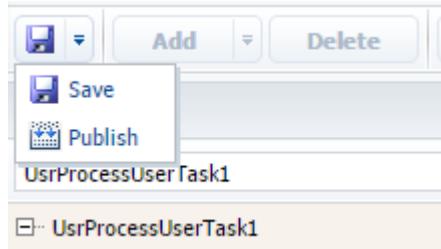
The parameters must be addressed directly by name. Return the *true* value at the end of the script to signal the successful execution of the element and continue the process.

Select the [User task] checkbox (located under the script input field (Fig. 4, 3)) to enable using the custom user task in business processes.

4. Saving and publishing the schema

After assigning values to the necessary properties of the created process task schema, save the schema and publish it (Fig. 5).

Fig. 5. Saving and publishing the schema



You can use the user task to create business processes after successfully publishing the schema.

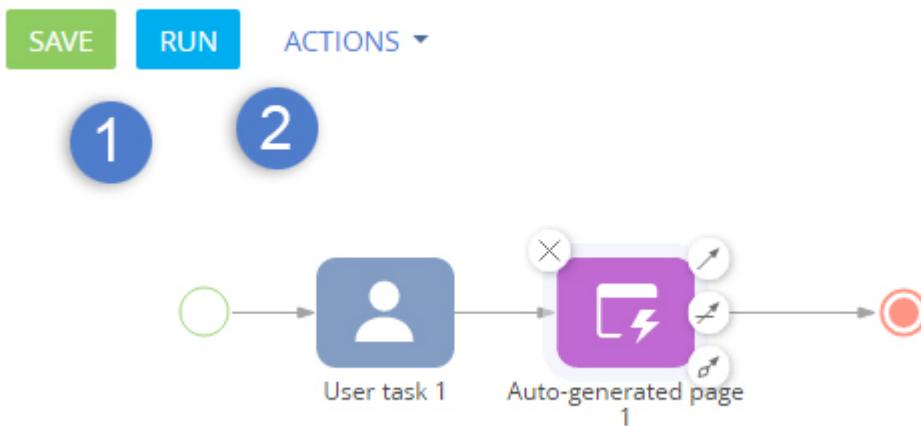
5. Testing

Create a new business process to test the user task. Learn more about creating business processes in the "[How to create business processes](#)" article.

In the process designer, add the [User task] and the [Auto-generated page] elements to the process diagram (Fig. 6).

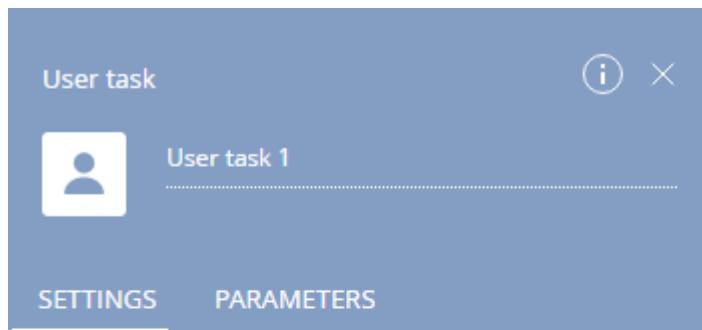
Fig. 6. Business process diagram.

Business process 1



Specify the schema title (see step 1) in the [Which user task to perform?] property of the [User task 1] element (Fig. 7).

Fig. 7. Selecting a custom process task



Which user task to perform?

User Task

User Task 1

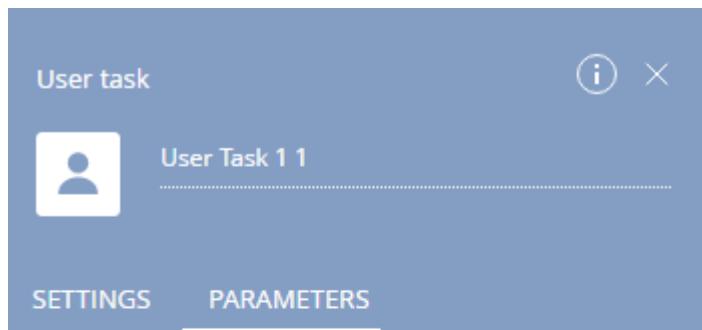
UserTask1

Enable logging

Serialize in DB

Set the values of the parameters whose sum will be calculated (Fig. 8). The [Sum of numbers] parameter value will be determined by the script (see step 3), so the value entry field for this parameter can be left unpopulated.

Fig. 8. User task parameter values



123 First Number

2

123 Second Number

16

123 Sum Of Numbers

Select value

The [Auto-generated page 1] element displays the task result, i.e. the sum of parameter values of the user task. To

display the sum of parameters, add an integer to the page element collection (Fig. 9) and set the title of the auto-generated page element (Fig. 10, 1). To set the displayed value, call the value formula dialog window by clicking the lightning icon in the [Value] field (Fig.10, 2).

Fig. 9. Adding page elements



Page title

Auto-generated page 1

Buttons +

Page Items +

To whom sh

[#System var

Recommendat

User hints

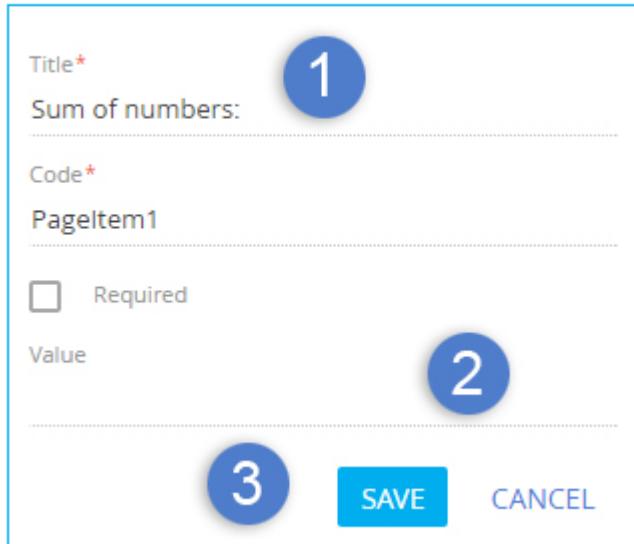
Which record

Connected ob

- Notes
- Item block
- Text field
- Selection field
- Boolean
- Date/time
- Integer
- Decimal

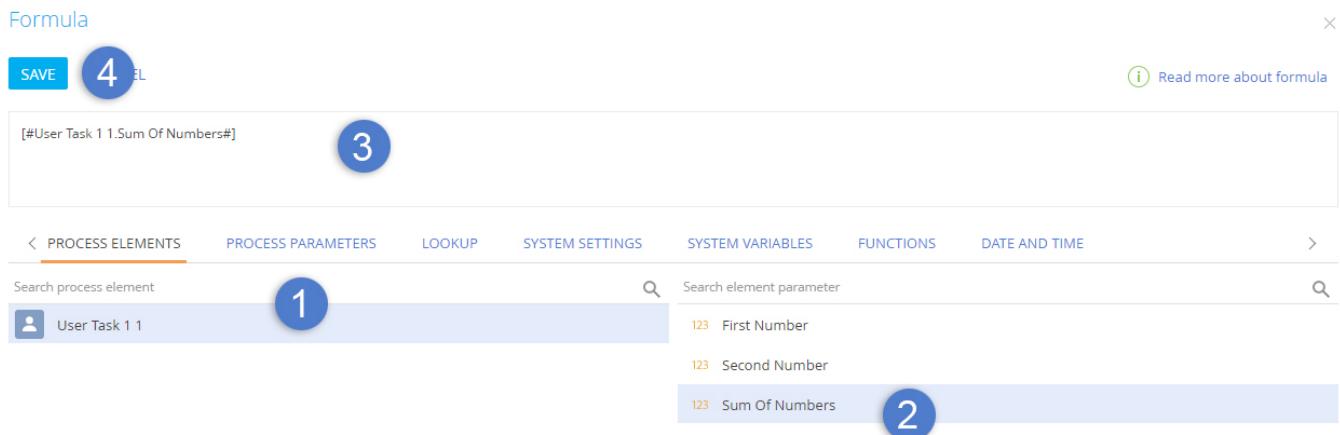
Fig. 10. Page element parameters

Page Items +



In the [Formula] dialog window that appears on the [Process Elements] tab, select the [User task 1] element (Fig. 11, 1) and double-click the [Sum of Numbers] element parameter (Fig. 11, 2). The formula used to calculate the auto-generated page value will be displayed (Fig. 11, 3).

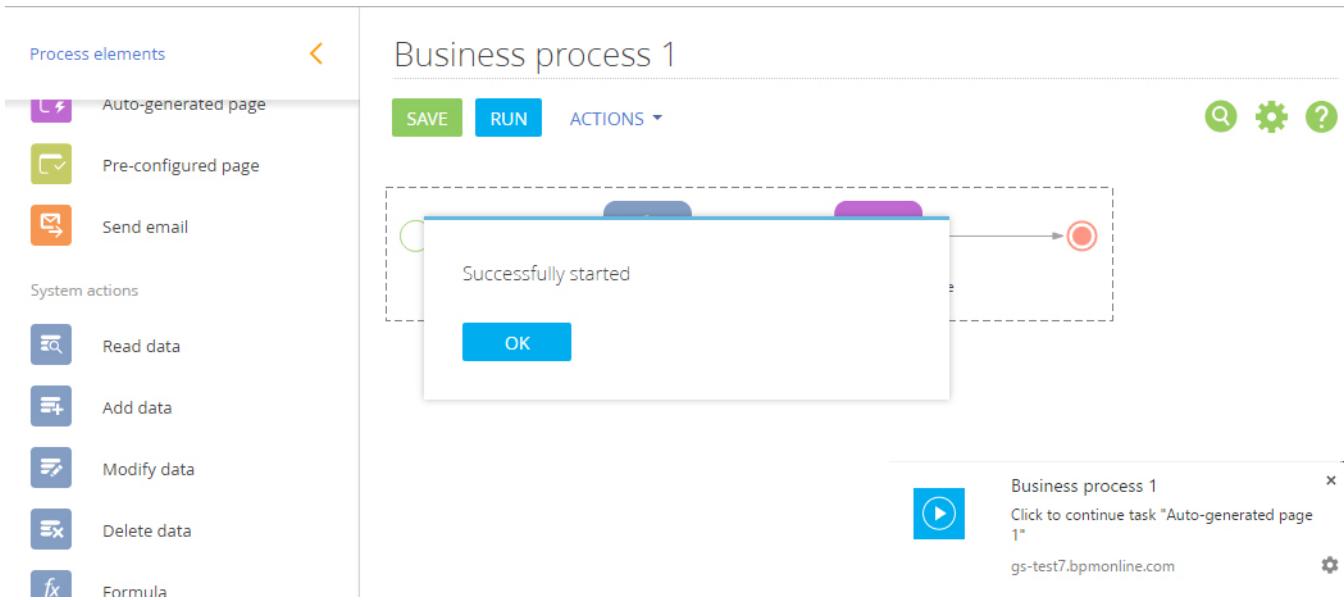
Fig. 11. The [Formula] dialog window



Save the formula (Fig. 11, 4) and the added auto-generated page element (Fig. 10, 3).

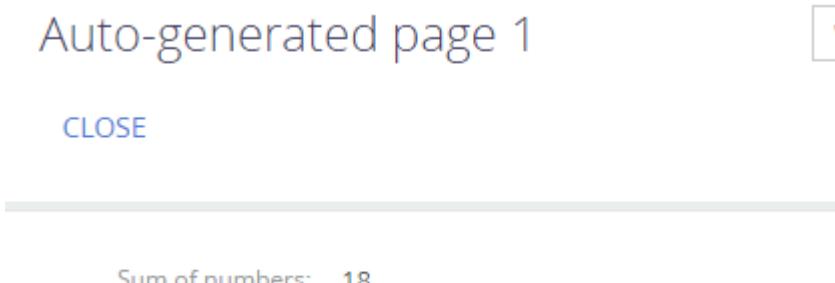
Run the process (Fig. 6, 2) after saving it (Fig. 6, 1). The message will alert that the process has started. A notification will enable you to display the business process log (Fig. 12).

Fig. 12. Business process start message



The business process result is shown in Fig. 13.

Fig. 13. Result of the business process execution.



After changing the parameter values of the custom user task, change the currently displayed page before starting the business process again (go to any Creatio section, for example). If you do not leave the auto-generated business process page, it will display the previous result.

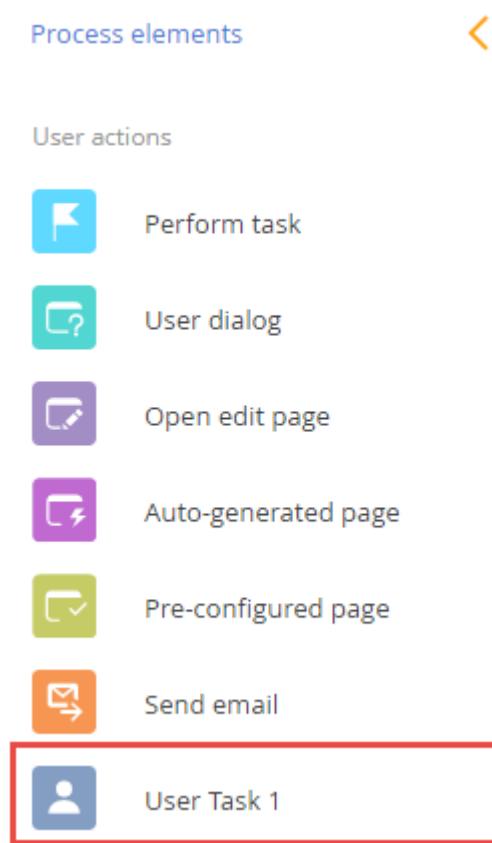
Adding the user process task to the [Process elements] tab

If the created user process task element is planned for regular use, it can be added to the [Process elements] tab in the process designer. To do so, execute the following SQL script in the database:

```
-- UsrProcessUserTask1 - name of the process task schema.  
insert into SysProcessUserTask(SysUserTaskSchemaUID, Caption)  
select s.UId, s.Caption from SysSchema s  
where s.Name = 'UsrProcessUserTask1'
```

After restarting (or compilation) the application, the element will be displayed on the tab (Fig. 14).

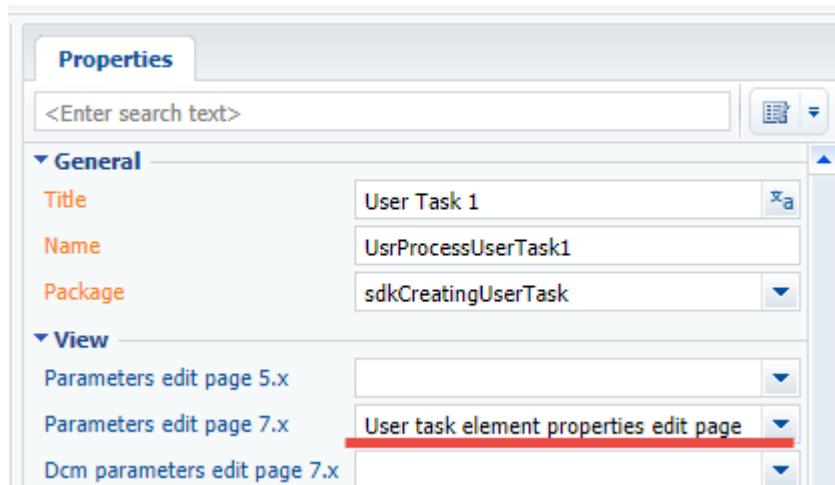
Fig. 14. User task element on the tab



Possible issues

If the parameters tab of the created element is not displayed (see Fig. 8), specify the *UserTaskPropertiesPage* ("User task element properties edit page") schema in the "Parameters edit page 7.x" property (see Fig. 15).

Fig. 15. — "Parameters edit page 7.x" property



How to customize notifications for the [User task] process element

Beginner Easy Medium **Advanced**

Introduction

The [User task] process element can create notification on the [Business Process Tasks] panel, just like any other process action. To activate the notification mechanism, select the [Serialize in DB] checkbox in the process properties and define the *ShowExecutionPage* logical parameter.

Default process actions are able to create notifications for process steps. Custom process actions must be manually assigned this parameter.

Case of creating a user task with a notification

Case description

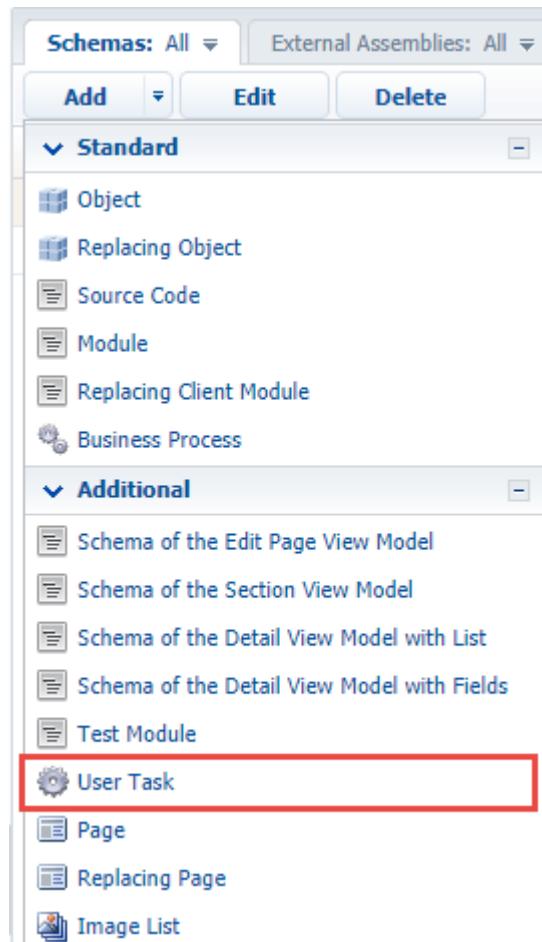
Create a simple custom process action ([User task]) that would automatically add a notification with “Attention” in its title and “Very important!” as its text. The notification must be displayed on the [Business process task] panel

Case implementation algorithm

1. Create a user task schema

To do this, go to the [Configuration] section of the system designer, select a custom package and on the [Schemas] tab, execute the menu command [Add] - [User Task] (Fig.1).

Fig. 1. Creating a user task schema



Set the property values specified in table 1 for the created schema.

Table 1. Custom process task properties

Property	Value
Title	Customized User Task
Name	UsrCustomizedUserTask

Small vector image

Scalable Vector Graphics The notification will be displayed on the [Business process task] panel. For this particular case we will use the SVG image available under the [link](#).

Notification Icon (54x54 px)

Image for the notification pop-up window in PNG (Portable network graphics) format, 54x54 px. In the current case we will use the following image:



Serialize in DB

Select the checkbox.

User task

Select the checkbox.

After making the changes, save the schema's meta data.

2. Add the ShowExecutionPage parameter

To enable automatic adding of the notification when the user task is run, add a new parameter (ShowExecutionPage) to the Parameters section of the custom user task (Fig. 2). Its primary properties are listed in table 2.

Fig. 2. Adding a schema parameter

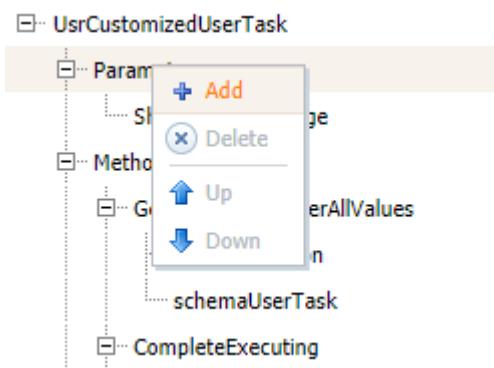


Table 2. Properties of the ShowExecutionPage parameter

Property	Value
Title	ShowExecutionPage
Name	ShowExecutionPage
Data type	Boolean

The value of the *ShowExecutionPage* parameter does not affect the notification mechanism. If the specified parameter exists in the process action, then before any process step implemented by this User task is executed, an automatic notification will be created for this step.

After making the changes, save the schema's meta data.

3. Override the GetNotificationData() method

The contents of the business process step notification is generated via the *GetNotificationData()* method that can be overridden.

The method must return an instance of the *Terrasoft.Core.Process.ProcessElementNotification* class that contains data for the business process step notification. We recommend calling the base method first, which will return instance of the *ProcessElementNotification*, populated with default values, and then customize this instance. Full description of the *ProcessElementNotification* class properties is available in the **.NET class libraries of platform core (on-line documentation)**.

The properties that are most useful for customization are available in table 3.

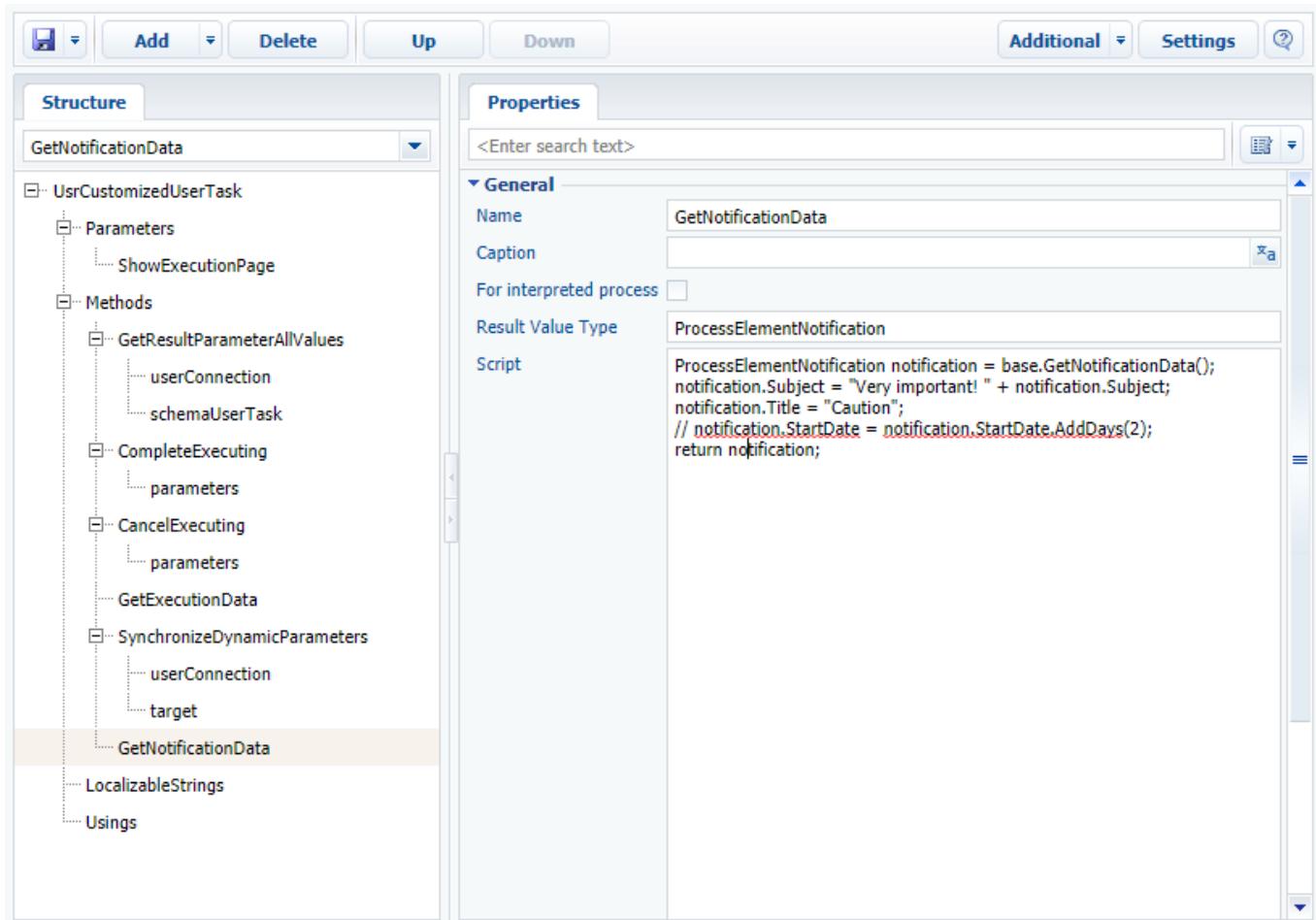
Table 3. Primary properties of the *Terrasoft.Core.Process.ProcessElementNotification* class

Property	Value
Title	Heading of the business process step notification. Default value – business process element name on the diagram.
Subject	Text of the business process step notification. Provide any process element specifics that are relevant to the notification recipient user here. Default value is the <i>NotificationCaption</i> process parameter value of the corresponding process. Thus, all steps of a business process will have the save value of the <i>Subject</i> parameter.
StartDate	Date and time of notification for system user. Default value – moment when the notification about the business process step has been created. This notification will be displayed for the user immediately after the process step is activated.

To execute the case conditions, override the *GetNotificationData()* method of the created schema. To do this, select the GetNotificationData node in the schema structure. Add the following code in the [Script] field (Fig. 3):

```
// Executing base method and getting an instance of the ProcessElementNotification
// class generated by default.
ProcessElementNotification notification = base.GetNotificationData();
// Customizing the notification element.
notification.Subject = "Very important! " + notification.Subject;
notification.Title = "Attention";
// You can postpone date and time of the notification.
// notification.StartDate = notification.StartDate.AddDays(2);
// The method returns customized instance of ProcessElementNotification
return notification;
```

Fig. 3. Overriding the *GetNotificationData()* method

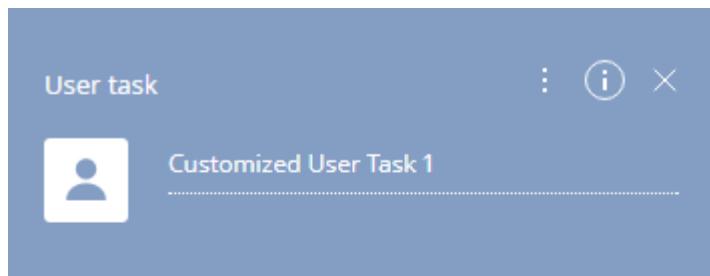


After making the changes, publish the schema.

4. Use the created element in the business process.

After publishing the user task schema, this action becomes available for use in Creatio business processes. To use this custom action, add the [User Task] element on the process diagram and in the [Which user task to perform?] field, select "Customized User Task". After this, *ShowExecutionPage* Boolean parameter will be added to the user task parameters (Fig. 4). This parameter is optional.

Fig. 4. The *ShowExecutionPage* user task parameter



Which user task to perform?

Customized User Task

Process element parameters

ShowExecutionPage

Select value

To correctly use the user task, select the [Serialize in DB] checkbox for it. Enable advanced mode in the process element properties (Fig. 5), and select the needed check box (Fig. 6, 1).

Fig. 5. Opening advanced properties of a process element



Fig. 6. The [Serialize in DB] check box

Business Process With Cusomized User Task

SAVE CANCEL ACTIONS ▾

2

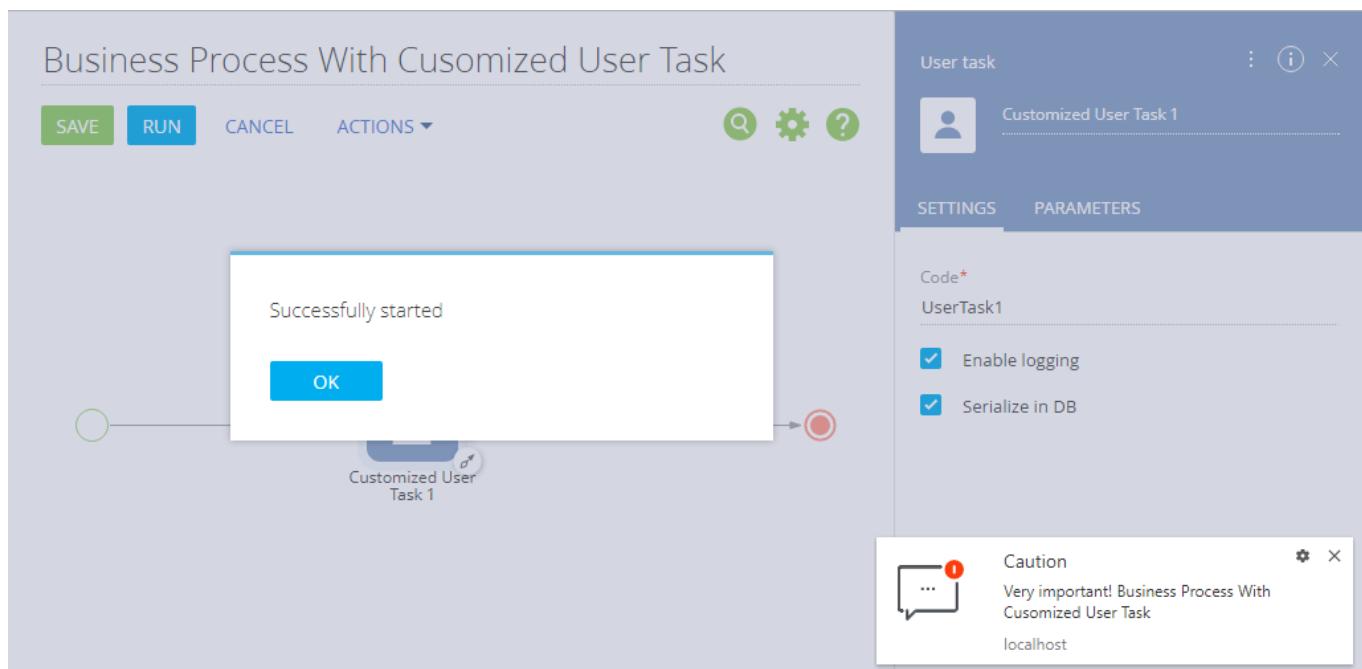
3

The screenshot shows a business process diagram with a start event, a 'Customized User Task 1' node, and an end event. The task node has three outgoing transitions. To the right, a settings dialog for 'Customized User Task 1' is open, showing the 'SETTINGS' tab. It includes fields for 'Code*' (UserTask1) and checkboxes for 'Enable logging' and 'Serialize in DB'. The 'Serialize in DB' checkbox is highlighted with a red circle containing the number 1.

Save the process (Fig. 6, 2) and run it (Fig. 6, 3).

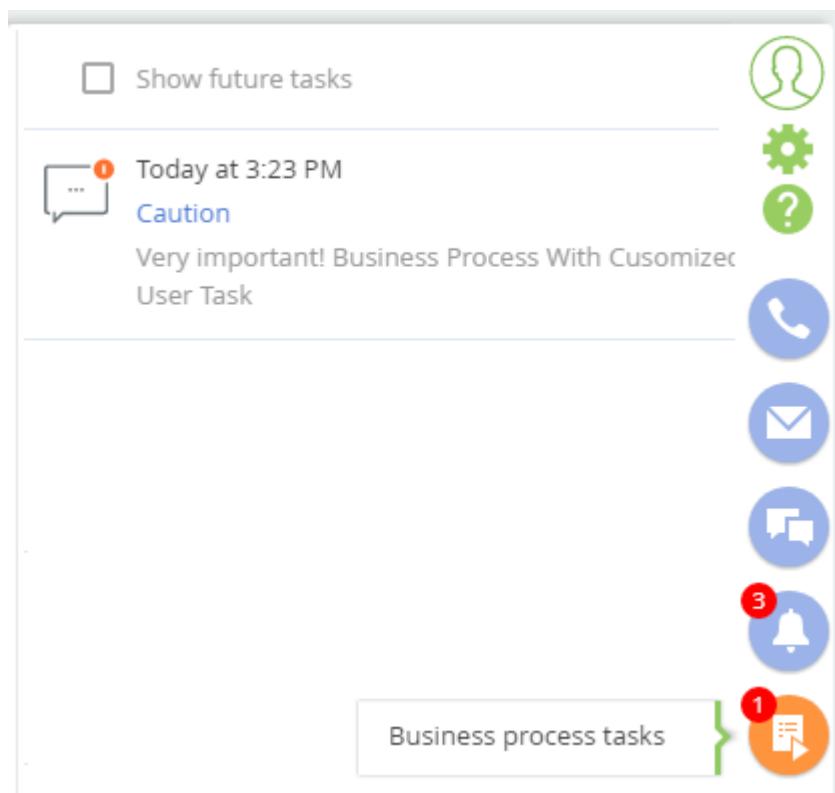
As a result, the corresponding notification (configured in the `GetNotificationData()` method) will be displayed (Fig. 7).

Fig. 7. Case result: Custom notification



The [Process tasks] panel will display a notification with the same properties (Fig. 8).

Fig. 8. Case result: Custom notification



How to run Creatio processes via web service

Beginner **Easy** **Medium** **Advanced**

Introduction

The `ProcessEngineService.svc` web service is used to run business processes from the third-party applications. Main

features of the *ProcessEngineService.svc* web service are described in “**The ProcessEngineService.svc web service**” article.

Case description

Run a demo business processes of creating and reading Creatio contacts from the browser address bar or third-party application via the *ProcessEngineService.svc* web service.

Case implementation algorithm

To implement the case:

1. Create the demo processes of adding new contact and reading all contacts.
2. Check the operability of the *ProcessEngineService.svc* web service from the browser address bar.
3. In the third-party application, create a class and implement logic of interaction with the *ProcessEngineService.svc* web service in this class.

1. Creating the demo business processes

Best practices of business process creation in Creatio are provided in the [business process guide](#).

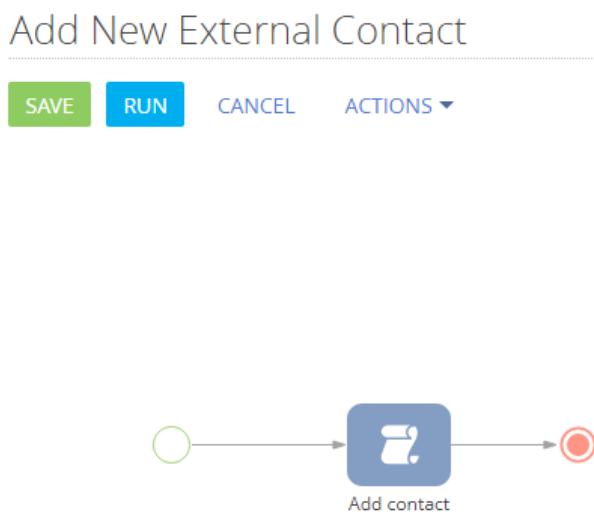
1.1. Creating the process of adding a new contact

The business process of adding a new contact has start event, end event and [ScriptTask] element in which the logic of adding a new contact is implemented. The values of business process properties (Fig. 1):

- [Name] – "Add New External Contact"
- [Code] – "UsrAddNewExternalContact".

Default values may be used for the other properties.

Fig. 1. Properties of the UsrAddNewExternalContact business process



Process

Add New External Contact

SETTINGS PARAMETERS METHODS

Code*
UsrAddNewExternalContact

Version
0

Tag
Business Process

Process description

Package*
sdkWorkWithBpmByWebServices

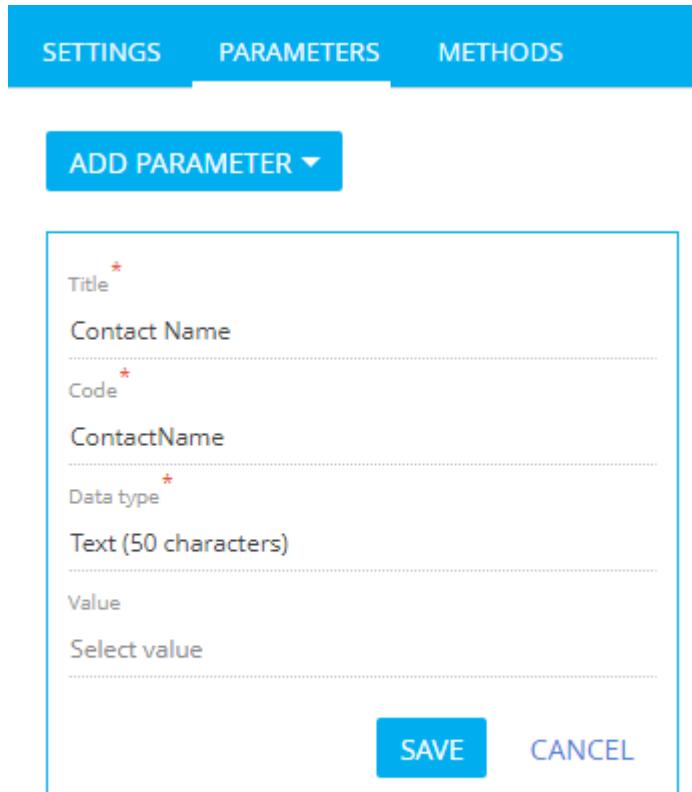
Maximum Number of Repetitions
100

Process instance caption

The business process contains two text parameters (Fig. 2). The contact details are sent to the process via these parameters:

- *ContactName* – contains a name of the new contact
- *ContactPhone* – contains a phone number of the new contact.

Fig. 2. The parameters of the business process.



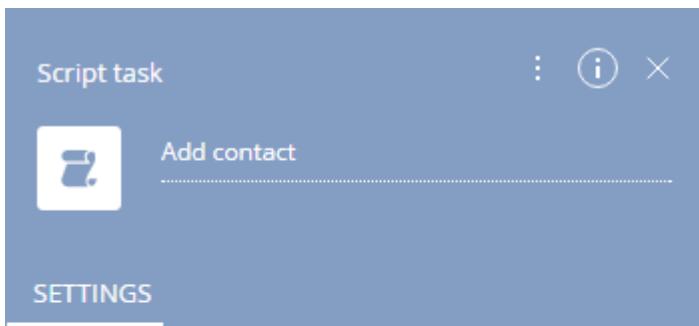
T Contact Phone

Select value

Logic of adding a new contact is implemented in the [ScriptTask] element. The values of element properties (Fig. 3):

- [Name] – "Add contact"
- [Code] – "ScriptTaskAddContact"
- [For interpreted process] – checkbox unchecked.

Fig. 3. [ScriptTask] element properties



Code*

ScriptTaskAddContact

For interpreted process

The source code for the ScriptTaskAddContact element:

```
// Create an instance of the schema of the "Contact" object.  
var schema = UserConnection.EntitySchemaManager.GetInstanceByName("Contact");  
// Create an instance of a new object.  
var entity = schema.CreateEntity(UserConnection);  
// Set the default values for the object columns.  
entity.SetDefColumnValues();  
// Set the value of the "Name" column from the process parameter.  
entity.SetValue("Name", ContactName);  
// Set the value of the "Phone" column from the process parameter.  
entity.SetValue("Phone", ContactPhone);  
// Saving a new contact.  
entity.Save();  
return true;
```

Save the business process to apply changes.

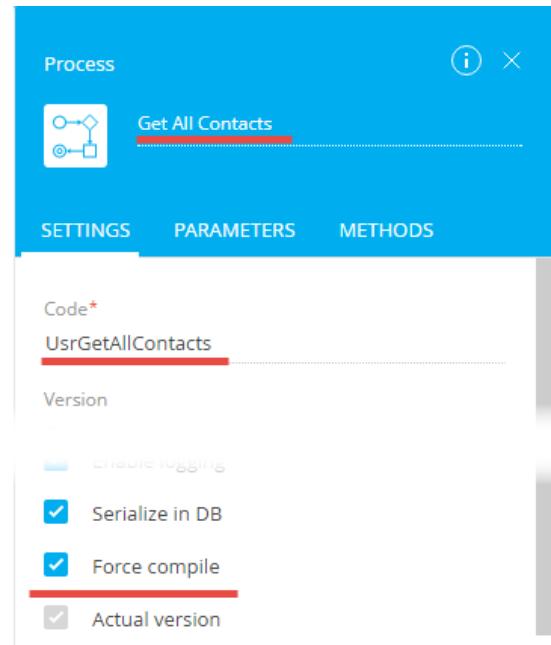
Creating the process of reading contacts

Business process that generates a list of all contacts is also contains one [ScriptTask] element in which the necessary logic is implemented. The values of business process properties (Fig. 4):

- [Name] – "Get All Contacts"
- [Code] – "Usr GetAllContacts"
- [Force compile] – checkbox checked.

Default values may be used for the other properties.

Fig. 4. Properties of the contacts reading business process



The *Usr GetAllContacts* process contains the *ContactList* parameter. The process will get a list of all contacts as a JSON object through this parameter. Parameter type – unlimited length string. Parameter properties are listed on Fig. 5.

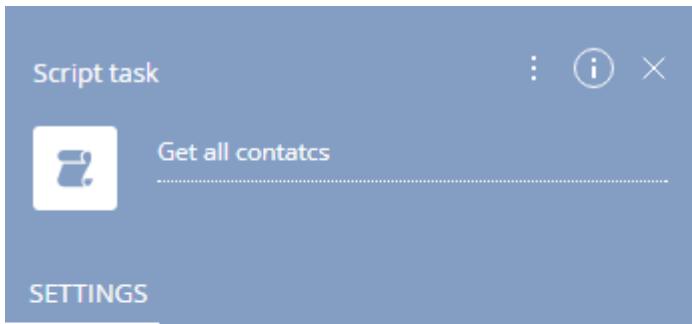
Fig. 5. Parameter properties

SETTINGS	PARAMETERS	METHODS
ADD PARAMETER ▾ Title * Contact List Code * ContactList Data type * Unlimited length text Value Select value SAVE CANCEL		

Logic of getting contacts is implemented in the [ScriptTask] process element. The values of element properties (Fig. 6):

- [Name] – "Get all contates"
- [Code] – "ScriptTaskGetAllContacts"
- [For interpreted process] – checkbox unchecked.

Fig. 6. [ScriptTask] element properties



Code*

ScriptTask GetAllContacts

For interpreted process

The source code for the *ScriptTaskGetAllContacts* element:

```
// Create an EntitySchemaQuery instance.
EntitySchemaQuery query = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"Contact");
// A flag for required selection of the primary column (Id).
query.PrimaryQueryColumn.IsAlwaysSelect = true;
// Adding columns to the request.
query.AddColumn("Name");
query.AddColumn("Phone");
// Getting the result collection.
var list = query.GetEntityCollection(UserConnection);
// Creating a list of contacts for serialization in JSON.
List<object> contacts = new List<object>();
foreach (var item in list)
{
    var contact = new
    {
        Id = item.GetTypedColumnValue<Guid>("Id"),
        Name = item.GetTypedColumnValue<string>("Name"),
        Phone = item.GetTypedColumnValue<string>("Phone")
    };
    contacts.Add(contact);
}
// Save the serialized JSON collection of contacts to the ContactList parameter.
ContactList = JsonConvert.SerializeObject(contacts);
return true;
```

Save the business process to apply changes.

2. Run business processes from the browser address bar

The call to the service method is possible using an HTTP GET request and you can use a standard browser to start a business process. General URL formats of calling a service for business processes with parameters are described in the “**The ProcessEngineService.svc web service**” article.

To launch the process of creation a new contact, enter the following URL in the browser address bar:

```
http[s]://<Creatio_application_address>/0/ServiceModel/ProcessEngineService.svc/UsrAd
dNewExternalContact/Execute?ContactName=John Johanson&ContactPhone=+1 111 111 1111
```

After executing the request, a new contact will be added to Creatio (Fig. 7).

Fig. 7. New contact

The screenshot shows the Creatio Contacts module interface. At the top, there are navigation icons for 'Contacts' (with a bar chart icon), 'Actions' (with a gear icon), and a search bar 'What can I do for you?'. On the right, the 'Creatio' logo is visible. Below the header, there are buttons for 'NEW CONTACT' (green) and 'ACTIONS ▾' (grey). Underneath, there are filters for 'Filter' and 'Tag'. A table displays contact details: Contact name (John Johanson), Account (Accom (sample)), Job title (Specialist), Business phone (+1 617 440 2031), Mobile phone (+1 617 221 5187), and Email (a.baker@ac.com). Below the table, there are buttons for 'OPEN' (blue), 'COPY', 'DELETE', and 'PRINT'. The contact information for Andrew Baker (sample) is also shown: Supervisor (Our company), Email Supervisor (a.baker@ac.com). The contact for John Johanson is highlighted with a red border.

A new contact will be created after each successful request to the service. If you run a number of queries with the same parameters, multiple duplicate contacts will be created.

To launch the process of reading all contacts, enter the following URL in the browser address bar:

```
http[s]://<Creatio_application_address>/0/ServiceModel/ProcessEngineService.svc/UsrGetAllContacts/Execute?ResultParameterName=ContactList
```

After executing the request, a JSON object with collection of contacts will be displayed in the browser window (Fig. 8).

Fig. 8. Result of the contacts reading process

The screenshot shows a browser window with the URL 'imuta/0/ServiceModel/ProcessEngineService.svc/UsrGetAllContacts/Execute?ResultParameterName=ContactList'. The page content is an XML document with the following structure:

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">
  [{"Id": "\0e9dccfa-0c73-4ca4-970e-323c2361f690", "Name": "John Johanson", "Phone": "+1 111 111 1111"}, {"Id": "c4ed336c-3e9b-40fe-8b82-5632476472b4", "Name": "Andrew Baker (sample)", "Phone": "+1 617 440 2031"}, {"Id": "eaca1536-1212-4154-a434-9d01c0825304", "Name": "John Johanson", "Phone": "+1 111 1111"}, {"Id": "410006e1-ca4e-4502-a9ec-e54d922d2c00", "Name": "Supervisor", "Phone": ""}, {"Id": "5266908c-f3d1-4c5c-9097-f0dfbdbf900b", "Name": "Email Supervisor", "Phone": ""}]
</string>
```

The contact for John Johanson is highlighted with a red border.

3. Run business processes from the third-party application

Before making requests to *ProcessEngineService.svc*, a third party application must be authenticated. Use the *AuthService.svc* authentication service for this. Information and examples of authentication of third-party application can be found in the “**The AuthService.svc authentication service (on-line documentation)**” article. Console application created according the example can be used for the case below.

Full source code of the console application used for running business processes via the *ProcessEngineService.svc* service is available [by a link](#).

To generate requests to the *ProcessEngineService.svc* service, add a string field that contains base service URL to the *Program* class source code:

```
private const string processServiceUri = baseUri +  
@"/0/ServiceModel/ProcessEngineService.svc/";
```

To run the business process of adding a new contact, add the following method to the source code of the *Program* class:

```
public static void AddContact(string contactName, string contactPhone)  
{  
    // Generating the URL request.  
    string requestString = string.Format(processServiceUri +  
        "UsrAddNewExternalContact/Execute?ContactName={0}&ContactPhone={1}",  
        contactName, contactPhone);  
    // Generating Http request.  
    HttpWebRequest request = HttpWebRequest.Create(requestString) as HttpWebRequest;  
    request.Method = "GET";  
    request.CookieContainer = AuthCookie;  
    // Execute the request and analyze the Http response.  
    using (var response = request.GetResponse())  
    {  
        // Because the service returns an empty string,  
        // you can display the http response properties.  
        Console.WriteLine(response.ContentLength);  
        Console.WriteLine(response.Headers.Count);  
    }  
}
```

Add a method of starting the process of reading contacts:

```
public static void GetAllContacts()  
{  
    // Generating the URL request.  
    string requestString = processServiceUri +  
        "UsrGetAllContacts/Execute?ResultParameterName=ContactList";  
    HttpWebRequest request = HttpWebRequest.Create(requestString) as HttpWebRequest;  
    request.Method = "GET";  
    request.CookieContainer = AuthCookie;  
    // Generating Http request.  
    using (var response = request.GetResponse())  
    {  
        // Executing the request and output the result.  
        using (var reader = new StreamReader(response.GetResponseStream()))  
        {  
            string responseText = reader.ReadToEnd();  
            Console.WriteLine(responseText);  
        }  
    }  
}
```

The added methods can be called in the main program method after successful authentication:

```
static void Main(string[] args)  
{  
    if (!TryLogin("Supervisor", "Supervisor"))  
    {  
        Console.WriteLine("Wrong login or password. Application will be  
terminated.");  
    }  
    else  
    {  
        try  
        {  
            // Calling methods for starting business processes.  
        }
```

```

        AddContact("John Johanson", "+1 111 111 1111");
        GetAllContacts();
    }
    catch (Exception)
    {
        // Exception Handling.
        throw;
    }
}
Console.WriteLine("Press ENTER to exit...");
Console.ReadLine();
}

```

The program result is shown in Fig. 9.

Fig. 9. Result of executing the application

```

0
8
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">[{"Id": "0e9dccfa-0c73-4ca4-970e-323c2361f690", "Name": "John Johanson", "Phone": "+1 111 111 1111"}, {"Id": "a0c4b7fe-8060-4611-8c91-35e339f524f1", "Name": "John Johanson", "Phone": "+1 111 111 1111"}, {"Id": "c4ed336c-3e9b-40fe-8b82-5632476472b4", "Name": "Andrew Baker (sample)", "Phone": "+1 617 440 2031"}, {"Id": "d6799742-ae56-4149-8f55-cbddc7387d0b", "Name": "John Johanson", "Phone": "+1 111 111 1111"}, {"Id": "410006e1-ca4e-4582-a9ec-e54d922d2c00", "Name": "Supervisor", "Phone": "", "Id": "5266908c-f3d1-4c5c-9097-f0dfbdbf900b", "Name": "Email Supervisor", "Phone": ""}]</string>
Press ENTER to exit...
-
```

How to save the record without closing the edit page which is opened by the business process

[Beginner](#)

[Easy](#)

[Medium](#)

[Advanced](#)

Introduction

If the record edit page is opened by the [Open edit page] business process element, the saving of the record (by clicking the [Save] button or by the `this.save()` method in the schema source code) will cause the closing of the page. Edit page is being closed even if the [Open edit page] element is not complete (configured in the [When is the element considered complete?] element property).

If you need to save the record several times without closing the edit page, pass the configuration object with the `isSilent` property set to `true` to the `this.save()` method. Example:

```
this.save({isSilent : true});
```

Case description

Create a business process that will open the invoice edit page. Save the Id of the edited record in the process parameter. In the source code of the edit page schema implement the program logic of saving the record each time the [Product in invoice] detail is being modified. Ensure the ability to edit detail records without closing the invoice edit page.

Case implementation

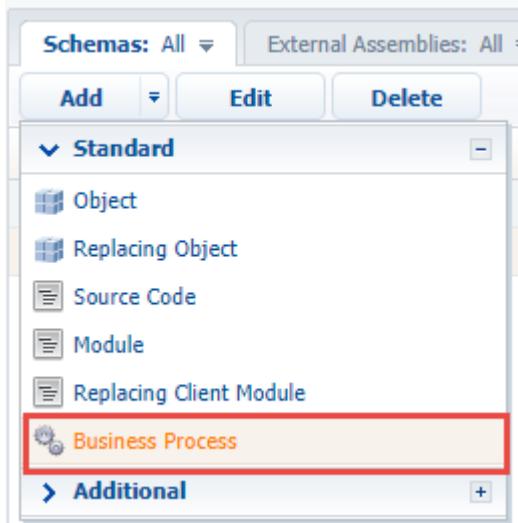
1. Business process creation

To do so, execute the following steps.

1.1 Create business process

In the [Configuration] section execute the [Add] – [Business process] action (Fig. 1).

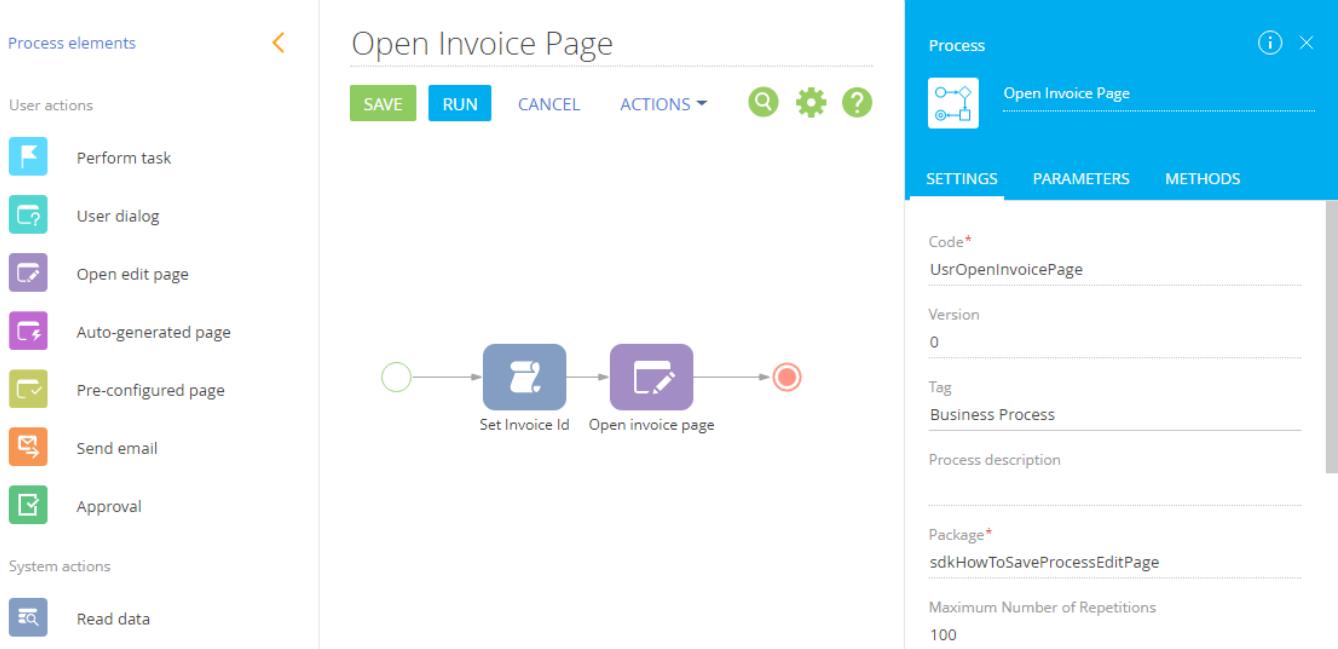
Fig. 1. [Add] – [Business process] action



In the opened process designer set the following values for the properties (Fig. 2):

- [Title] – "Open Invoice Page".
- [Code] – "UsrOpenInvoicePage".

Fig. 2. The properties of the business process



1.2 Add parameter

Add the parameter to the business process created on the previous step. This parameter will store the Id of the opened order record. Set following properties for the parameter (Fig. 3):

- [Title] – "Invoice Id".
- [Code] – "InvoiceId".
- [Data type] – "Unique identifier".

Fig. 3. The properties for the parameter of the business process

SETTINGS PARAMETERS METHODS

ADD PARAMETER ▾

Title

Code

Data type

Value

SAVE **CANCEL**

1.3 Add the [ScriptTask] element

You can find the value of the invoice record Id from the browser navigation bar by opening a record for editing (Fig. 4).

Fig. 4. Getting the record Id

INV-1 (sample)

Sales

Dashboards

Feed

Leads

Accounts

Contacts

Activities

Opportunities

CLOSE ACTIONS VIEW

GENERAL INFORMATION

Number INV-1 (sample) Date * 10/1/2016

Owner * Supervisor Order ORD-1 (sample)

Customer * Accom (sample) Customer details Billing, USD

Supplier Our company Supplier details For invoices (USD)

Amount

Amount, US Dollar 5,265.00

This value can be saved to the *InvoiceId* parameter by the program code executed by the [ScriptTask] element.

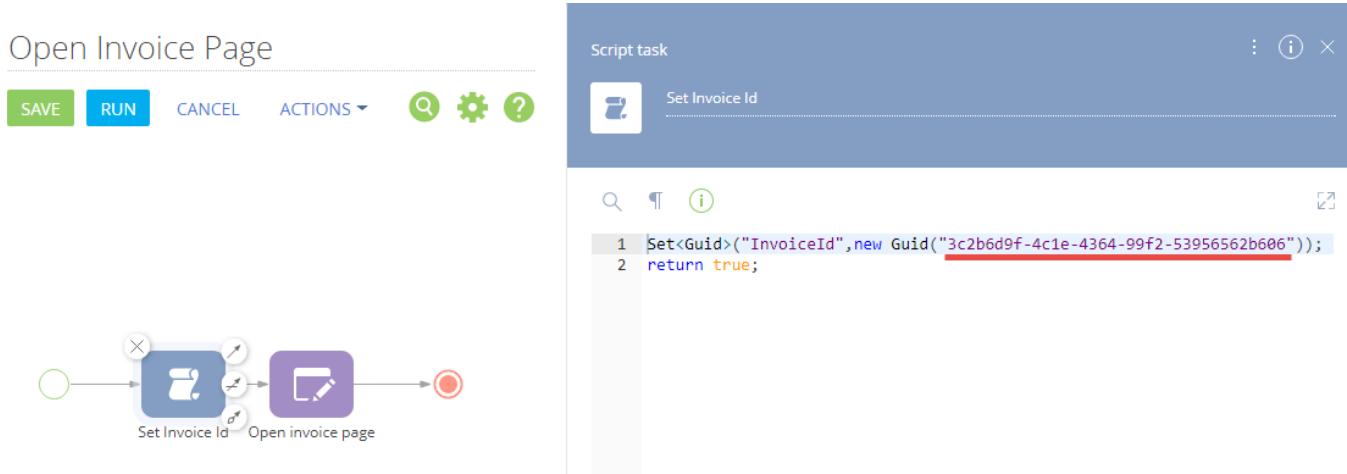
For this add the [ScriptTask] element to the business process. The [Title] property of the element can be set to "Set

Invoice Id". The element can execute following program code

```
Set<Guid>("InvoiceId", new Guid("3c2b6d9f-4c1e-4364-99f2-53956562b606"));
return true;
```

The InvoiceId parameter is set here. The instance of the *Guid* class is created on the basis of the string with the invoice record Id obtained from the browser navigation bar (Fig. 5).

Fig. 5. [ScriptTask] element properties



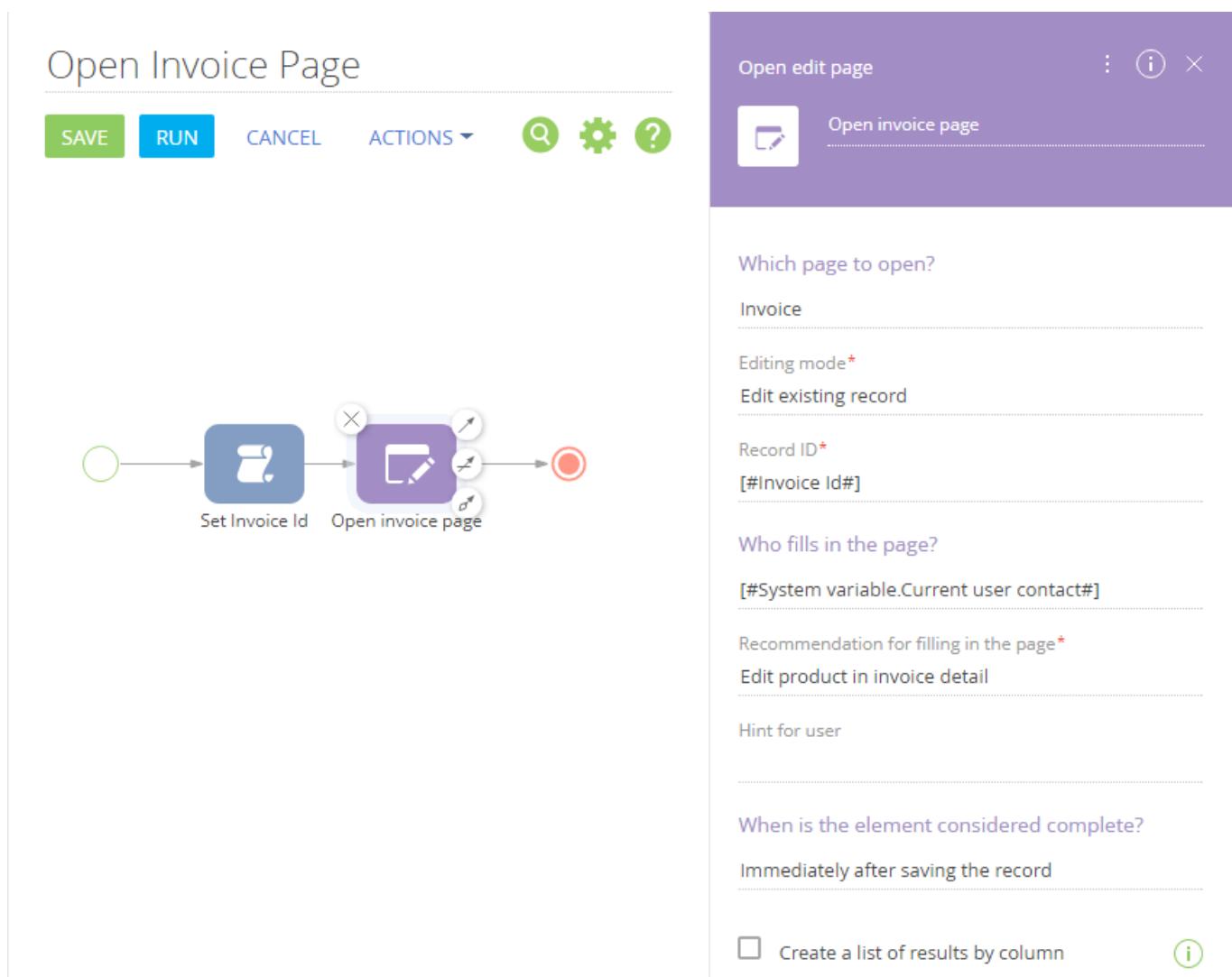
The Id can be obtained by the instance of the *EntitySchemaQuery* class (see “**The use of EntitySchemaQuery for creation of queries in database (on-line documentation)**”).

1.5. Add the [Open edit page] element

Use the [Open edit page] element to open the page for editing during the process execution. Set following properties for this element (Fig. 6):

- [Title] – "Open invoice Page".
- [Which page to open?] [Which page to open?] – "Invoice".
- [Editing mode] – "Edit existing record".
- [Record Id] – select the [Invoice Id] process parameter added on the Step 1.2.
- [Recommendation for filling in the page] – "Edit product in invoice detail".
- [When element is considered complete?] – "Immediately after saving the record".

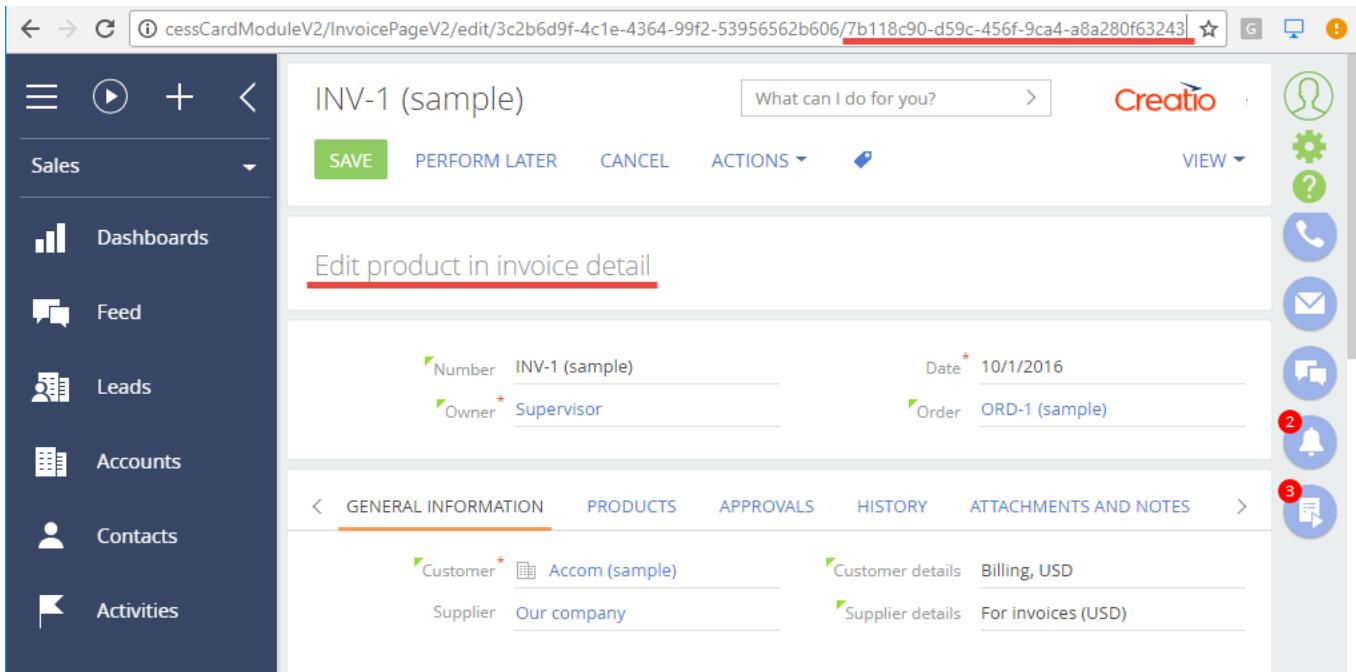
Fig. 6. [Open edit page] element properties



Save the business process to apply changes.

The start of the business process will open the record edit page which will be automatically closed when saving (Fig. 7).

Fig. 7. Edit page opened by the business process



2. Add the program logic to the edit page schema

To save the record when modifying the [Product in invoice] detail without closing the edit page, execute the following steps.

2.1 Add a replacing schema of the invoice edit page

The procedure for creating a replacing schema of the edit page is covered in the **“Creating a custom client module schema”** article. Select the "Invoice edit page" (*InvoicePageV2*) schema as a parent object.

2.2 Override the `onDetailChanged()` method

In the replacing schema of the invoice edit page override the `onDetailChanged()` method implemented in the *BaseEntityPage* base schema. This method is the handler of the received message about modification of the detail on the edit page.

To ensure editing of the records of the [Product in invoice] detail without closing the invoice edit page, add the following source code to the schema.

```
define("InvoicePageV2", [], function() {
    return {
        entitySchemaName: "Invoice",
        methods: {
            // The handler for the detail change message.
            onDetailChanged: function(detail, args) {
                this.callParent(arguments);
                // Only for the [Products in invoice] detail
                if (detail.schemaName === "InvoiceProductDetailV2") {
                    // Save a record with the automatic closing of the edit page.
                    //this.save();
                    // Save the record without closing the edit page.
                    this.save({isSilent : true});
                }
            }
        },
        diff: /**SCHEMA_DIFF*/ [
        ]/**SCHEMA_DIFF*/
    };
});
```

```
});
```

Save the schema to apply changes.

As a result, after start of the business process the record edit page will open (Fig. 7). The page will be closed only after clicking the [Save] button. The record opened for edit will be saved after each modification of the [Product in invoice] detail without closing the edit page.

Typical customizations

Contents

- **Creating pop-up summaries (mini pages)**
- **Adding pop-up summaries (mini pages) to a module**
- **Creating a pop-up summary (mini page) for adding records**
- **Adding pop-up hints**
- **How to modify sales pipeline calculations**
- **How to enable additional filtering in a sales pipeline**
- **Adding a custom dashboard widget**
- **Using the Terrasoft.AlignableContainer custom control**
- **Adding a duplicate search rule**
- **Adding a rule for duplicates search when saving a record**
- **Junk case custom filtering**
- **How to display custom implementation of approving in the section wizard**
- **How to create custom reminders and notifications**
- **Creating the [Timeline] tab tiles bound to custom section**
- **Adding multi-language email templates to a custom section**
- **Integration of third-party sites via iframe**

Creating pop-up summaries (mini pages)

Beginner

Easy

Medium

Advanced

Introduction

Starting from version 7.7 we have introduced a new module in Creatio - a pop-up summary. For regular users, pop-up summaries are improved screen tips containing additional functions based on the current section. Using pop-up summaries enables receiving information about the account address and opening its location on a map, sending emails or making contact calls directly from the section without opening the edit page. You can see examples of pop-up summaries by hovering the cursor over hyperlinks pointing at edit pages in the [Accounts] and [Contact] sections.

Primary purposes of using pop-up summaries:

- Enabling users to get the necessary information by record without opening edit pages.
- Providing possibility to quickly add records to sections with populating only the required fields without opening full record pages.

The structure of a pop-up summary view model schema does not differ from the general structure of Creatio module schema. Required properties of pop-up summary schema structure include:

- *entitySchemaName* containing the object schema name bound to a pop-up summary
- the *diff* modification array

These parameters enable building a module view in Creatio custom interface.

You can also use other general schema structure elements to implement the necessary functions, such as attributes, methods, mixins and messages that can be used to add custom control elements, register messages and form the pop-up summary business logics. The appearance of pop-up summary visual elements can be modified using custom styles.

Pop-up summaries do not support the mechanism of business logics setup via business rules.

To add a custom pop-up summary to a current Creatio section:

1. Add a pop-up view model schema to the custom package. Select *BaseMiniPage* schema as a parent object.
2. Modify the *SysModuleEdit* system table in the Creatio database via a special SQL query.
3. Add the necessary pop-up summary functions to the schema source code. Specify the object schema name in the *entitySchemaName* element bound to the pop-up summary and perform at least one modification in the *diff* array.
4. Apply styling to the pop-up summary.
5. Add the *Has[Section code]MiniPageAddMode* setting.

To bind a pop-up summary to specific section objects, specify the unique pop-up summary identifier in the *MiniPageSchemaUid* column of these objects. Currently you can only do it by modifying the *SysModuleEdit* system table of Creatio database via an SQL-query.

Pay high attention to creating and executing the SQL query. Executing an incorrect SQL query can damage the existing data and disrupt the system.

For Creatio sections with default pop-up summaries, there are system settings, whose codes have the following format *Has[Section code]MiniPageAddMode* (for instance, *HasAccountMiniPageAddMode*). These system settings are used to toggle between the two modes: adding new records and editing existing records via pop-up summaries.

You can create a pop-up summary for any Creatio object.

Case description

Creating a custom pop-up summary for the [Knowledge base] section. The pop-up summary will be used for viewing the basic [Name] and [Tags] fields with a possibility to download the attached files.

Source code

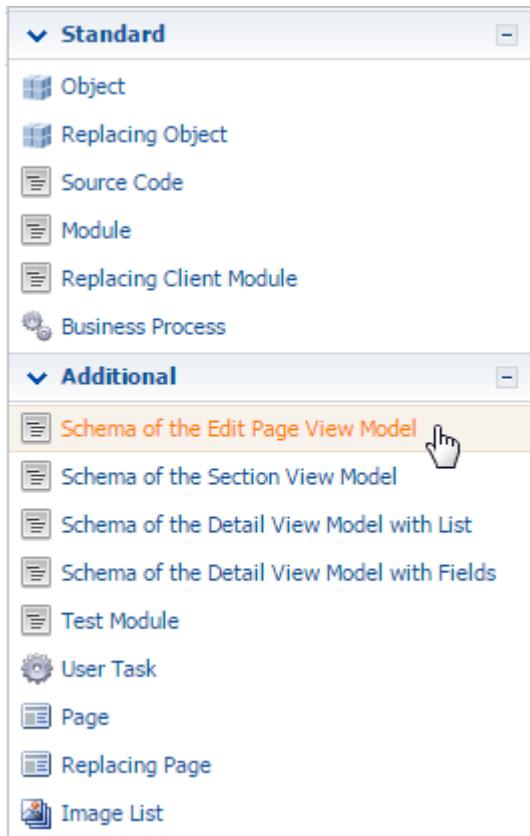
Use the following [link](#) to download a package with the [Knowledge base] section pop-up summary schema implemented according to this case.

Case implementation algorithm

1. Creating a pop-up summary view model schema

Open the [Schemas] tab in the [Configuration] section and select the [Add] — [Schema of the Edit Page View Model] command from the menu (Fig.1).

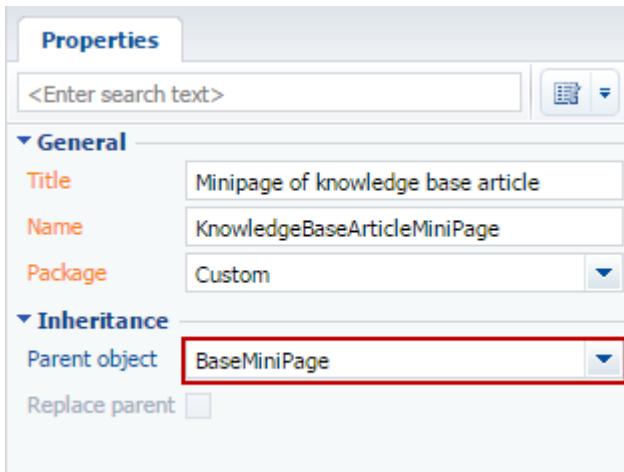
Fig.1 Adding a pop-up summary view schema



Populate the following properties of the pop-up summary view schema (Fig.2):

- [Title] – “UsrKnowledgeBaseArticleMiniPage”
- [Name] – “KnowledgeBase Mini Page”
- [Package] – the custom package, in which the development is performed, for instance, *UsrPackage*
- [Parent object] - the *BaseMiniPage* schema from the *NUI* package

Fig.2 Properties of the pop-up summary view model schema



2. Registering a the pop-up summary in the database

Execute the following SQL query to perfrom modifications in the database:

```
DECLARE
    -- Name of the created pop-up summary view schema.
    @ClientUnitSchemaName NVARCHAR(100) = 'UsrKnowledgeBaseArticleMiniPage',
    -- Name of the object schema bound to the pop-up summary.
    @EntitySchemaName NVARCHAR(100) = 'KnowledgeBase'
```

```

UPDATE SysModuleEdit
SET MiniPageSchemaUid = (
    SELECT TOP 1 UId
    FROM SysSchema
    WHERE Name = @ClientUnitSchemaName
)
WHERE SysModuleEntityId = (
    SELECT TOP 1 Id
    FROM SysModuleEntity
    WHERE SysEntitySchemaUid = (
        SELECT TOP 1 UId
        FROM SysSchema
        WHERE Name = @EntitySchemaName
        AND ExtendParent = 0
    )
);

```

As a result of this query execution you will have a unique pop-up identifier, populated in *SysModuleEdit* table of the record *MiniPageSchemaUid* field that corresponds to the [Knowledge base] section (Fig.3).

Fig.3 Unique pop-up summary identifier value in *SysModuleEdit* table

CardSchemaUId	ActionKindCaption	ActionKindName	PageCaption	MiniPageSchemaUid
9DBD0611-FA52-4A90-9542-E5FD997B4AFD	New article	KnowledgeBase	Knowledge base article	48A427D4-0BF8-4017-A1CC-4A1F96C39243
NULL				NULL
38B9E577-152C-4EAC-8A68-C296183B690F				NULL
625E1D4C-BC26-4872-B76E-267C473ECD...				NULL

3. Displaying primary object fields

The pop-up summary code structure is identical to the edit page structure. Specify the *KnowledgeBase* schema as the object schema and add the necessary modifications to the *diff* view model modification array.

The base pop-up summary consists of the following elements:

- *MiniPage* – *Terrasoft.GridLayout* – pop-up summary field
- *HeaderContainer* – *Terrasoft.Container* – pop-up summary name (initially is placed in the first row of the pop-up summary field)

Two objects that configure the [Name] and [Keywords] fields are added to the *diff* modification array.

At this stage the pop-up summary can already be used and the following actions are not required.

Source code of the pop-up summary view model schema:

```

define("UsrKnowledgeBaseArticleMiniPage", [], function() {
    return {
        entitySchemaName: "KnowledgeBase",
        attributes: {
            "MiniPageModes": {
                "value": [this.Terrasoft.ConfigurationEnums.CardOperation.VIEW]
            }
        },
        diff: /**SCHEMA_DIFF*/ [
            {
                "operation": "insert",
                "name": "Name",
                "parentName": "HeaderContainer",
                "propertyName": "items",
                "index": 0,
                "values": {
                    "labelConfig": {
                        "visible": false
                    },
                    ...
                }
            }
        ]
    }
});

```

```

        "isMiniPageModelItem": true
    }
},
{
    "operation": "insert",
    "name": "Keywords",
    "parentName": "MiniPage",
    "propertyName": "items",
    "values": {
        "labelConfig": {
            "visible": false
        },
        "isMiniPageModelItem": true,
        "layout": {
            "column": 0,
            "row": 1,
            "colSpan": 24
        }
    }
}
] /*SCHEMA_DIFF*/
);
});

```

4. Adding a function button to the pop-up summary

As per the example conditions, the pop-up summary must enable downloading files bound to the knowledge base.

You can access additional data via a drop-down list of a pre-configured button. To add a button of selecting files from the knowledge base article:

1. Add the button description to the *diff* array – the *FilesButton* element.
2. Add an attribute binding the primary and additional records – the *Article* virtual column.
3. Add the *MiniPageModes* attribute – the array containing a collection of necessary operations performed by the pop-up summary.
4. Add the button image to Creatio resources. For example, you can add the following image –  . Adding an image to resources is covered in the "**How to add a field with an image to the edit page**" article.
5. To add methods of working with a drop-down list of a file selection button:
 - override the *init()* method.
 - override the *onEntityInitialized()* method
 - set the *Article* attribute value via the *setArticleInfo()* method
 - get information about the current knowledge base article files via the *initFilesMenu(files)* method
 - populate the drop-down list collection of the file selection button via the *initFilesMenu(files)* method
 - initiate the file upload and adding to the drop-down list of the file selection button via the *fillFilesExtendedMenuData()* method
 - initiate the selected file download via the *downloadFile()* method

5. Applying styling to the pop-up summary.

Create the *UsrKnowledgeBaseArticleMiniPageCss* module and specify the necessary styles on the LESS tab.

```
div[data-item-marker="UsrKnowledgeBaseArticleMiniPageContainer"] > div {
    width: 250px;
}
```

Specify the pop-up summary schema dependency on the style module in the page designer and add this module download in the source code.

Below is the full source code of a pop-up summary:

```
define("UsrKnowledgeBaseArticleMiniPage",
["terrasoft", "KnowledgeBaseFile", "ConfigurationConstants",
"css!UsrKnowledgeBaseArticleMiniPageCss"],
function(Terrasoft, KnowledgeBaseFile, ConfigurationConstants) {
    return {
        entitySchemaName: "KnowledgeBase",
        attributes: {
            "MiniPageModes": {
                "value": [this.Terrasoft.ConfigurationEnums.CardOperation.VIEW]
            },
            "Article": {
                "type": Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                "referenceSchemaName": "KnowledgeBase"
            }
        },
        methods: {
            // Initiates the drop-down list collection of the file selection
            button.
            init: function() {
                this.callParent(arguments);
                this.initExtendedMenuButtonCollections("File", ["Article"]),
                this.close();
            },
            // Initiates the attribute value binding the primary and additional
            records.
            // Populates the drop-down list collection of the file selection
            button.
            onEntityInitialized: function() {
                this.callParent(arguments);
                this.setArticleInfo();
                this.fillFilesExtendedMenuData();
            },
            // Initiates the file download and adding to the drop-down list of
            the file selection button.
            fillFilesExtendedMenuData: function() {
                this.getFiles(this.initFilesMenu, this);
            },
            // Sets the attribute value binding the primary and additional
            records.
            setArticleInfo: function() {
                this.set("Article", {
                    value: this.get(this.primaryColumnName),
                    displayValue: this.get(this.primaryDisplayColumnName)
                });
            },
            // Receives information about files of the current knowledge base
            article.
            getFiles: function(callback, scope) {
                var esq = this.Ext.create("Terrasoft.EntitySchemaQuery", {
                    rootSchema: KnowledgeBaseFile
                });
                esq.addColumn("Name");
                var articleFilter =
                this.Terrasoft.createColumnFilterWithParameter(
                    this.Terrasoft.ComparisonType.EQUAL, "KnowledgeBase",
                this.get(this.primaryColumnName));
                var typeFilter = this.Terrasoft.createColumnFilterWithParameter(
                    this.Terrasoft.ComparisonType.EQUAL, "Type",
                ConfigurationConstants.FileType.File);
                esq.filters.addItem(articleFilter);
                esq.filters.addItem(typeFilter);
                esq.getEntityCollection(function(response) {
```

```
        if (!response.success) {
            return;
        }
        callback.call(scope, response.collection);
    }, this);
},
// Populates the drop-down list collection of the file selection
button.
initFilesMenu: function(files) {
    if (files.isEmpty()) {
        return;
    }
    var data = [];
    files.each(function(file) {
        data.push({
            caption: file.get("Name"),
            tag: file.get("Id")
        });
    }, this);
    var recipientInfo = this.fillExtendedMenuItems("File",
["Article"]);
    this.fillExtendedMenuData(data, recipientInfo,
this.downloadFile);
},
// Initiates the selected file download.
downloadFile: function(id) {
    var element = document.createElement("a");
    element.href = "../rest/FileService/GetFile/" +
KnowledgeBaseFile.uId + "/" + id;
    document.body.appendChild(element);
    element.click();
    document.body.removeChild(element);
}
},
diff: /**SCHEMA_DIFF*/[
{
    "operation": "insert",
    "name": "Name",
    "parentName": "HeaderContainer",
    "propertyName": "items",
    "index": 0,
    "values": {
        "labelConfig": {
            "visible": true
        },
        "isMiniPageModelItem": true
    }
},
{
    "operation": "insert",
    "name": "Keywords",
    "parentName": "MiniPage",
    "propertyName": "items",
    "values": {
        "labelConfig": {
            "visible": true
        },
        "isMiniPageModelItem": true,
        "layout": {
            "column": 0,
            "row": 1,
            "colSpan": 24
        }
    }
}];
```

```
        }
    },
},
{
    "operation": "insert",
    "parentName": "HeaderContainer",
    "propertyName": "items",
    "name": "FilesButton",
    "values": {
        "itemType": Terrasoft.ViewItemType.BUTTON,
        // Button image setup.
        "imageConfig": {
            // You need to preliminary add the image to the pop-up
summary resources.
            "bindTo": "Resources.Images.FilesImage"
        },
        // Drop-down list setup.
        "extendedMenu": {
            // Drop-down list element name.
            "Name": "File",
            // The name of pop-up summary attribute binding the
primary and additional records.
            "PropertyName": "Article",
            // Setup of button click handler.
            "Click": {
                "bindTo": "fillFilesExtendedMenuData"
            }
        }
    },
    "index": 1
},
]/**SCHEMA_DIFF*/
);
});
```

6. Adding the HasProductMiniPageAddMode system setting

Add a system setting with the following properties to the [System settings] section of the system designer (Fig.4):

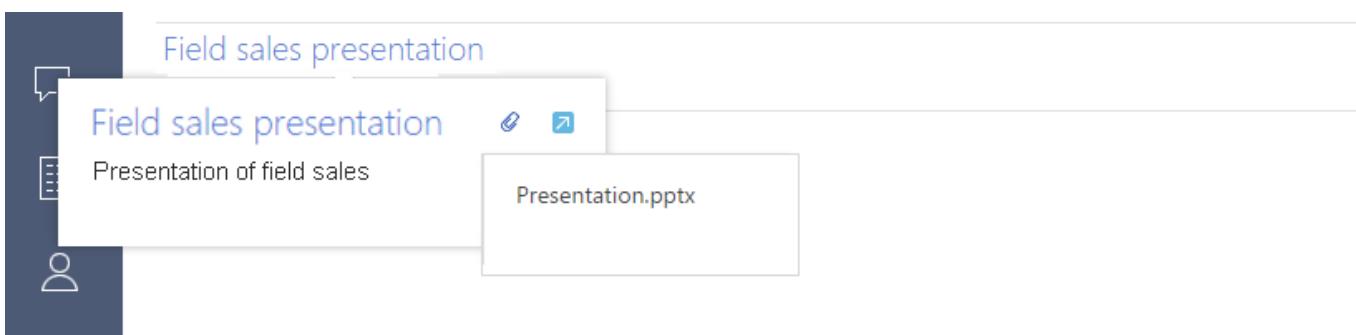
- [Name] – “HasKnowledgeBaseMiniPageAddMode”
- [Code] – “HasKnowledgeBaseMiniPageAddMode”
- [Type] – “Boolean”
- [Default value] – checkbox selected

Fig. 4. System setting

The screenshot shows the 'HasKnowledgeBaseMiniPageAddMode' schema element configuration. It includes fields for Name (HasKnowledgeBaseMiniPageAddMode), Type (Boolean), Default value (checked), Code (HasKnowledgeBaseMiniPageAddMode), Cached (checked), Personal (unchecked), and Allow for portal users (unchecked). There is also a Description field.

After you save the schema and update the application web-page, a custom pop-up summary containing the record bound files will be displayed when you hover over a name in the [Knowledge base] section. You will be able to download the displayed files (Fig.5).

Fig. 5. Case result



Adding pop-up summaries (mini pages) to a module

Beginner Easy Medium **Advanced**

Introduction

When adding object pop-up summaries, it sometimes becomes required to connect them to Creatio modules. **Modules** enable creating links to specific objects in Creatio. A pop-up summary displayed upon hovering over such a link provides additional information about the object without opening the object section.

In Creatio base version, an object pop-up summary is connected to the following modules:

- telephony in the communication panel
- email in the communication panel
- notification center in the communication panel
- the [Feed] section in the communication panel
- chart-list in the dashboards section

Case description

Display the current Creatio user in the application top right corner next to the user profile. Open a pop-up summary upon hovering over the current Creatio user link.

Source code

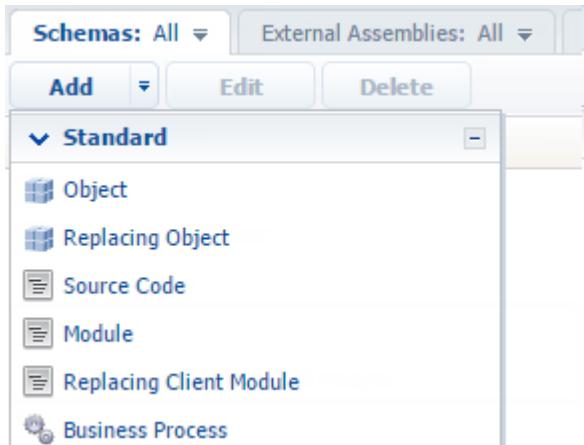
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Creating a module

Perform the [Add] – [Standard] – [Module] menu command on the [Schemas] tab in the [Configuration] section (Fig. 1).

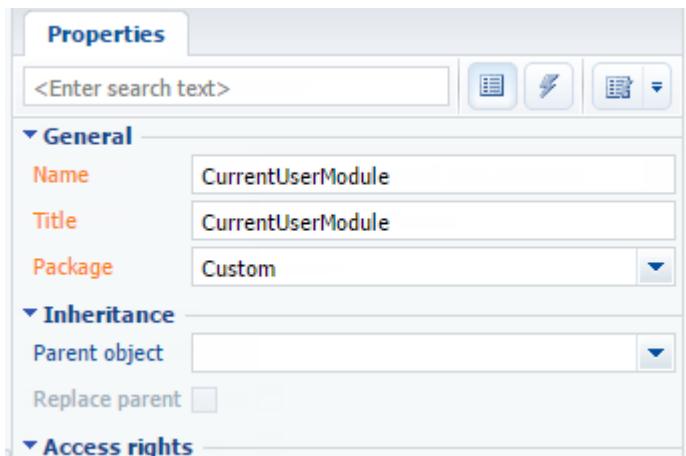
Fig. 1. Adding a module



Specify properties for the created module (Fig. 2):

- [Name] – “UsrCurrentUserModule”
- [Title] – “Current user module”

Fig. 2. Module properties



2. Creating a view and a module view model

To create a view model in the `UsrCurrentUserModule` module, implement the class inherited from `Terrasoft.BaseViewModel`. Connect the `Terrasoft.MiniPageUtilities` utility class to the `mixins` property of the module view model, which enables using the pop-up summary call methods.

To create the view, implement the class inherited from `Terrasoft.BaseModule`.

Override the `init()` and `render()` methods of the `Terrasoft.BaseModule` base class in the created class. The `init()` method initializes the module view model and the `render()` method connects the view model with the view display in container rendered in the `renderTo` parameter. To create the view model, use the `getViewModel()` method. The link to the received view model is stored in the `viewModel` property.

Define the `getView()` method for receiving the view for its further display. The view must display the full name of the current user and a hyperlink to the contact edit page. When creating a hyperlink, define the event handler of hovering over the mouse cursor.

Below you can find the complete source code:

```
// Defining the module.
define("UsrCurrentUserModule", ["MiniPageUtilities"], function() {
    // Defining the CurrentUserViewModel class.
    Ext.define("Terrasoft.configuration.CurrentUserViewModel", {
        // Parent class name.
        extend: "Terrasoft.BaseViewModel",
        // Shortened class name.
        alternateClassName: "Terrasoft.CurrentUserViewModel",
        // Used mixins.
        mixins: {
            MiniPageUtilitiesMixin: "Terrasoft.MiniPageUtilities"
        }
    });
    // Defining the UsrCurrentUserModule class.
    Ext.define("Terrasoft.configuration.UsrCurrentUserModule", {
        // Shortened class name.
        alternateClassName: "Terrasoft.UsrCurrentUserModule",
        // Parent class name.
        extend: "Terrasoft.BaseModule",
        // The Ext object.
        Ext: null,
        // The sandbox object.
        sandbox: null,
        // The Terrasoft object.
        Terrasoft: null,
        // View model.
        viewModel: null,
        // Creates module views.
        getView: function() {
            // Receiving the contact of the current user.
            var currentUser = Terrasoft.SysValue.CURRENT_USER_CONTACT;
            // View – the Terrasoft.Hyperlink class instance.
            return Ext.create("Terrasoft.Hyperlink", {
                // Populating the link caption with the contact name.
                "caption": currentUser.displayValue,
                // Event handler of hovering over the link.
                "linkMouseOver": {"bindTo": "linkMouseOver"},
                // The property containing additional object parameters.
                "tag": {
                    // Current user identifier.
                    "recordId": currentUser.value,
                    // Object schema name.
                    "referenceSchemaName": "Contact"
                }
            });
        },
        // Creates module view model.
        getViewModel: function() {
            return Ext.create("Terrasoft.CurrentUserViewModel");
        },
        // Module initialization.
        init: function() {
            this.viewModel = this.getViewModel();
        },
        // Displays the module view.
        render: function(renderTo) {
            // Receiving the view object.
            var view = this.getView();
            // Connecting the view with the view model.
            view.bind(this.viewModel);
```

```

        // Displaying the view in the renderTo element.
        view.render(renderTo);
    }
});

return Terrasoft.UsrCurrentUserModule;
);
}

```

Add styles to the created module for a better display of the hyperlink. To do this, add the following code to the LESS tab of the module designer:

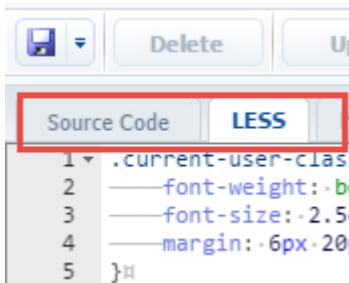
```

.current-user-class a {
    font-weight: bold;
    font-size: 2.0em;
    margin: 6px 20px;
}

.current-user-class a:hover {
    text-decoration: none;
}

```

Fig. 3. The LESS tab of the module designer



Save the created module.

3. Creating the view display container

To display a link in the user profile in the top right corner of the application, locate the container and download the view of the created module into it. Create a replacing client module that would extend the *MainHeaderSchema* schema functionality implemented in the *NUI* package. The procedure for creating a replacing client module is covered in the “**Creating a custom client module schema**” article.

To display the view, use the *diff* property in the replacing schema source code. To display the container in the top right corner of the page, set the *RightHeaderContainer* element as a parent element of the created container. Override the *onRender()* method and download the created module.

Below you can find the complete source code.

```

// Defining a module.
define("MainHeaderSchema", [], function() {
    return {
        methods: {
            // Performs the action after the view display.
            onRender: function() {
                // Calling the parent method.
                this.callParent(arguments);
                // Downloading the module of the current user.
                this.loadCurrentUserModule();
            },
            // Downloads the module of the current user.
            loadCurrentUserModule: function() {
                // Receiving the container for downloading the module.
                var currentUserContainer = this.Ext.getCmp("current-user-container");
                // Verifying if a container is available.
                if (currentUserContainer && currentUserContainer.rendered) {

```

```
// Downloading the module into a container.
this.sandbox.loadModule("UsrCurrentUserModule", {
    // Container name.
    renderTo: "current-user-container"
});
}

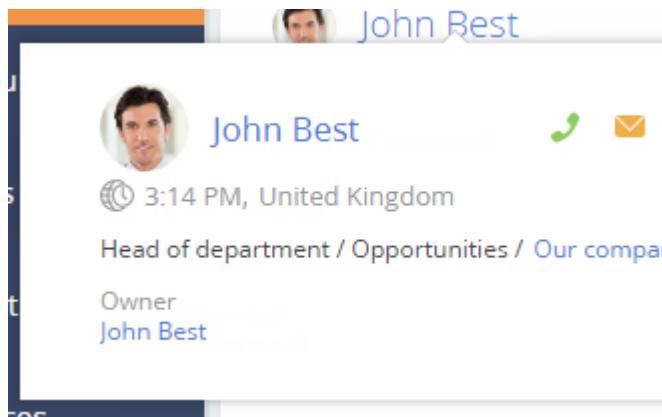
},
diff: [
{
    // Element insert operation.
    "operation": "insert",
    // Element name.
    "name": "CurrentUserContainer",
    // Parent container name.
    "parentName": "RightHeaderContainer",
    // Property name.
    "propertyName": "items",
    // Element values.
    "values": {
        // Container identifier.
        "id": "current-user-container",
        // Element type.
        "itemType": Terrasoft.ViewItemType.CONTAINER,
        // Conatiner classes.
        "wrapClass": ["current-user-class"],
        // Container elements.
        "items": []
    }
}
]
};

});
```

Save the created module.

After you update the application page, the full name with a link to the contact edit page will be displayed in the top right corner. When you hover the cursor over the link, a pop-up summary with the current user details will appear (Fig. 4).

Fig. 4. The contact pop-up summary



Creating a pop-up summary (mini page) for adding records

Beginner Easy Medium **Advanced**

Introduction

You can quickly add and view records using Creatio pop-up summaries. Information about adding pop-up summaries that display record details is available in the “**Creating pop-up summaries (mini pages)**” and “**Adding pop-up summaries (mini pages) to a module**” articles.

To implement a custom pop-up summary page for adding new records in an existing section:

1. Add a pop-up view model schema to the custom package. Select *BaseMiniPage* schema as a parent object.
2. Modify the *SysModuleEdit* system table in the Creatio database via a special SQL query.
3. Add the necessary pop-up summary functionality to the schema source code.
4. Add the *HasProductMiniPageAddMode* system setting.

For Creatio sections with default pop-up summaries, there are system settings, whose codes have the following format *Has[Section code]MiniPageAddMode* (for instance, *HasAccountMiniPageAddMode*). These system settings are used to toggle between the two modes: adding new records and editing existing records.

Case description

Create a custom pop-up summary page for adding new records in the [Products] section. The pop-up summary must contain a base set of fields: [Name] and [Code].

Source code

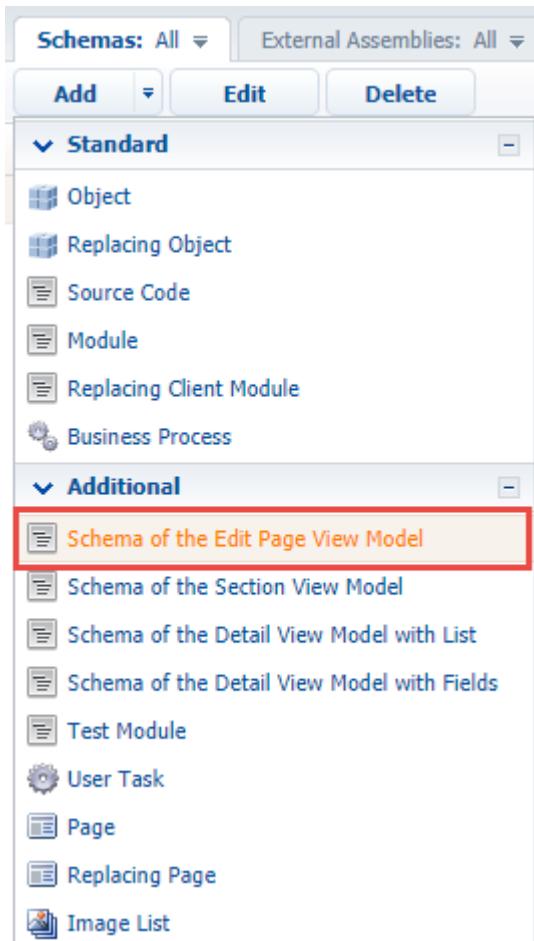
Use the following [link](#) to download a package with the [Products] section pop-up summary schema, implemented according to this case.

Case implementation algorithm

1. Create a pop-up summary view model schema

Execute the [Add] – [Additional] – [Schema of the Edit Page View Model] menu command on the [Schemas] tab in the [Configuration] section (Fig.1).

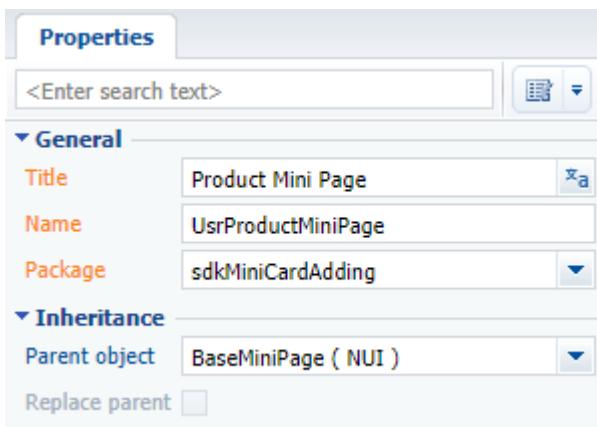
Fig. 1. Adding a pop-up summary view schema



Populate the following properties of the pop-up summary view schema (Fig.2):

- [Name] – “UsrProductMiniPage”.
- [Subject] – “Product Mini Page”.
- [Package] – the custom package, in which the development is performed, for instance, *Custom*.
- [Parent object] – the *BaseMiniPage* schema from the *NUI* package.

Fig. 2. Properties of the pop-up summary view model schema



2. Register the pop-up summary in the database

Execute the following SQL query to make changes in the database:

```
DECLARE
    -- The name of the created pop-up summary view schema.
    @ClientUnitSchemaName NVARCHAR(100) = 'UsrProductMiniPage',
```

```
-- The name of the pop-up summary object schema.
@EntitySchemaName NVARCHAR(100) = 'Product'

UPDATE SysModuleEdit
SET MiniPageSchemaUid = (
    SELECT TOP 1 UId
    FROM SysSchema
    WHERE Name = @ClientUnitSchemaName
)
WHERE SysModuleEntityId = (
    SELECT TOP 1 Id
    FROM SysModuleEntity
    WHERE SysEntitySchemaUid = (
        SELECT TOP 1 UId
        FROM SysSchema
        WHERE Name = @EntitySchemaName
        AND ExtendParent = 0
    )
);

```

As a result of this query execution you will have a unique pop-up identifier, populated in *SysModuleEdit* table of the record *MiniPageSchemaUid* field that corresponds to the [Products] section (Fig.3).

Fig. 3. Unique pop-up summary identifier value in *SysModuleEdit* table

CardSchemaUid	ActionKindCaption	ActionKindName	PageCaption	MiniPageSchemaUid	SearchRowSchemaUid
0DAEC87E-A84D-44BC-9DD0-C90B8D1BAA33	New product	Product	Product	15D85C82-D78F-400F-B10C-44BB13C72282	NULL

Since the changes were made directly in the database, log in to your Creatio again to see them. You may need to compile the application using the corresponding action in the [Configuration] section.

3. Add fields from the primary object to the pop-up summary

Add the source code below to the created pop-up summary view model schema.

```
define("UsrProductMiniPage", ["UsrProductMiniPageResources"],
    function(resources) {
        return {
            entitySchemaName: "Product",
            details: /**SCHEMA_DETAILS*/ {}/**SCHEMA_DETAILS*/,
            attributes: {
                "MiniPageModes": {
                    "value": [this.Terrasoft.ConfigurationEnums.CardOperation.ADD]
                },
                diff: /**SCHEMA_DIFF*/ [
                    {
                        "operation": "insert",
                        "parentName": "MiniPage",
                        "propertyName": "items",
                        "name": "Name",
                        "values": {
                            "isMiniPageModelItem": true,
                            "layout": {
                                "column": 0,
                                "row": 1,
                                "colSpan": 24
                            },
                            "controlConfig": {
                                "focused": true
                            }
                        }
                    }
                ],
            }
        };
    }
);
```

```

    {
        "operation": "insert",
        "parentName": "MiniPage",
        "propertyName": "items",
        "name": "Code",
        "values": {
            "isMiniPageModelItem": true,
            "layout": {
                "column": 0,
                "row": 2,
                "colSpan": 24
            }
        }
    }
] /**SCHEMA_DIFF*/
);
}
);

```

The array containing the collection of the necessary pop-up summary operations is assigned to *MiniPageModes* attribute, which was declared in the base schema. Two objects that configure the [Name] and [Code] fields are added to the *diff* array.

Add *this.Terrasoft.ConfigurationEnums.CardOperation.VIEW* value to the array assigned to *MiniPageModes* attribute if you also need to display the pop-up summary on the section page (see “**Creating pop-up summaries (mini pages)**”).

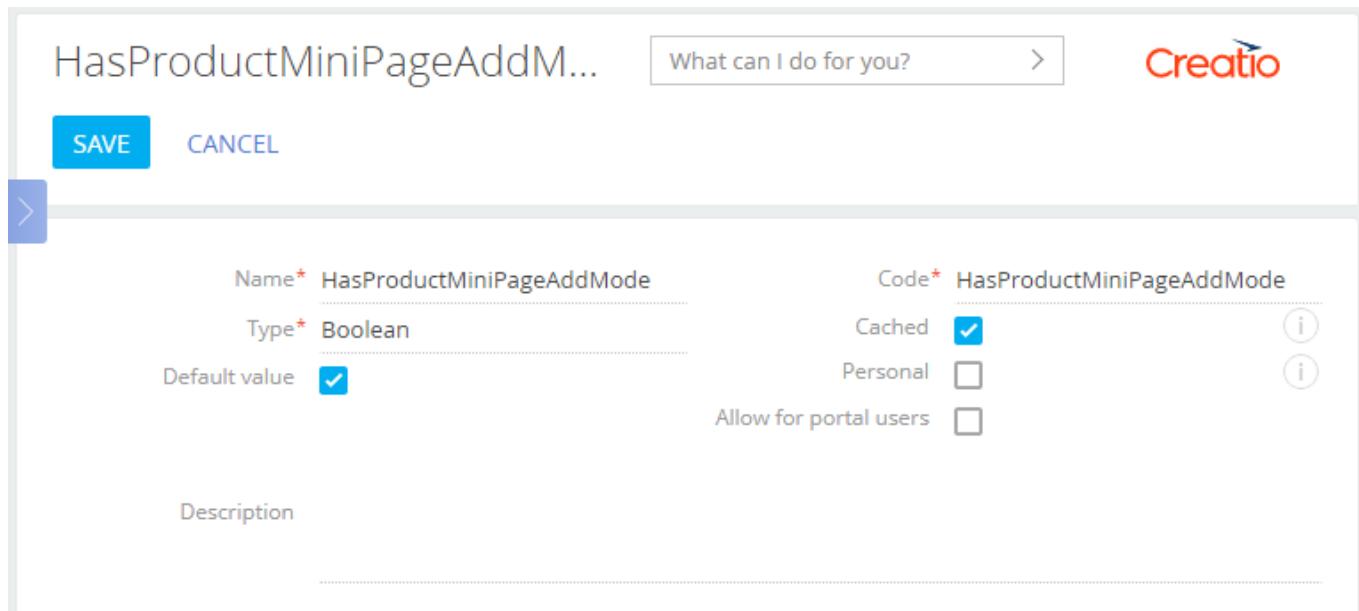
If the required columns are not indicated in the *diff* array, they will be displayed at the bottom of the pop-up summary.

4. Add the HasProductMiniPageAddMode system setting

In the [System settings] section of the system designer, add a new system setting with the following properties (fig. 4)

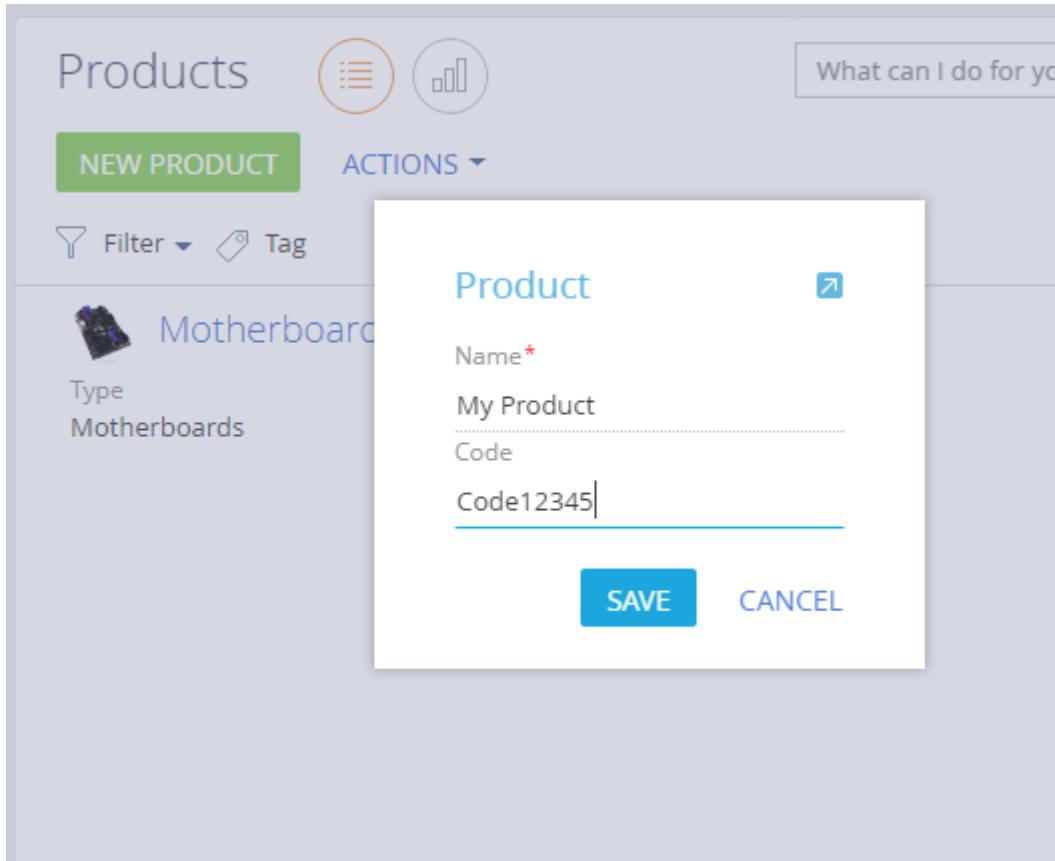
- [Name] – "HasProductMiniPageAddMode".
- [Code] – "HasProductMiniPageAddMode".
- [Type] – "Boolean".
- [Default value] – true.

Fig. 4. – System setting



As a result, a pop-up summary with two fields will be displayed when you add a new product (Fig.4).

Fig. 4. Implemented pop-up summary



After saving the pop-up summary, a corresponding record will appear in the section list (Fig.5).

Fig. 5. Records in the [Products] section

Products		
NEW PRODUCT	ACTIONS ▾	VIEW ▾
<input type="button" value="OPEN"/> <input type="button" value="COPY"/> <input type="button" value="DELETE"/>		
Motherboard UT165LZ-32P1 (sample) Type: Motherboards Price: 900.00 Currency: USD		

The record will only display in the section list after you update the browser page. To display the record immediately after saving the pop-up summary, add the corresponding functions to pop-up summary and the section page schema via the message mechanism (for more information, see “**Module message exchange**”).

Adding pop-up hints

Beginner

Easy

Medium

Advanced

Introduction

You can add pop-up hints to Creatio elements - text messages providing additional information about the element functionality and rules of its population.

Pop-up hints can be divided into 3 main groups:

1. A pop-up hint to a field (if such hint is available, the field caption is marked by a small green triangle symbol. The hint appears when a cursor is hovered over the triangle or upon clicking the field caption).
2. A pop-up hint to other control elements (buttons, completion indicators, images). The hint appears when a cursor is hovered over the control element.

3. Information button  . The hint appears when a cursor is hovered over the information button.

General algorithm of adding pop-up hints to standard control elements:

1. Create a replacing schema of the page or section.
2. Add the pop-up hint text to the schema localizable string collection.
3. Describe the necessary schema element modifications in the diff array.

Source code

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

Example 1

Case description

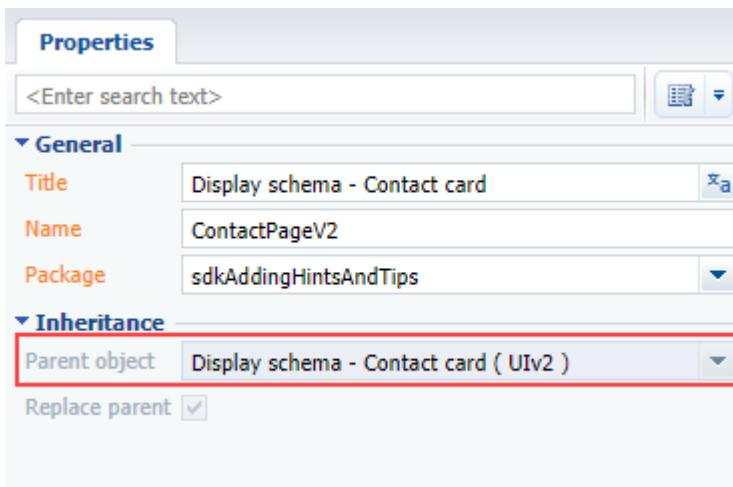
Add a pop-up hint to the [Save] button of contact edit page.

Case implementation algorithm

1. Creating a replacing contact page

Create a replacing client module and specify [Display schema – Contact card] (*ContactPageV2*) as parent object (Fig. 1). Creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 1. Properties of the replacing contact edit page

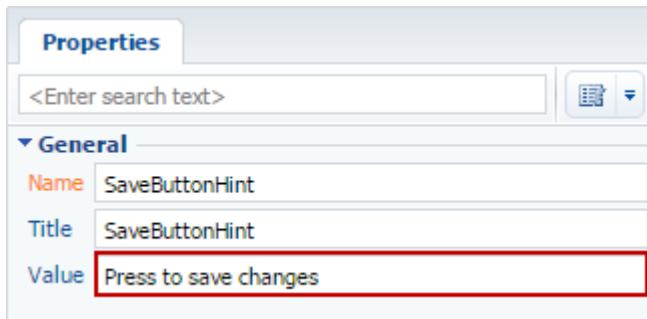


2. Adding a localized string with the pop-up hint text

Add a string with the pop-up hint text to the collection of localizable strings of the edit page replacing schema. Properties of the created string (Fig. 2):

- [Name] – “SaveButtonHint”
- [Value] – “Press to save the changes”

Fig. 2. Properties of the custom localizable string



3. Adding a button configuration object to the `diff` array

There are several methods to add a pop-up hint to a control element.

Method 1

Add the `hint` property containing the pop-up hint text to the control element `values` property.

The source code of the contact edit page replacing schema when adding the pop-up hint using method 1:

```
define("ContactPageV2", [],
function () {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Contact",
        //Pop-up hint visualization setup.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding pop-up hint to the button.
            {
                // Modification of the existing element.
                "operation": "merge",
                "parentName": "LeftContainer",
                "propertyName": "items",
                "name": "SaveButton",
                "values": {
                    // Pop-up hint for a button.
                    "hint": { "bindTo": "Resources.Strings.SaveButtonHint" }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

Method 2

Add the `tips` array to control element `values` property. Add the pop-up hint configuration object to the `tips` array using the `insert` operation. Specify the `content` property, which is the pop-up hint text in the `values` property of this object. You can have a more individual approach to pop-up hint setup using this method. You can change the image style, connect the pop-up hint visibility to a view model event, add control elements, etc.

The specified method works for `itemType`:

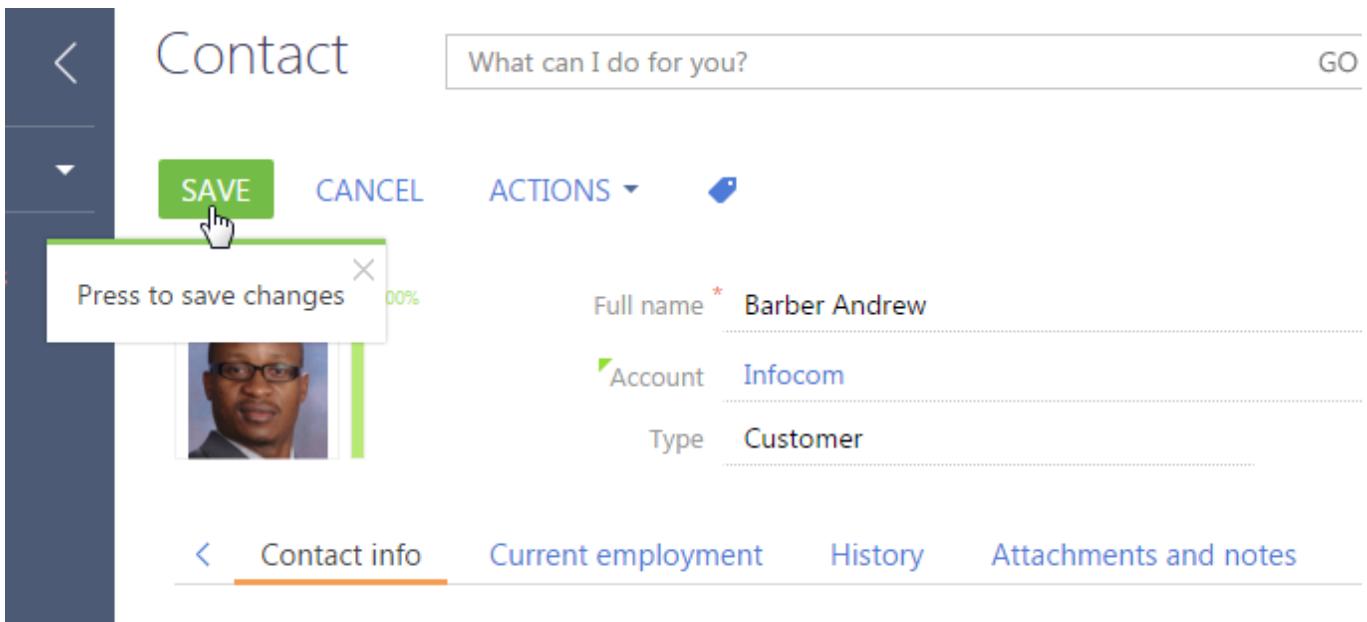
- `Terrasoft.ViewItemType.BUTTON`
- `Terrasoft.ViewItemType.LABEL`
- `Terrasoft.ViewItemType.COLOR_BUTTON`
- `Terrasoft.ViewItemType.HYPERLINK`
- `Terrasoft.ViewItemType.INFORMATION_BUTTON`
- for elements with the specified `generator` property

The source code of the contact edit page replacing schema when adding the pop-up hint using method 2:

```
define("ContactPageV2", [],
function () {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Contact",
        //Pop-up hint visualization setup.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding pop-up hint to the button.
            {
                // Modification of the existing element.
                "operation": "merge",
                "parentName": "LeftContainer",
                "propertyName": "items",
                "name": "SaveButton",
                "values": {
                    // Pop-up hint array for a button.
                    "tips": []
                }
            },
            // Simple hint configuration object.
            {
                // Adding a new element.
                "operation": "insert",
                "parentName": "SaveButton",
                "propertyName": "tips",
                "name": "CustomShowedTip",
                "values": {
                    // Pop-up hint text.
                    "content": {"bindTo": "Resources.Strings.SaveButtonHint"}
                    // You can setup additional parameters of pop-up hint
                    // display and operation.
                }
            },
        ],
    }/**SCHEMA_DIFF*/
},
});
```

After you save the schema, a pop-up hint will appear next to the [Save] button on the contact edit page (Fig. 3).

Fig. 3. Case result demonstration



Example 2

Case description

Add a pop-up hint to the [Type] field of contact edit page.

Case implementation algorithm

1. Create a replacing contact page

Create a replacing client module and specify [Display schema – Contact card] (*ContactPageV2*) as parent object (Fig. 1).

2. Adding a localized string with the pop-up hint text

Add a string with the pop-up hint text to the collection of localizable strings of the edit page replacing schema. Properties of the created string (Fig. 4):

- [Name] – “TypeTipContent”
- [Value] – ‘Choose the type of contact from the list’

Fig. 4. Properties of the custom localizable string

The screenshot shows the 'Properties' dialog for a localizable string. The 'General' section is expanded, showing:

Name	AccountTipContent
Title	AccountTipContent
Value	Enter or choose the name of the account from the list

The 'Value' field is highlighted with a red border.

3. Adding a field configuration object to the *diffarray*

Add the *tip* property containing the *content* property to the *values* field property. The *content* property value will be

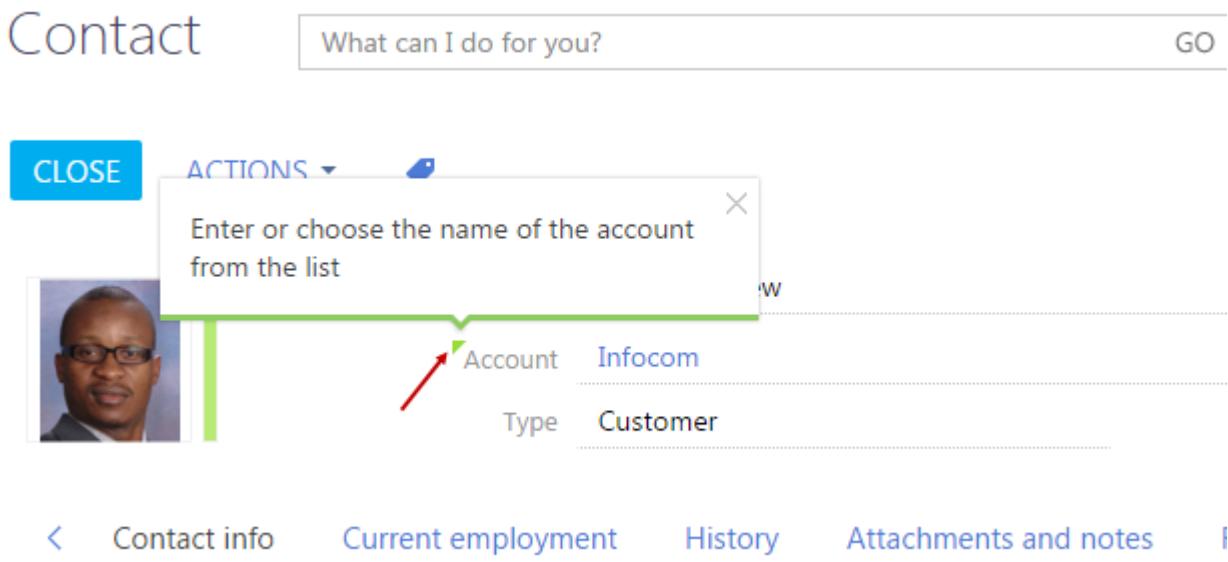
the pop-up hint text.

Below is the source code of the page replacing schema.

```
define("ContactPageV2", [],
function () {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Contact",
        //Pop-up hint visualization setup.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding pop-up hint to the field.
            {
                // Modification of the existing element.
                "operation": "merge",
                "name": "Type",
                "parentName": "ContactGeneralInfoBlock",
                "propertyName": "items",
                "values": {
                    // Field property responsible for displaying the pop-up hint.
                    "tip": {
                        // Pop-up hint text.
                        "content": { "bindTo": "Resources.Strings.TypeTipContent" },
                        // Pop-up hint display mode.
                        // WIDE is the default mode - thickness of a green band,
                        // displayed in the pop-up hint.
                        "displayMode": Terrasoft.controls.TipEnums.displayMode.WIDE
                    }
                }
            }
        ]/**SCHEMA_DIFF*/
    };
});
```

After you save the schema, a pop-up hint will appear in the [Type] field on the contact edit page (Fig. 5).

Fig. 5. Case result demonstration



Example 3

Case description

Add an information button to the [Full name] contact edit page.

Case implementation algorithm

1. Create a replacing contact page

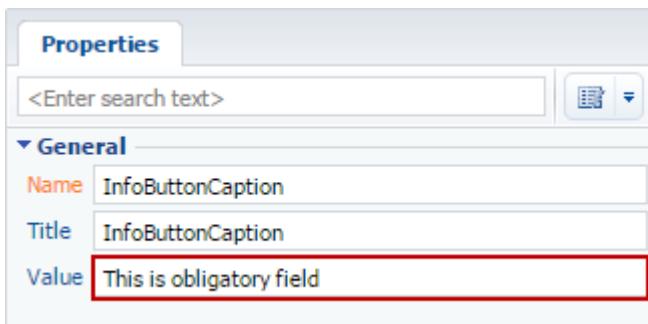
Create a replacing client module and specify [Display schema – Contact card] (*ContactPageV2*) as parent object (Fig. 1).

2. Adding a localized string with the pop-up hint text

Add a string with the pop-up hint text to the collection of localizable strings of the edit page replacing schema. Properties of the created string (Fig. 6):

- [Name] – “InfoButtonCaption”
- [Value] – “This is obligatory field”

Fig. 6. Properties of the custom localizable string



3. Adding a button configuration object to the diffarray

Add a new element with the *Terrasoft.ViewItemType.INFORMATION_BUTTON* type and the *content* property to the *diff* array. The *content* property value will be the pop-up hint text.

The source code of the edit page replacing schema:

```
define("ContactPageV2", [],
function () {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "Contact",
        //Pop-up hint visualization setup.
        diff: /**SCHEMA_DIFF*/[
            // Metadata for adding pop-up hint to the button.
            {
                // Modification of the existing element.
                "operation": "merge",
                "parentName": "ProfileContainer",
                "propertyName": "items",
                "name": "AccountName",
                "values": {
                    "layout": { "column": 0, "row": 1, "colSpan": 22, "rowSpan": 1 }
                }
            },
            {
                // Adding a new element.
                "operation": "insert",
                "parentName": "ProfileContainer",
                "propertyName": "items",
                "name": "SimpleInfoButton",
                "values": {

```

```

        "layout": { "column": 22, "row": 1, "colSpan": 1, "rowSpan": 1 },
        "itemType": Terrasoft.ViewItemType.INFORMATION_BUTTON,
        "content": { "bindTo": "Resources.Strings.InfoButtonCaption" }
    }
}
] /*SCHEMA_DIFF*/
;
}) ;
})
;
```

After you save the schema, the pop-up hint will appear in the [Account] field on the contact edit page (Fig. 7).

Fig. 7. Case result demonstration

The screenshot shows a contact edit form for Andrew Barber. The account field is populated with 'Infocom'. A green info icon (with a white 'i') is positioned next to the account field. A red arrow points from the text 'Fig. 7. Case result demonstration' to this icon.

	100%	Full name *	Barber Andrew	
		Account	Infocom	
		Type	Customer	
		Owner	John Best	

This screenshot is similar to Fig. 7, but the pop-up hint is more detailed. It includes a close button, a link to 'Read more', and icons for email and print. The text in the hint says 'This is obligatory field'.

	100%	Full name *	Barber Andrew	
		Account	Infocom	
		Type	Customer	
		Owner	John Best	

Example 4. Adding a link to web resource to the pop-up hint

You can add links to web resources or context help to pop-up hints. Add an html code of the link directly to the localizable string of the pop-up hint text (Fig. 8).

Fig. 8. Example of defining the pop-up hint with a link

The screenshot shows the 'Properties' panel for an item named 'AccountTipContent'. The 'Value' field contains the following text:

```

<a href="#" data-context-help-id="1023">Read more</a>

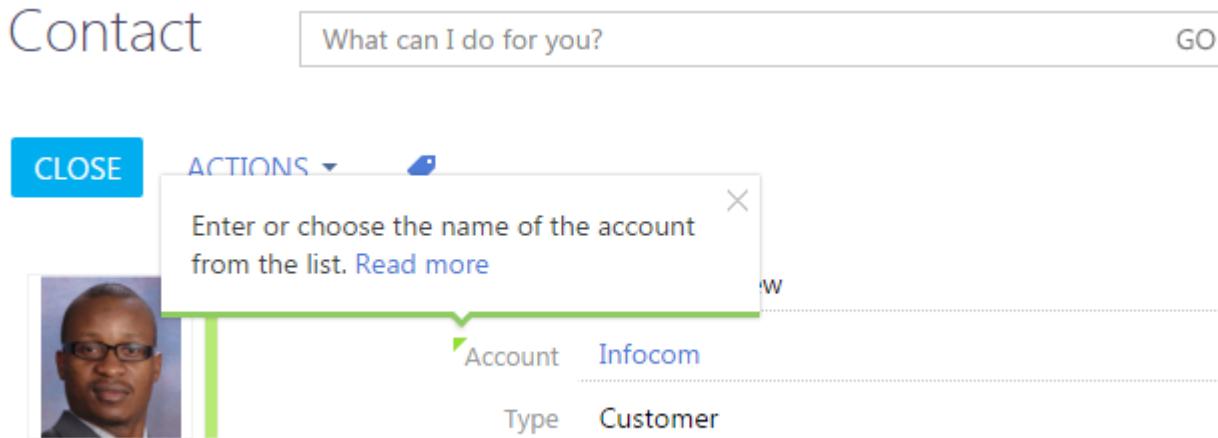
```

Example of adding a direct link to web resource:

```
<a href="https://academy.creatio.com/" target="_blank">Learn more</a>
```

As a result, the pop-up hint will look as shown in figure 9.

Fig. 9. Example of displaying the pop-up hint with a link



How to modify sales pipeline calculations

Beginner Easy Medium **Advanced**

You can modify the way values are calculated for the sales pipeline dashboard element in the [Opportunities section]. To do this, you need to create a new module for calculations and replace the sales pipeline display client schema.

To modify the sales pipeline calculations:

1. Create a new class inherited from *FunnelBaseDataProvider* and specify the calculation logic.
2. Create a replacing *FunnelChartSchema* client schema and use the new calculation class in it.

Example of modifying the calculations displayed in the "Number of opportunities" view of the sales pipeline

Case description

Modify the sales pipeline calculation algorithms by replacing the number of opportunities with the number of products added to opportunities.

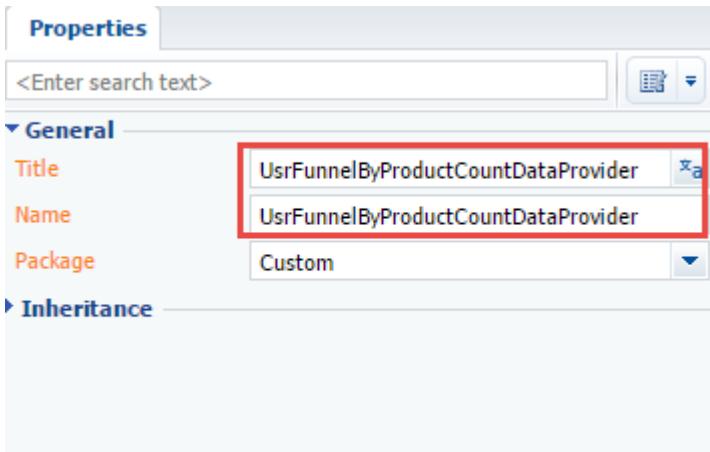
Case implementation algorithm

1. Create a new module in the custom package

Create a new calculation provider client module in the custom package. *Calculation provider* is a class responsible for selecting, filtering and processing data for sales pipeline chart.

Specify a name and caption for the new module, for example, *UsrFunnelByProductCountDataProvider* (Fig. 1).

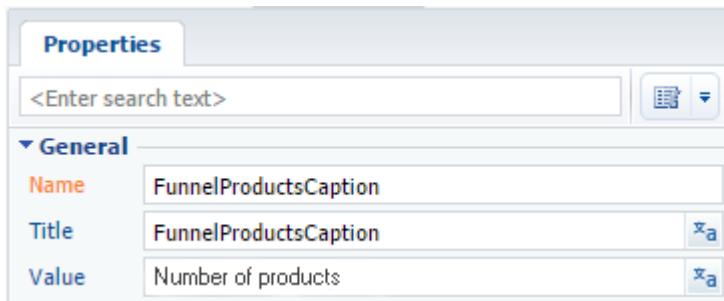
Fig. 1. Calculation provider module properties



2. Add localizable strings

Add a string with the *Number of products* value to the collection of localizable strings of the created module. To do this, right-click the [LocalizableStrings] structure node and select [Add] from the context menu. Set the properties for the new string as shown on Fig. 2.

Fig. 2. Localizable string properties



Add *CntOpportunity* localizable string with the *Number of opportunities* value in the similar way.

3. Add implementation to the provider module

To modify sales pipeline calculations, override the following methods:

- *addQueryColumns* column generation method for data selection
- methods for selection data processing.

To process one record from the selection, define the *getSeriesDataConfigByItem* method. To process the whole collection, define the *prepareFunnelResponseCollection* method. To filter the records, define the *applyFunnelPeriodFilters* method.

Below is the source code of the new calculation provider module for the sales pipeline.

```
define("UsrFunnelByProductCountDataProvider", ["ext-base", "terrasoft",
"UsrFunnelByProductCountDataProviderResources",
    "FunnelBaseDataProvider"],
function(Ext, Terrasoft, resources) {
    // Defining a new calculation provider.
    Ext.define("Terrasoft.configuration.UsrFunnelByProductCountDataProvider", {
        // Inheriting from the basic provider.
        extend: "Terrasoft.FunnelBaseDataProvider",
        // New provider short name
        alternateClassName: "Terrasoft.UsrFunnelByProductCountDataProvider",
        // Collection processing method
        prepareFunnelResponseCollection: function(collection) {
            this.callParent(arguments);
        },
        // ...
    });
});
```

```

// Extending the FunnelBaseDataProvider base model method.
// Sets the column number of products for data sampling
addQueryColumns: function(entitySchemaQuery) {
    // Parent method calling
    this.callParent(arguments);
    // Adds the number of products column to the sample
    entitySchemaQuery.addAggregationSchemaColumn([
        [OpportunityProductInterest:Opportunity].Quantity,
        Terrasoft.AggregationType.SUM, "ProductsAmount");
    },
    // Extending the FunnelBaseDataProvider base class method.
    // Sets sample filtration
    applyFunnelPeriodFilters: function(filterGroup) {
        // Parent method calling
        this.callParent(arguments);
        // Creates a filter group.
        var endStageFilterGroup = Terrasoft.createFilterGroup();
        // Sets the group operator type.
        endStageFilterGroup.logicalOperation =
Terrasoft.LogicalOperatorType.OR;
        // Sets the filter that shows whether the sale stage is over yet.
        endStageFilterGroup.addItem(
Terrasoft.createColumnIsNullFilter(this.getDetailColumnPath("DueDate")));
        // Sets the filter that shows whether the sale stage is final.
        endStageFilterGroup.addItem(
Terrasoft.createColumnFilterWithParameter(Terrasoft.ComparisonType.EQUAL,
    this.getDetailColumnPath("Stage.End"), true,
Terrasoft.DataValueType.BOOLEAN));
        filterGroup.addItem(endStageFilterGroup);
    },
    // Extending the FunnelBaseDataProvider base model method.
    // Processes data for the stages in the pipeline.
    getSeriesDataConfigByItem: function(responseItem) {
        // Object that stores localizable strings.
        var lczi = resources.localizableStrings;
        // Receives a stage data object from the parent method.
        var config = this.callParent(arguments);
        // Receives data about the number of products in an opportunity from
the sample result.
        var products = responseItem.get("ProductsAmount");
        products = Ext.isNumber(products) ? products : 0;
        // Formats the strings.
        var name = Ext.String.format("{0}<br/>{1}: {2}<br/>{3}: {4}",
            config.menuHeaderValue, lczi.CntOpportunity, config.y,
lczi.FunnelProductsCaption, products);
        var displayValue = Ext.String.format("<br/>{0}: {1}",
lczi.FunnelProductsCaption, products);
        // Installs new data in the data object and returns it.
        return Ext.apply(config, {
            name: name,
            displayValue: displayValue
        });
    }
});
});

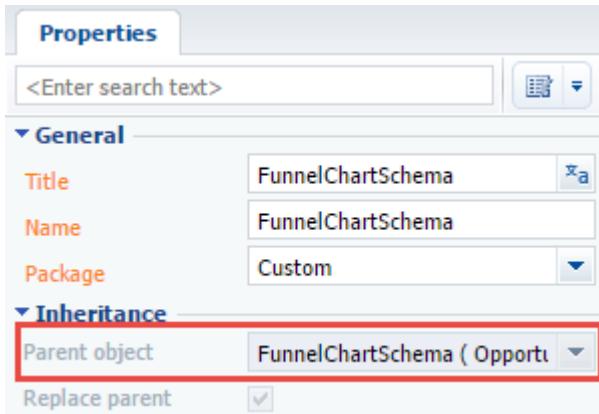
```

4. Create a sales pipeline replacing schema

To use the new provider module in the calculations, override the sales pipeline calculation provider generator method.

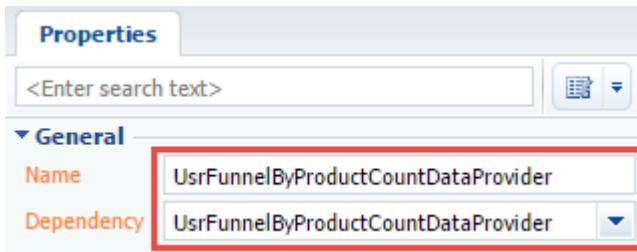
To do this, create a replacing client module and specify *FunnelChartSchema* as a parent (Fig. 3).

Fig. 3. Properties of the replacing module



Add the new calculation module to dependencies (the *Dependencies* section), by specifying its name in the [Dependency] field and the *UsrFunnelByProductCountDataProvider* value in the [Name] field (Fig. 4).

Fig. 4. Sales pipeline schema dependency properties



5. Specify the new calculation provider in the sales pipeline replacing schema

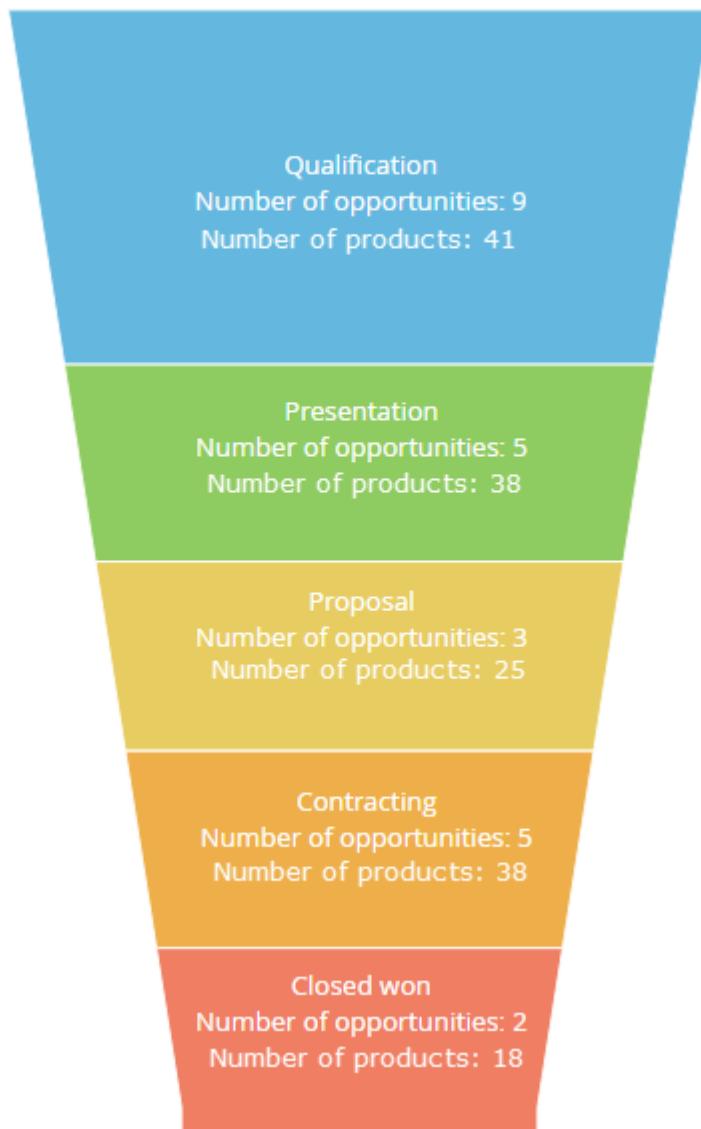
To do this, override the *getProvidersCollectionConfig* method in the replacing schema that gets the configuration object with the collection of providers.

```
define("FunnelChartSchema", ["UsrFunnelByProductCountDataProvider"],
  function() {
    return {
      entitySchemaName: "Opportunity",
      methods: {
        getProvidersCollectionConfig: function() {
          // Calls parent method.
          // Gets array of providers.
          var config = this.callParent();
          // Searches data provider in the measurement by the number of
opportunities.
          var byCount = Terrasoft.findItem(config, {tag:
"byNumberConversion"});
          // Replaces with new class.
          byCount.item.className =
"Terrasoft.UsrFunnelByProductCountDataProvider";
          return config;
        }
      }
    };
  });
});
```

After saving the schema, the new calculation module will be used in the sales pipeline and the sales pipeline itself will display the total number of products by stages (Fig. 5).

Fig. 5. Sales pipeline displaying the number of products added to opportunities

NUMBER OF OPPORTUNITIES STAGE CONVERSION RATE PIPELINE CONVERSION



How to enable additional filtering in a sales pipeline

Beginner

Easy

Medium

Advanced

Introduction

In Creatio, you can enable additional filtering for calculations in sales pipeline charts.

To do it this:

1. Create a new class inherited from the calculation provider and implement the necessary filtering logic.
2. Create a replacing FunnelChartSchema client schema and use the new calculation class in it.

Case description

Add filtering to sales pipeline calculations displayed in the “Number of opportunities” view for selecting the opportunities whose [Customer] field is populated with an account.

Source code

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

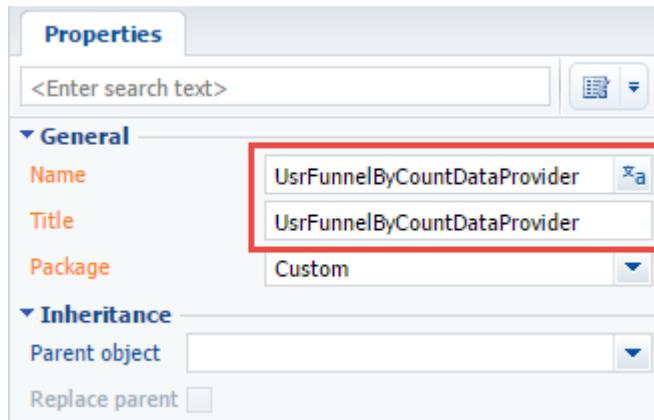
Case implementation algorithm

1. Creating a new module in the custom package

Create a new calculation provider client module in the custom package. *Calculation provider* is a class responsible for selecting, filtering and processing data for sales pipeline chart.

Specify a name and caption for the new module, for example, *UsrFunnelByCountDataProvider* (fig. 1).

Fig. 1. Calculation provider module properties



2. Defining the new provider class and specifying the filtering logic

Inherit the created class from the *FunnelByCountDataProvider* class and override the *getFunnelFixedFilters* method.

The module source code:

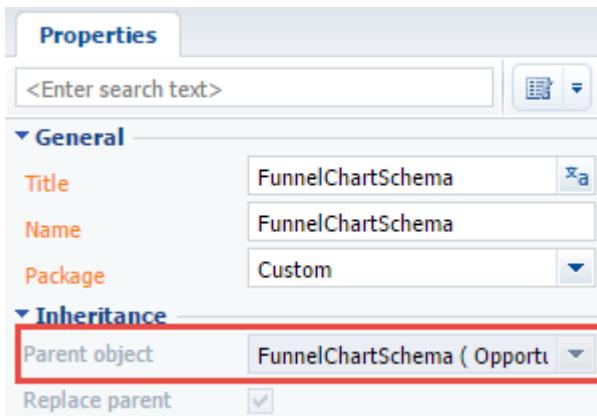
```
define("UsrFunnelByCountDataProvider", ["ext-base",
    "terrasoft", "UsrFunnelByCountDataProviderResources",
    "FunnelByCountDataProvider"],
    function(Ext, Terrasoft, resources) {
        // Defining the new calculation provider.
        Ext.define("Terrasoft.configuration.UsrFunnelByCountDataProvider", {
            // Inheritance from the provider "by number".
            extend: "Terrasoft.FunnelByCountDataProvider",
            // Contracted name of the new provider.
            alternateClassName: "Terrasoft.UsrFunnelByCountDataProvider",
            // Extending the FunnelByCountDataProvider base module method.
            // Returns filter for selection.
            getFunnelFixedFilters: function() {
                // Calling the parent method.
                var esqFiltersGroup = this.callParent(arguments);
                // Adds filter specifying that the customer of an opportunity is an
                account.
                esqFiltersGroup.addItem(
                    Terrasoft.createColumnIsNotNullFilter("Account"));
                return esqFiltersGroup;
            }
        });
    });
});
```

Save the module.

3. Implementing the pipeline chart module in custom package

To use the new provider module in calculations, create a replacing client module and specify *FunnelChartSchema* from the *Opportunity* package as a parent schema (fig. 2).

Fig. 2. Properties of the replacing module



4. Specify the new calculation provider in the sales pipeline replacing schema

Override the provider generator method of sales pipeline calculation in the replacing schema and specify the new provider class for calculations.

The replacing schema source code is as follows:

```
define("FunnelChartSchema", ["UsrFunnelByCountDataProvider"], function() {
    return {
        entitySchemaName: "Opportunity",
        methods: {
            getProvidersCollectionConfig: function() {
                // Calls parent method returning the provider array.
                var config = this.callParent();
                // Searches for data provider for displaying in the "Number of opportunities" view.
                var byCount = Terrasoft.findItem(config, {tag: "byNumberConversion"});
                // Changes for a new class.
                byCount.item.className = "Terrasoft.UsrFunnelByCountDataProvider";
                return config;
            }
        };
    });
});
```

After you save the schema, the new calculation module will be used in the sales pipeline. It will display the opportunities whose [Customer] field is populated with an account.

Adding a custom dashboard widget

Beginner

Easy

Medium

Advanced

Introduction

Dashboard widgets (analytic elements) are used for data analysis of sections. Go to the “Dashboards” view of the required section to work with its analytics. Use the [Dashboards] section to work with the entirety of Creatio section data analytics.

To learn more about Creatio dashboard widgets, please refer to the “**Section analytics**” article.

You can create custom dashboard widgets in Creatio.

Creating a custom widget

To create a custom widget you need to:

1. Create new or select the existing module. More information about dashboard widget modules can be found in the “**Dashboard widgets**” article.
2. Custom module must be an inheritor of the *BaseNestedModule* module or one of its inheritors: *ChartModule*, *IndicatorModule*, *GaugeModule*, etc. Add the source code that implements the necessary functionality to the created module.
3. Specify the module dependency in the [Dependencies] block of the module properties. Add messages that are used.
4. Set the widget parameters in the [Module parameters] field when adding widgets on the dashboards panel. More information about parameters can be found in the “**Dashboard widgets**” article.

Case description

Create custom widget that shows currency exchange rate.

Source code

Use this [link](#) to download the case implementation package.

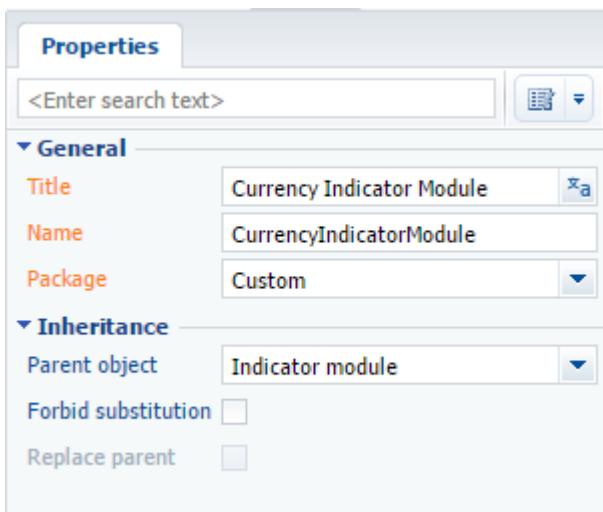
Case implementation algorithm

1. Create a currency indicator module.

Go to the [Configuration] section in the system designer and on the [Schemas] tab, select [Add] -> [Standard] -> [Module] command. For the created module specify (Fig. 1):

- [Name] – “UsrCurrencyIndicatorModule”.
- [Title] – “Currency Indicator Module”.

Fig. 1. Currency indicator module properties



2. Add the source code

The module source code:

```
define("UsrCurrencyIndicatorModule", ["UsrCurrencyIndicatorModuleResources",
"IndicatorModule"], function() {

    // Class that generates the configuration of the currency indicator module view..
    Ext.define("Terrasoft.configuration.CurrencyIndicatorViewConfig", {
        extend: "Terrasoft.BaseModel",
        alternateClassName: "Terrasoft.CurrencyIndicatorViewConfig",
        // Generates the configuration of the currency indicator module view.
    });
});
```

```
generate: function(config) {
    var style = config.style || "";
    var fontStyle = config.fontStyle || "";
    var wrapClassName = Ext.String.format("{0}", style);
    var id = Terrasoft.Component.generateId();
    // The returned configuration view object.
    var result = {
        "name": id,
        "itemType": Terrasoft.ViewItemType.CONTAINER,
        "classes": {wrapClassName: [wrapClassName, "indicator-module-wrapper"]},
        "styles": {
            "display": "table",
            "width": "100%",
            "height": "100%"
        },
        "items": [
            {
                "name": id + "-wrap",
                "itemType": Terrasoft.ViewItemType.CONTAINER,
                "styles": {
                    "display": "table-cell",
                    "vertical-align": "middle"
                },
                "classes": {wrapClassName: ["indicator-wrap"]},
                "items": [
                    // Display the name of the currency.
                    {
                        "name": "indicator-caption" + id,
                        "itemType": Terrasoft.ViewItemType.LABEL,
                        "caption": {"bindTo": "CurrencyName"},
                        "classes": {"labelClass": ["indicator-caption"]}
                    },
                    // Display the currency exchange rate.
                    {
                        "name": "indicator-value" + id,
                        "itemType": Terrasoft.ViewItemType.LABEL,
                        "caption": {
                            "bindTo": "CurrencyValue"
                        },
                        "classes": {"labelClass": ["indicator-value " +
fontStyle]}
                    }
                ]
            }
        ]
    };
    return result;
});
};

// Class of the view model of the currency indicator module.
Ext.define("Terrasoft.configuration.CurrencyIndicatorViewModel", {
    extend: "Terrasoft.BaseModel",
    alternateClassName: "Terrasoft.CurrencyIndicatorViewModel",
    Ext: null,
    Terrasoft: null,
    sandbox: null,
    columns: {
        // Currency name.
        CurrencyName: {
            type: Terrasoft.core.enums.ViewModelSchemaItem.ATTRIBUTE,
```

```
        dataType: Terrasoft.DataValueType.TEXT,
        value: null
    },
    // Currency value.
    CurrencyValue: {
        type: Terrasoft.core.enums.ViewModelSchemaItem.ATTRIBUTE,
        dataType: Terrasoft.DataValueType.FLOAT,
        value: null
    }
},
onRender: Ext.emptyFn,
// Returns the currency value, depending on the name. This method is given as
an example.
// For each specific task, you should select an individual method to obtain
data,
// for example REST API, database query, etc.
getCurrencyValue: function(currencyName, callback, scope) {
    var result = 0;
    if (currencyName === "USD") {
        result = 26;
    }
    if (currencyName === "EUR") {
        result = 32.3;
    }
    if (currencyName === "RUB") {
        result = 0.45;
    }
    callback.call(scope || this, result);
},
// Gets the data and displays them on the widget.
prepareIndicator: function(callback, scope) {
    this.getCurrencyValue(this.get("CurrencyName"), function(currencyValue) {
        this.set("CurrencyValue", currencyValue);
        callback.call(scope);
    }, this);
},
// Initializes the widget.
init: function(callback, scope) {
    this.prepareIndicator(callback, scope);
}
});

// Widget module class.
Ext.define("Terrasoft.configuration.CurrencyIndicatorModule", {
    extend: "Terrasoft.IndicatorModule",
    alternateClassName: "Terrasoft.CurrencyIndicatorModule",
    // The name of the wdgte view model class.
    viewModelClassName: "Terrasoft.CurrencyIndicatorViewModel",
    // The name of the view configuration generating class.
    viewConfigClassName: "Terrasoft.CurrencyIndicatorViewConfig",
    // Subscribing to messages from third-party modules.
    subscribeMessages: function() {
        this.sandbox.subscribe("GenerateIndicator", this.onGenerateIndicator,
this, [this.sandbox.id]);
    }
});

return Terrasoft.CurrencyIndicatorModule;
});
```

3. Add a style to the LESS tab

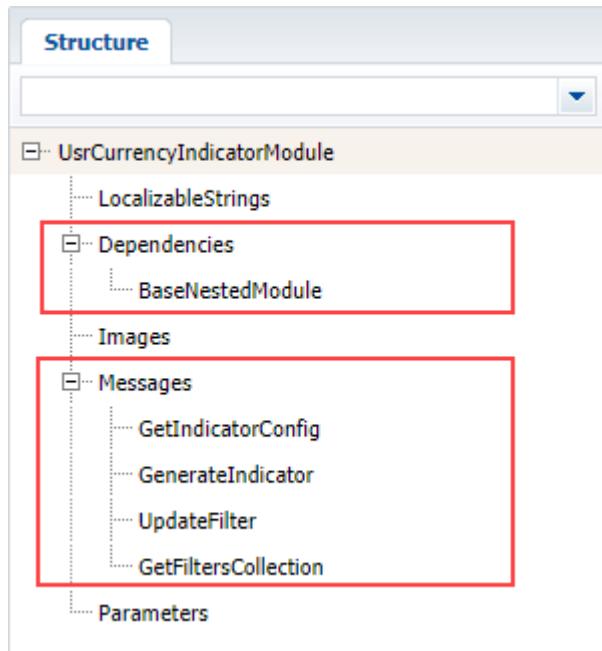
To display the widget text at the center, add the following style to the LESS tab of the module:

```
.indicator-module-wrapper {  
    text-align: center;  
}
```

3. Add the dependencies and messages

The dependencies and messages of parent module should automatically display in the created module (Fig. 2).

Fig. 2. Dependencies and messages of created module



If it doesn't happen, add them manually:

- Add a parent module to the [Dependencies] block
- Add the *GetIndicatorConfig* message to the [Messages] block. Sett the “Publish” direction for the message and the *GenerateIndicator* as address message with the “Follow” direction.

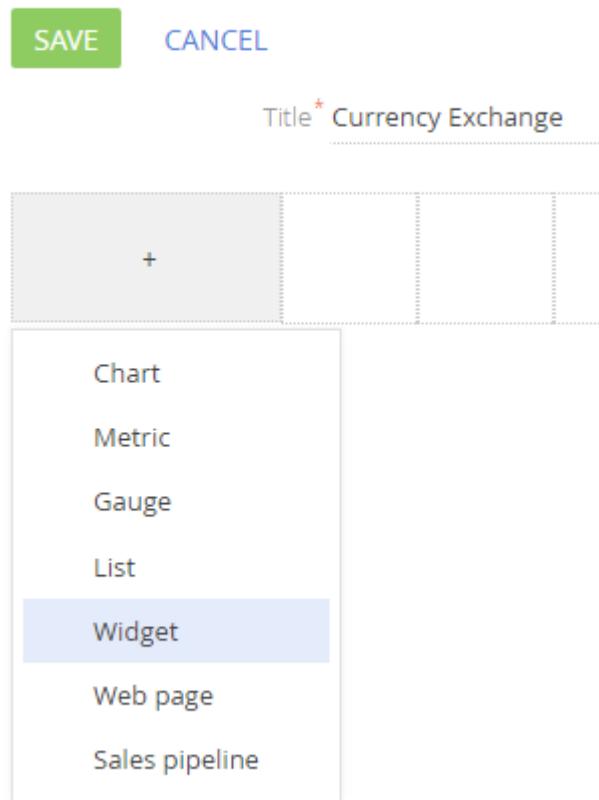
Save the new module.

5. Add the widget to the dashboard panel and set its parameters

To display the widget, add it to the dashboard panel (Fig. 3).

Fig. 3. Adding the widget to the dashboard panel

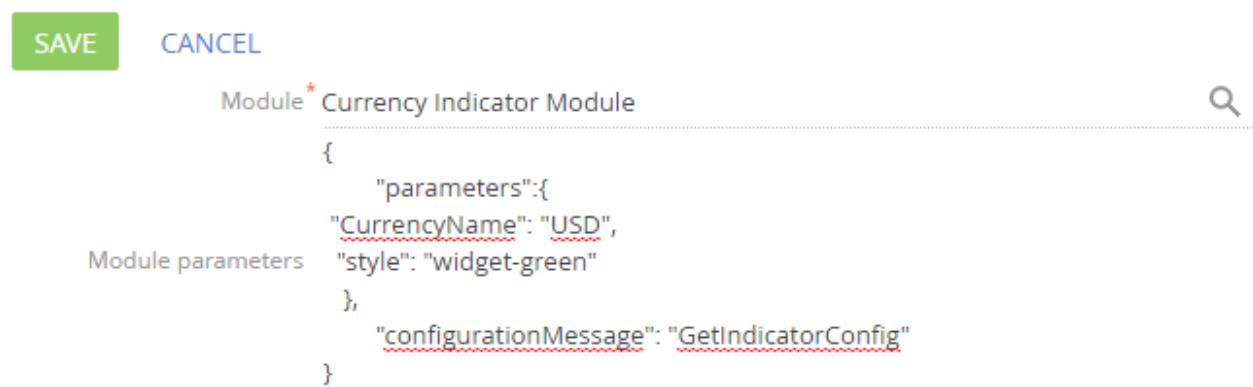
New dashboard panel



In addition, you need to set the parameters of the module bound to the widget (Fig. 4).

Fig. 4. Configuration of the added widget module

Module setting



To bind the module to the added widget, add the “Currency Indicator Module” value in the [Module] field and add the configuration JSON object with the required parameters to the [Module parameters] field.

```
{  
    "parameters": {  
        "CurrencyName": "USD",  
        "style": "widget-blue"  
    },  
    "configurationMessage": "GetIndicatorConfig"  
}
```

```
    "configurationMessage": "GetIndicatorConfig"
}
```

A “*CurrencyName*” parameter sets the currency for which the exchange rate is displayed. A “*style*” parameter sets the widget style and “*configurationMessage*” parameter sets the message name that will be used to transfer the configuration object.

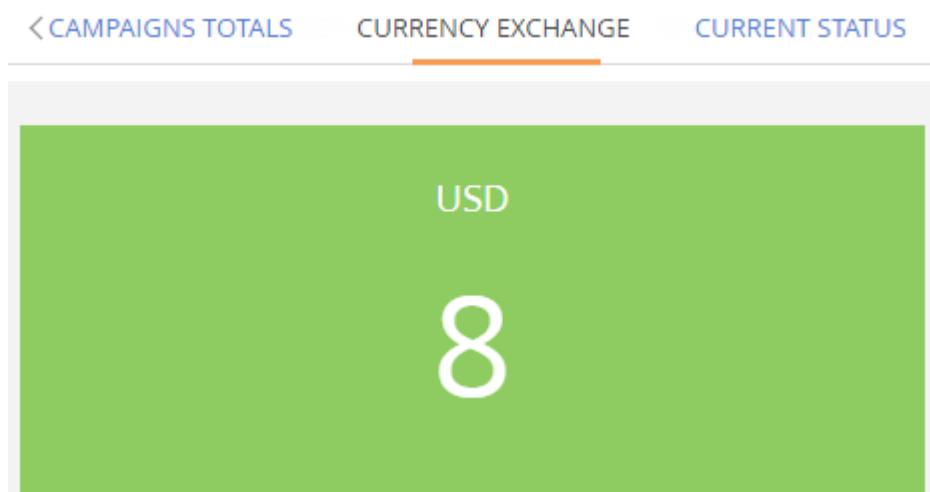
You can set up any of Creatio system colors in the *style* parameter as widget color (Fig. 5).

Fig. 5. Style types of the widget

- █ Green
- █ Mustard
- █ Orange
- █ Coral
- █ Violet
- █ Blue
- █ Light blue
- █ Dark turquoise
- █ Turquoise

After saving the created widget and refreshing the page, the custom widget will be displayed on the dashboards panel (Fig. 6).

Fig. 6. Currency exchange rate widget



Using the Terrasoft.AlignableContainer custom control

Beginner Easy Medium **Advanced**

Introduction

Starting with version 7.8 Creatio has a new custom control – the *Terrasoft.AlignableContainer*. This control is inherited from the *Terrasoft.Container* control and contains properties associated with a fixed container positioning relative to another controls.

Displaying the *Terrasoft.AlignableContainer* depends of the control used to position the container. In the absence of a control, the container is positioned in the center of the screen. The default order of container positioning is defined by the following sequence:

1. The container displays under the control at first.
2. If there is no space under the control, the container displays above it.
3. If placing either below or above is impossible, the container displays on the right.
4. If placing on the right is impossible, the container displays to the left of the control.

For the *Terrasoft.AlignableContainer* control you can also specify the background on which the container will be placed. These features of the *Terrasoft.AlignableContainer* control are applied in Creatio for positioning the mini page of the object.

Case description

When you click on a photo in the [Contacts] section, display an enlarged image with a background in the center of the screen. When hovering over a photo, display a larger version of the contact image relative to the photo container.

Source code

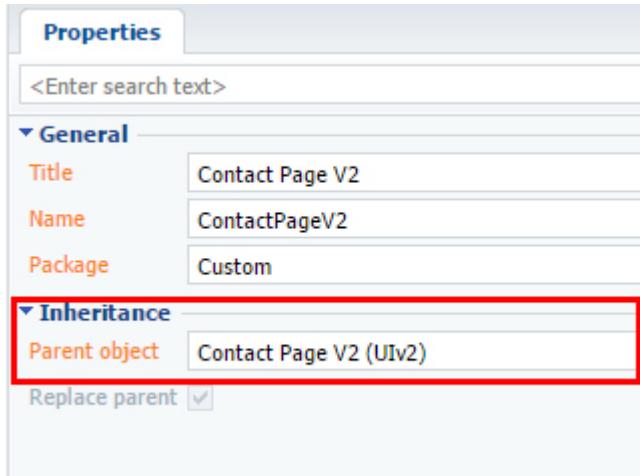
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Create a replacing client module

Create a replacing client module and specify [Display schema – Contact card] (*ContactPageV2*) as parent object (Fig. 1). Creating a replacing page is covered in the “**Creating a custom client module schema**” article.

Fig. 1. Properties of the *ContactPageV2* replacing schema



2. Add required methods to the page schema

- *openLargeSizeImage()* – handler for clicking on the contact's photo to display the photo in the center of the screen.
- *openMiddleSizeImage()* – handler for hover on the contact's photo to display the photo relative to the parent control.
- *setAlignToEl()* – sets control near which the container needs to be displayed.
- *subscribePhotoContainerEvents()* – creates a subscription to the mouse hover event.
- *onEntityInitialized()* – an overridden base class method that performs actions after loading an object entity.

- `close()` – hides containers with the photos.

3. Add the necessary configuration objects to the diff modification array.

To display image at the center of the screen add the `Terrasoft.AlignableContainer` custom control without parameters (do not specify a link on control near which you want to display the container as a parameter). Set the `true` value to the checkbox which is responsible for displaying the background for it to show up.

Pass parent control name to the `Terrasoft.AlignableContainer` container to to display a photo relative to the control.

Add a closing button to hide the image.

The source code of the schema of contact edit page:

```
// Defining a module and it's dependencies.
define("ContactPageV2", ["css!UsrContactPhotoContainerCSS"], function() {
    return {
        // Object schema name.
        entitySchemaName: "Contact",
        attributes:
            // The identifier of the element near which to display the container.
            "AlignToElementId": {
                // Element type.
                dataType: this.Terrasoft.DataValueType.TEXT,
                // Column type.
                type: this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                // Element value.
                value: "ContactPageV2AccountPhotoContainerContainer"
            },
            // The flag responsible for displaying the container when hovering over
            // the photo.
            "MiddleSizeContainerVisible": {
                dataType: this.Terrasoft.DataValueType.BOOLEAN,
                type: this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                value: false
            },
            // The flag responsible for displaying the container when clicking on a
            // photo.
            "LargeSizeContainerVisible": {
                dataType: this.Terrasoft.DataValueType.BOOLEAN,
                type: this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
                value: false
            }
        },
        methods: {
            // Performs actions after loading the entity entity.
            onEntityInitialized: function() {
                // Call the parent class method.
                this.callParent(arguments);
                // Set the element near which to display the container.
                this.setAlignToEl();
                // Creates a subscription to the mouse hover event.
                this.subscribePhotoContainerEvents();
            },
            // Set the element near which to display the container.
            setAlignToEl: function() {
                // Gets the ID of the element.
                var alignToElementId = this.get("AlignToElementId");
                // Gets the DOM element.
                var alignToEl = this.Ext.get(alignToElementId);
                // Writes the value of the DOM element to the AlignToEl parameter.
                this.set("AlignToEl", alignToEl);
            }
        }
    }
});
```

```
// Creates a subscription to the mouse hover event.
subscribePhotoContainerEvents: function() {
    // Get the DOM element of the container with the photo.
    var container = this.get("AlignToEl");
    // Performs a subscription to the mouse hover event.
    container.on("mouseover", this.openMiddleSizeImage, this);
},
// Displays the image after hovering on the container.
openMiddleSizeImage: function() {
    // Sets the container to visible with an average-sized image.
    this.set("MiddleSizeContainerVisible", true);
    // Hides a container that displays large photo.
    this.set("LargeSizeContainerVisible", false);
},
// A method that displays a container with a larger photo.
openLargeSizeImage: function() {
    // Sets the container to visible with a larger photo.
    this.set("LargeSizeContainerVisible", true);
    // Hides a container that displays an average-sized image.
    this.set("MiddleSizeContainerVisible", false);
},
// method that hides containers with images.
close: function() {
    // Hides a container that displays a large photo.
    this.set("LargeSizeContainerVisible", false);
    // Hides a container that displays an average-sized image.
    this.set("MiddleSizeContainerVisible", false);
}
},
diff: /**SCHEMA_DIFF*/ [
{
    // Connecting element properties.
    "operation": "merge",
    // Element name.
    "name": "Photo",
    // Parent element name.
    "parentName": "AccountPhotoContainer",
    // Property name.
    "propertyName": "items",
    // Element value.
    "values": {
        // Adding a method handler for a container click event.
        "onImageClick": {
            // Binding to the method handler of a container click event.
            "bindTo": "openLargeSizeImage"
        }
    }
},
{
    // Element inserting.
    "operation": "insert",
    // Element name.
    "name": "AlignablePhotoContainer",
    // Element value.
    "values": {
        // Container id.
        "id": "AlignablePhotoContainer",
        // Element type.
        "itemType": Terrasoft.ViewItemType.CONTAINER,
        // Element object classes.
        "className": "Terrasoft.AlignableContainer",
        // Element classes.
        "classNames": [
            "AlignableContainer"
        ]
    }
}
]
```

```
"wrapClass": ["photo-alignable-container", "middle-size-image-
container"],
    // Container visibility method handler.
    "visible": {"bindTo": "MiddleSizeContainerVisible"},
    // The element near which you want to display the container.
    "alignToEl": {"bindTo": "AlignToEl"},
    // Background display flag.
    "showOverlay": false,
    // Container elements.
    "items": []
}
},
{
    "operation": "insert",
    "name": "ClosePhotoButton",
    // Parent element name.
    "parentName": "AlignablePhotoContainer",
    // Property name.
    "propertyName": "items",
    // Element values.
    "values": {
        // Element type.
        "itemType": Terrasoft.ViewItemType.BUTTON,
        // Element classes of css style.
        "classes": {
            "imageClass": ["close-no-repeat-button"],
            "wrapperClass": ["close-button-wrapper"]
        },
        // The property of the background of the button as an image.
        "imageConfig": {
            "bindTo": "Resources.Images.CloseButtonImage"
        },
        // Method-handler for pressing the button to close the element.
        "click": {"bindTo": "close"}
    }
},
{
    // Inserting an element.
    "operation": "insert",
    // Element name.
    "name": "AccountResizedPhotoContainer",
    // Parent element name.
    "parentName": "AlignablePhotoContainer",
    // Property name.
    "propertyName": "items",
    // Element values.
    "values": {
        // Element type.
        "itemType": Terrasoft.ViewItemType.BUTTON,
        // Element object class.
        "className": "Terrasoft.ImageView",
        // Method of obtaining a link to an image.
        "imageSrc": {"bindTo": "getContactImage"}
    }
},
{
    // Inserting an element.
    "operation": "insert",
    // Element name.
    "name": "AlignableLargePhotoContainer",
    // Element values.
    "values": {
```

```
// Container id.
"id": "AlignableLargePhotoContainer",
// Element type.
"itemType": Terrasoft.ViewItemType.CONTAINER,
// Element object class.
"className": "Terrasoft.AlignableContainer",
// Element classes.
"wrapClass": ["photo-alignable-container", "large-size-image-
container"],
// Method handler of container visibility.
"visible": {"bindTo": "LargeSizeContainerVisible"},
// The element near which you want to display the container.
"alignToEl": null,
// A flag for displaying the background.
"showOverlay": {"bindTo": "LargeSizeContainerVisible"},
// Container elements.
"items": []
},
{
// Inserting an element.
"operation": "insert",
// Element name.
"name": "CloseLargePhotoButton",
// Parent element name.
"parentName": "AlignableLargePhotoContainer",
// Property name.
"propertyName": "items",
// Element values.
"values": {
    // Element type.
    "itemType": Terrasoft.ViewItemType.BUTTON,
    // Element classes.
    "classes": {
        "imageClass": ["close-no-repeat-button"],
        "wrapperClass": ["close-button-wrapper"]
    },
    // Forming an button with an image property.
    "imageConfig": {
        "bindTo": "Resources.Images.CloseButtonImage"
    },
    // Method-handler for pressing the button to close the element.
    "click": {"bindTo": "close"}
}
},
{
// Inserting an element.
"operation": "insert",
// Element name.
"name": "AccountLargeResizedPhotoContainer",
// Parent element name.
"parentName": "AlignableLargePhotoContainer",
// property name.
"propertyName": "items",
// Element values.
"values": {
    // Element type.
    "itemType": Terrasoft.ViewItemType.BUTTON,
    // Element object class.
    "className": "Terrasoft.ImageView",
    // Method of getting an image link.
    "imageSrc": {"bindTo": "getContactImage"}
```

```
        }
    }
] /**SCHEMA_DIFF*/
);
});
```

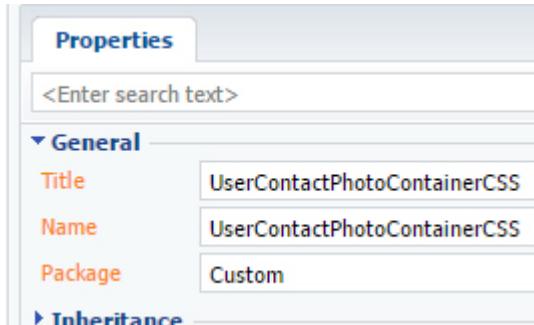
4. Define the CSS styles for displaying containers

In order to set the required dimensions of the displayed photo container, you need to define its CSS-style.

To do this, go to the [Configuration] section, and select [Add] > [Standard] > [Module] in the [Schemas] tab.

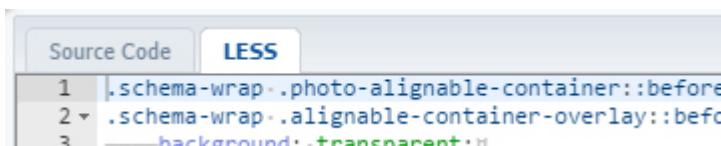
In module properties, set the title and header to "UserContactPhotoContainerCSS" (Fig. 2).

Fig. 2. Module properties



Module styles are defined on the LESS tab (Fig. 3).

Fig. 3. The LESS tab of the module



Add the following CSS selectors:

```
.schema-wrap .photo-alignable-container::before,
.schema-wrap .alignable-container-overlay::before {
    background: transparent;
}

.schema-wrap .photo-alignable-container.alignable-container {
    background: white;
}

.photo-alignable-container.middle-size-image-container {
    width: 250px;
    height: 275px;
}

.photo-alignable-container.large-size-image-container {
    width: 500px;
    height: 525px;
}

.photo-alignable-container .close-no-repeat-button {
    background-repeat: no-repeat;
}

.photo-alignable-container .close-button-wrapper:hover {
    background: transparent;
```

```
}
```

```
#ContactPageV2AccountLargeResizedPhotoContainerButton-image-view,  
#ContactPageV2AccountResizedPhotoContainerButton-image-view  
{  
    height: 90%;  
}
```

Save the schema, update the application page and clear the cache. A larger contact photo in the page center on a semi-transparent gray background will show up as the result (Fig.4). A larger contact photo will show up next to the main one as the result (Fig. 5).

Fig. 4. A larger contact phone in the screen center

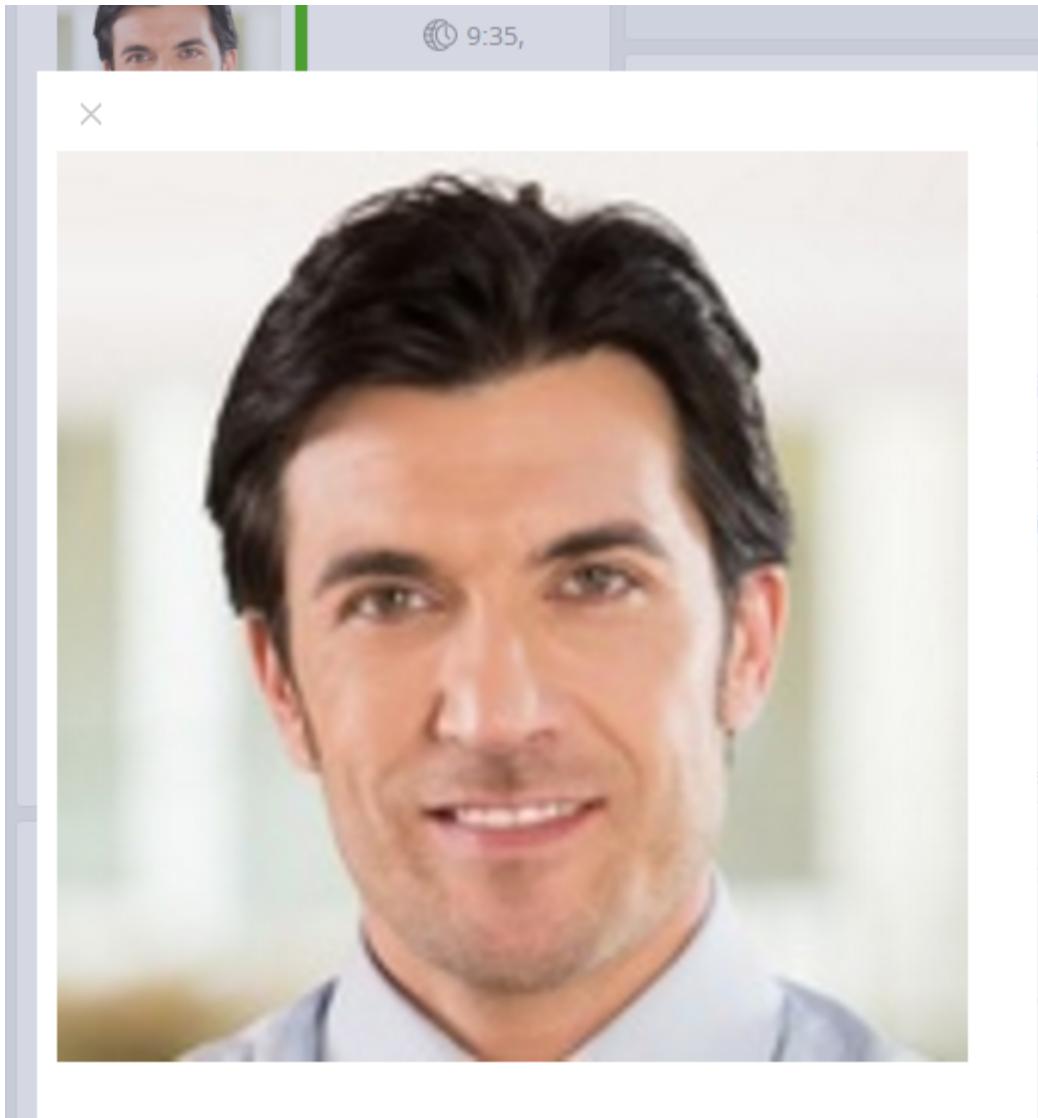
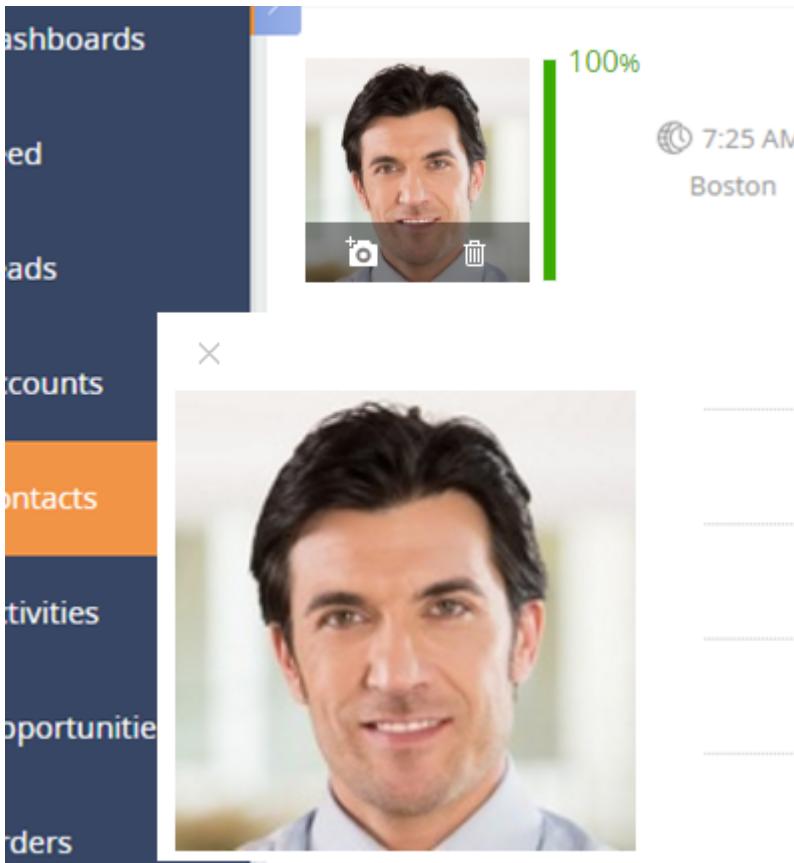


Fig. 5. A larger contact photo next to the main one



Adding a duplicate search rule

Beginner Easy Medium **Advanced**

Introduction

This description of adding the duplicates search rule is only available for Creatio version 7.13.2 and lower. You can find the description of the functionality for Creatio version 7.13.4 and up in the "[Finding and merging duplicates](#)" article.

Deduplication process uses the rules created as procedures and stored in Creatio. When a rule is being executed, it populates the *ContactDuplicateSearchResult* table with a list of duplicates. Search results are grouped by rules and displayed on the duplicate page.

To add a duplicate rule:

1. Add a column to the object schema (if needed). The column value will be used for the search of duplicates.
2. Add the stored search procedure to the application database.
3. Register the stored procedure as a new rule.

Case description

When launching the duplicate search process, the contacts with the same [Taxpayer ID] column values should be considered duplicates and should be displayed as search result.

Case implementation algorithm

1. Adding a field whose value will be used for the duplicate search

Since the [Taxpayer ID] field is not available on a standard contact edit page, add the field to the page (e.g., via a section wizard). For more information about adding fields to edit pages see the "[Adding a new field to the edit page](#)" article.

The new field properties:

- [Title] – “Taxpayer ID”
- [Name in DB] – “UsrInn”

2. Adding the stored search procedure to the application database

Add the stored duplicate search procedure by the [Taxpayer ID] field to the database. To do so, execute the following SQL script:

```
-- Verifying if the stored tsp_FindContactDuplicateByInn procedure is available.
IF NOT OBJECT_ID('[dbo].[tsp_FindContactDuplicateByInn]') IS NULL
BEGIN
    -- Deleting the stored procedure.
    DROP PROCEDURE [dbo].[tsp_FindContactDuplicateByInn];
END;
GO
-- Creating the stored procedure.
CREATE PROCEDURE [dbo].[tsp_FindContactDuplicateByInn] (
    -- This table parameter is only rendered if a new contact is stored.
    -- Contains the new contact data.
    -- If duplicate global search process is launched, the rendered parameter
contains no data.
    @parsedConfig CreatingObjectInfo READONLY,
    -- Unique identifier of user who launched the duplicate search.
    @sysAdminUnit UNIQUEIDENTIFIER,
    -- Identifier of the current rule from the [ContactDuplicateSearchResult] table.
    -- This identifier is created after a rule is registered in the system.
    @ruleId UNIQUEIDENTIFIER
)
AS
BEGIN
    -- Receiving the quantity of records from the accepted table for defining the
duplicate global search launch.
    DECLARE @parsedConfigRowsCount INT = (SELECT COUNT(*) FROM @parsedConfig);
    -- Creating temporary table with contact data for the search.
    CREATE TABLE #searchContact (
        [UsrInn] INT,
        [SortDate] DATETIME
    );
    -- In case of global search, the temporary table is populated with data.
    IF @parsedConfigRowsCount = 0
    BEGIN
        -- Adding data for duplicate search to the temporary table.
        INSERT INTO #searchContact ([UsrInn], [SortDate])
        -- Query for contact data selection.
        SELECT
            -- The Taxpayer ID columns of contact modification date are selected.
            [UsrInn],
            MAX([ModifiedOn])
        FROM [Contact]
        -- Grouping by fields is added to enable using quantity verification.
        GROUP BY [UsrInn]
        -- The table is populated only if more than one contact is available.
        HAVING COUNT(*) > 1;
    END;

    -- Populating the table of results.
    INSERT INTO [ContactDuplicateSearchResult] ([ContactId], [GroupId], [RuleId],
    [SysAdminUnitId])
    SELECT
        -- Contact duplicate identifier.
```

```

[vr].[Id],
-- Group numbering.
DENSE_RANK() OVER (ORDER BY [vr].[SortDate] DESC, [vr].[UsrInn]),
-- Rule identifier.
@ruleId RuleId,
-- Identifier of user who launched the duplicate search process.
@sysAdminUnit
FROM (
    -- Subquery populating the duplicates table.
    SELECT
        -- Contact identifier.
        [v].[Id],
        --Contact Taxpayer ID.
        [v].[UsrInn],
        -- Date of sorting.
        [r].[SortDate]
    -- Tables providing data.
    FROM [Contact] [v], #searchContact r
    -- The rule defining that contacts are duplicates.
    WHERE [v].[UsrInn] = [r].[UsrInn]
    -- Grouping of search results.
    GROUP BY [v].[UsrInn], [r].[SortDate], [v].[Id]
) [vr];
END;
GO

```

Sometimes you can come across the "Cannot resolve the collation conflict between "Cyrillic_General_CI_AS" and "Cyrillic_General_CI_AI" in the equal to operation" error. To fix it, specify the necessary COLLATE when creating the table column.

```
CREATE TABLE #searchContact ([Name] NVARCHAR(128) COLLATE Cyrillic_General_CI_AI,
[BirthDate] DATETIME, [SortDate] DATETIME );
```

3. Registering the stored procedure as a new rule

To register the stored procedure as a new duplicate search rule, add the corresponding record to the *DuplicatesRule* table. To do so, execute the following SQL script:

```

-- Variable that stores the UIId column value of the Contact schema.
DECLARE @ContactUIId UNIQUEIDENTIFIER;

-- Receives the UIId column value of the Contact schema.
Set @ContactUIId = (SELECT TOP 1 SysSchema.UIId FROM SysSchema
WHERE SysSchema.Name = 'Contact' AND SysSchema.ExtendParent = 0);

-- Adds a new rule to the system.
INSERT INTO DuplicatesRule ([IsActive], [ObjectId], [ProcedureName], [Name]) VALUES
(1, @ContactUIId, 'tsp_FindContactDuplicateByInn', 'ContactDuplicate. INN');

```

After you update the application page and clear the cache, you will have a new rule in the list of duplicate search rules (Fig. 1).

Fig. 1. Duplicate search rule by the [Taxpayer ID] field

Contact duplicates. Contact Email	Contact	Yes
Contact duplicates. Taxpayer ID	Contact	Yes
Contact duplicates. Contact name	Contact	Yes

Adding a rule for duplicates search when saving a record

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

Introduction

This guide for adding deduplication rules is intended for Creatio version 7.13.2 and below. For Creatio version 7.13.4 and up, deduplication and its features are covered in the "[Deduplication](#)" article. Bulk deduplication is covered in the "[Adding a duplicate search rule](#)" article.

Rules for duplicates search when saving a record have their specifics. When a record is saved, either [basic rules](#) or custom rules are applied (custom rules use the same fields as the basic rules).

To ensure that a custom deduplication rule is triggered not just during bulk deduplication but also when saving a record, take the following steps:

1. Execute the replacement for the *DuplicatesSearchUtilitiesV2* schema.
2. Create a *UsrSingleRequest* class.
3. Add the *UsrSingleRequestListener* class.
4. Create a *UsrDeduplicationProcessing* class.
5. Compile and restart the application.
6. Create a custom service.
7. Replace the *getDuplicatesServiceName* and *getFindDuplicatesServiceMethodName* methods.
8. Delete stored procedures from the database.
9. Execute the script for the deletion of the *CreatingObjectInfo* type.
10. Create a *tsp_FindDuplicate* stored procedure.
11. Execute the installation process for remote stored procedures.
12. Add the *tsp_FindAccountDuplicateByInn* stored procedure.

Case description

Implement executing a custom deduplication rule when saving a record for an **[Account]** object.

Case implementation algorithm

1. Replace the DuplicatesSearchUtilitiesV2 schema

In the custom schema, replace the *getDataForFindDuplicatesService()* method by adding *UsrINN* in the same way as "Name". The process for replacing the schema is covered in the "[Configuration architectural elements](#)" article.

The source code is available below.

```
getDataForFindDuplicatesService: function() {
    var communication = this.getCommunications();
    val email = this.get("Email");
    if (!this.Ext.isEmpty(email)) {
        communication.push({
```

```

        "Number": email,
        "CommunicationTypeId": ConfigurationConstants.CommunicationTypes.Email
    });
}
var data = {
    schemaName: this.entitySchemaName,
    request: {
        Id: this.get("Id"),
        Name: this.get("Name"),
        UsrINN: this.get("UsrINN"),
        AlternativeName: this.get("AlternativeName"),
        Communication: communication
    }
};
return data;
},

```

2. Create a *UsrSingleRequest* class

The *UsrSingleRequest* class must extend the *SingleRequest* class of the *SearchDuplicatesService* schema. Add the custom *UsrINN* property to the *UsrSingleRequest* class.

The source code is available below.

```

namespace Terrasoft.Configuration.SearchDuplicatesService {
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using System.Web;
    using Terrasoft.Common;
    using Terrasoft.Core;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Entities;
    using Terrasoft.Core.Scheduler;
    using System;
    using System.Data;
    using System.Collections.Generic;
    using System.Linq;
    using System.Runtime.Serialization;
    using System.Xml;
    using Quartz;
    using Quartz.Impl.Triggers;
    using Column = Terrasoft.Core.DB.Column;

    [DataContract]
    public class UsrSingleRequest: SingleRequest {
        [DataMember]
        public string UsrINN { get; set; }
    }
}

```

3. Add the *UsrSingleRequestListener* class

Add the *UsrSingleRequestListener* class, which will replace a call to *SingleRequest* with a call to *UsrSingleRequest*. The *UsrSingleRequestListener* class must implement the *IAppEventListener* interface. Add *ClassFactory.Bind()* to the *OnAppStart* method.

The source code is available below.

```

namespace Terrasoft.Configuration
{
    using Terrasoft.Core.Factories;
    using Terrasoft.Web.Common;

```

```

using Terrasoft.Configuration.SearchDuplicatesService;

#region Class: UsrSingleRequestListener

public class UsrSingleRequestListener: IAppEventListener
{
    #region Methods: Public

    public void OnAppStart(AppEventArgs context) {
        ClassFactory.Bind<SingleRequest, UsrSingleRequest>();
    }

    public void OnAppEnd(AppEventArgs context) {
    }

    public void OnSessionStart(AppEventArgs context) {
    }

    public void OnSessionStart(AppEventArgs context) {
    }

    #endregion
}

#endregion
}

```

4. Create a *UsrDeduplicationProcessing* class

Create a *UsrDeduplicationProcessing* class inheriting the *DeduplicationProcessing* schema. Take the following steps in the created class:

1. Add the *AddElementsToRow* and *GetPreparedXml* methods, of the *SearchDuplicatesService* schema and change their names to *UsrAddElementsToRow* and *UsrGetPreparedXml* accordingly.
 2. Add *UsrINN* to the *UsrAddElementsToRow* method.
 3. In the *UsrGetPreparedXml* method, replace the call to the *AddElementsToRow* method with a call to the *UsrAddElementsToRow* method.
- The source code is available below.

```

private XmlDocument UsrGetPreparedXml(UsrSingleRequest request) {
    XmlDocument xml = new XmlDocument();
    XmlElement elementRows = xml.CreateElement("rows");
    List<RequestCommunication> communicationsList = request.Communication && new
    List<RequestCommunication>();
    foreach (RequestCommunication communication in communicationsList) {
        XmlElement elementRow = xml.CreateElement("row");
        XmlElement elementCommunicationTypeId =
    xml.CreateElement("CommunicationTypeId");
        elementCommunicationTypeId.InnerText =
    communication.CommunicationTypeId.ToString();
        XmlElement elementNumber = xml.CreateElement("Number");
        elementNumber.InnerText = communication.Number;
        elementRow.AppendChild(elementCommunicationTypeId);
        elementRow.AppendChild(elementNumber);
        UsrAddElementsToRow(xml, elementRow, request);
        elementRows.AppendChild(elementRow);
    }
    if (communicationsList.Count < 1) {
        XmlElement elementRow = xml.CreateElement("row");
        UsrAddElementsToRow(xml, elementRow, request);
        elementRows.AppendChild(elementRow);
    }
}

```

```
        xml.AppendChild(elementRows);
        return xml;
    }

    private void UsrAddElementsToRow(XmlDocument xml, XmlElement elementRow,
UsrSingleRequest request) {
    XmlElement elementName = xml.CreateElement("Name");
    ele.InnerText = request.Name;
    elementRow.AppendChild(elementName);
    XmlElement elementUsrINN = xml.CreateElement("UsrINN");
    elementUsrINN.InnerText = request.UsrINN;
    elementRow.AppendChild(elementUsrINN);
    if (request.Id != Guid.Empty) {
        XmlElement elementId = xml.CreateElement("Id");
        elementId.InnerText = request.Id.ToString();
        elementRow.AppendChild(elementId);
    }
}
```

4. Override the *FindDuplicates* method.
5. Replace *GetPreparedXml* with *UsrGetPreparedXml*.
6. Cast the *data* parameter passed to the *UsrGetPreparedXml* to the *UsrSingleRequest* type.
7. In the *UsrAddElementsToRow* and *UsrGetPreparedXml* methods, change the type of the *request* parameter from *SingleRequest* to *UsrSingleRequest*.
The source code is available below.

```
public override List<Guid> FindDuplicates(string schemaName, SingleRequest data)
{
    XmlDocument xml = UsrGetPreparedXml((UsrSingleRequest)data);
    return FindDuplicates(schemaName, xml);
}
```

8. Add a parameterized constructor and call the basic implementation of the class constructor.

5. Compile and restart the application

After you complete steps 1 through 4, compile and restart the application.

6. Create a custom service

Create a custom service to replace *DeduplicationService* calls when saving accounts. Take the following steps in the created service:

1. Copy the *FindDuplicatesOnSave* method from the *DeduplicationService* schema.
2. Substitute *DeduplicationProcessing* with *UsrDeduplicationProcessing*.
The source code is available below.

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Data;
    using System.Linq;
    using System.Runtime.Serialization;
    using System.Xml;
    using Terrasoft.Common;
    using Terrasoft.Configuration.RightsService;
    using Terrasoft.Core;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Entities;
    using Terrasoft.Core.Factories;
```

```

using Terrasoft.Core.Scheduler;
using Terrasoft.Nui.ServiceModel.Extensions;
using Terrasoft.Configuration.SearchDuplicatesService;
using EntityCollection =
Terrasoft.Nui.ServiceModel.DataContract.EntityCollection;

public class UsrDeduplicationProcessing: DeduplicationProcessing
{
    public UsrDeduplicationProcessing(UserConnection userConnection):
base(userConnection)
    {

    }
}

```

7. Replace the `getDuplicatesServiceName` and `getFindDuplicatesServiceMethodName` methods

The `AccountPageV2` schema of the `Deduplication` package contains the `getDuplicatesServiceName` and `getFindDuplicatesServiceMethodName` methods. Replace the methods and pass to them the name of the custom service and the name of the `FindDuplicatesOnSave` method of your custom service.

8. Delete stored procedures from the database

Delete all stored deduplication procedures from the database (their names all start with `Find`).

9. Execute the script for the deletion of the `CreatingObjectInfo` type

Execute the script for the deletion of the `CreatingObjectInfo` type, which is created by `tsp_FindDuplicate` and is used in all stored deduplication procedures. To do so, execute the following SQL script:

```

IF TYPE_ID('[dbo].[CreatingObjectInfo]') IS NOT NULL
BEGIN
    drop type [CreatingObjectInfo];
END

```

10. Create a `tsp_FindDuplicate` stored procedure

Create a custom `tsp_FindDuplicate` stored procedure Take the following steps in the created procedure:

1. Create a `CreatingObjectInfo` type and add `UsrINN` to the new type. To do so, execute the following SQL script:

```

] IF TYPE_ID('[dbo].[CreatingObjectInfo]') IS NULL
] BEGIN
]     CREATE TYPE CreatingObjectInfo AS TABLE (
        Id NVARCHAR(36),
        ObjectModifiedOn NVARCHAR(128),
        CommunicationTypeId NVARCHAR(36),
        CityId NVARCHAR(36),
        Name NVARCHAR(128),
        UsrINN NVARCHAR(50),
        Number NVARCHAR(250),
        SearchNumber NVARCHAR(250),
        Web NVARCHAR(250)
    );
] END;
GO

```

2. Add `UsrINN` to `@parsedConfig`. To do so, execute the following SQL script:

```

IF @xmlRows <> ''
BEGIN
    SET @minimumGroupCount = 0;

```

```

SET @xmlRowsConfig = CAST(@xmlRows AS XML);
INSERT INTO @parsedConfig
SELECT
    NULLIF(b.value('(.//Id/text())[1]', 'NVARCHAR(36)'), NEWID()) AS [Id],
    NULLIF(b.value('(.//ContactModifiedOn/text())[1]', 'NVARCHAR(128)'), AS
[ObjectModifiedOn]),
    NULLIF(b.value('(.//CommunicationTypeId/text())[1]', 'NVARCHAR(36)'), AS
[CommunicationTypeId],
    NULL AS [CityId],
    NULLIF(b.value('(.//Name/text())[1]', 'NVARCHAR(128)'), AS [Name],
    NULLIF(b.value('(.//UsrINN/text())[1]', 'NVARCHAR(50)'), AS [UsrINN],
    [dbo].[fn_NormalizeString](NULLIF(b.value('(.//Number/text())[1]',
'NVARCHAR(250)'), ''), N'0-9a-zA-Z@.') AS [Number],
    [dbo].[fn_ExtractDigitLimitFromNumber](LTRIM(
[dbo].[fn_GetPhoneNumberSearchForm](NULLIF(b.value('(.//Number/text())[1]',
'NVARCHAR(250)'), '')),
)) AS [SearchNumber],
    NULLIF([dbo].[fn_ExtractDomainFromUrl](b.value('(.//Number/text())[1]',
'NVARCHAR(250)'), '')) AS [Web]
    FROM @xmlRowsConfig.nodes('/rows/row') as a(b);
    SET @processingRowId = (SELECT TOP 1 Id FROM @parsedConfig);
END;

```

3. Install *tsp_FindDuplicate* in the database.

11. Execute the installation process for remote stored procedures

Execute the database installation process for all stored procedures that have been removed from the database as described in step 9.

12. Add the *tsp_FindAccountDuplicateByInn* stored procedure

Add the user-stored *tsp_FindAccountDuplicateByInn* procedure and complete the procedure with an *ELSE* block when adding records to *#searchAccount*. To do so, execute the following SQL script:

```

IF @parsedConfigRowCount = 0
BEGIN
    INSERT INTO #searchAccount ([Name], [SortDate])
    SELECT
        [dedup].[Name],
        MAX([dedup].[SortDate]) [SortDate]
    FROM (
        SELECT [Id]
            [dbo].[fn_NormalizeString]([Name], @validChar) AS [Name],
            MAX([ModifiedOn]) [SortDate]
        FROM [VmAccountCleanDataValues] WITH (NOEXPAND)
        GROUP BY [Id], [Name]
    ) AS [dedup]
    GROUP BY [dedup], [Name]
    HAVING COUNT(*) > 1;
END;
ELSE
BEGIN
    INSERT INTO #searchAccount ([Name], [SortDate])
    SELECT
        [dbo].[fn_NormalizeString]([Name], @validChar) AS [Name],
        GETDATE() AS [SortDate]
    FROM @parsedConfig
END;

```

Install *tsp_FindAccountDuplicateByInn* in the database.

Junk case custom filtering

Beginner

Easy

Medium

Advanced

Introduction

Creatio users can filter unwanted cases by creating a list of email addresses and domains for automatic spam detection. Incoming cases from these domains or addresses are either not registered at all or have the “Canceled” status (the default status value is set in the [Junk Case Default Status] system setting).

Junk filter enables email header analysis based on certain flags, e.g. *Auto-Submitted*. Emails with those flags are either not registered or register with the pre-defined initial status, set in system settings.

To analyze email addresses and domains, simply add the required value to the [Blacklist of email addresses and domains for case registration] lookup. The emails will be marked as blacklisted during the analysis when you populate the lookup with values (their type is set automatically).

The algorithm of adding a new email filtering property

Email analysis is done through the [Email header properties management] lookup. Follow these steps to add a new analysis property:

1. Add a new class that implements the *IHeaderPropertyHandler* interface. This interface contains a single *Check()* method that returns a value of *bool* type. Check the property value and return the result during the *Check()* method implementation. If the method returns *true*, the system creates the case using the standard mechanism, if *false*, the system treats the email as blacklisted.
2. Add an [Email header properties management] lookup value. Specify the property name used for analysis in the *Name* column. Specify the class name (added in the previous paragraph) in the *handler* column.

Adding a new email filtering property

Case description

Add a new *No-reply* property for case analysis. If the property is found in the email header and its value is anything other than “No”, treat the case as blacklisted.

Case implementation algorithm

1. Add a new class that implements the *IHeaderPropertyHandler* interface.

Add a “Source code” schema in a custom package (e.g. *Custom*). In this schema, define a class that implements the *IHeaderPropertyHandler* interface, e.g.:

```
namespace Terrasoft.Configuration
{
    using System;
    public class NoreplyHandler: IHeaderPropertyHandler
    {
        public bool Check(object value) {
            return string.Equals(value.ToString(), "No",
StringComparison.OrdinalIgnoreCase);
        }
    }
}
```

Save and publish the new schema.

The *IHeaderPropertyHandler* interface is defined in the *JunkFilter* package, so it must be added to the custom package dependency.

2. Adding a lookup value

Add the *No-reply* property to the [Email header properties management] lookup. Select the *NoreplyHandler* class (created in the previous paragraph) as the handler class (*handler* property).

After receiving an email with a *No-reply* header flag and a value that is anything other than “No”, the following options are possible:

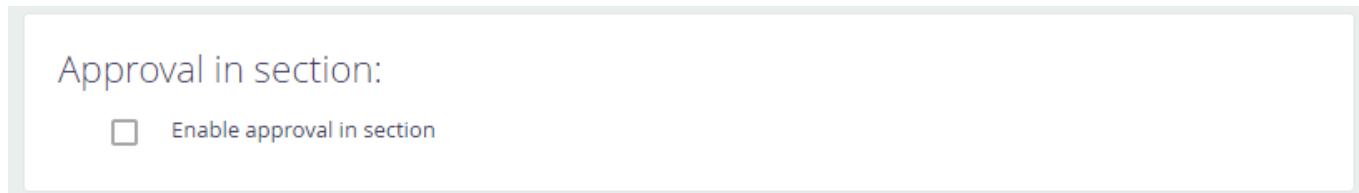
- the case is not created if the [Create Cases From Junk Emails] system setting value is *false*;
- the case is created with the [Junk Case Default Status] system setting status if the [Create Cases From Junk Emails] system setting value is *true*.

How to display custom implementation of approving in the section wizard

Beginner Easy Medium Advanced

Starting with version 7.11 Creatio can now implement approving functions and informing about approvals in any section. Approvals are enabled in the section wizard. The section wizard has the [Enable approval in section] checkbox to enable approvals (Fig. 1). Previous versions of Creatio needed a project complex solutions to enable approving functions.

Fig. 1. [Enable approval in section] checkbox in the section wizard



If the custom approving had been already implemented in Creatio, follow instructions below to enable the [Enable approval in section] checkbox in the section wizard.

1. Add a record in the SysModuleVisa table

To do so, execute the following SQL script:

```
insert SysModuleVisa (
    VisaSchemaUID,
    MasterColumnUID,
    UseCustomNotificationProvider)
select
    'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX',
    'YYYYYYYY-YYYY-YYYY-YYYY-YYYYYYYYYYYY',
    0
```

where

- *VisaSchemaUID* – *Uid* of user object inherited from the [Base approval] object.
- *MasterColumnUID* – *Uid* of the field of interaction with the section.
- *UseCustomNotificationProvider* – a flag of using custom provider. “0” if you need to use custom provider by default, “1” if user has created own message provider.

2. Update a record for the section in the SysModule table

In the *SysModule* table for corresponding section sill the *SysModuleVisaId* field with value of the *Id* of added record in the *SysModuleVisa* table. To do so, execute the following SQL script:

```
update SysModule
set SysModuleVisaId = 'ZZZZZZZZ-ZZZZ-ZZZZ-ZZZZ-ZZZZZZZZZZ'
where Code='KnowledgeBase'
```

How to create custom reminders and notifications

Beginner

Easy

Medium

Advanced

Introduction

Starting with version 7.12.0, reminder and notification sending mechanics has been reworked in Creatio.

Previously, to send a custom notification, you would have to:

- Creates a class that implements *INotificationProvider* interface or an inherited abstract *BaseNotificationProvider* class.
- Add logic for selecting custom notifications by Creatio.
- Register a class in the *NotificationProvider* table.

The notifications were sent once a minute, calling all classes from the *NotificationProvider* table.

Starting with version 7.12.0, it is sufficient to create a notification or reminder with the needed parameters. After this, the application will either send the notification immediately, or display a reminder at the specified time.

To set up custom notifications:

1. **Create a [Source code] schema** in the custom package and define a class for generating the notification text and pop-up window. The class must implement the *IRemindingTextFormer* interface (declared in the *IRemindingTextFormer* schema of the *Base* package).
2. Replace the needed object (such as [Lead]) or specify notification sending logic in it.
3. Replace the reminder tab schema *ReminderNotificationsSchema* for displaying notifications for the needed object.

Case description

Create a custom reminder about opportunity actualization date in leads. The date must be specified in the [Next actualization date] field on the [Opportunity info] tab.

Source code

A package with implemented example is available for download via the following [link](#).

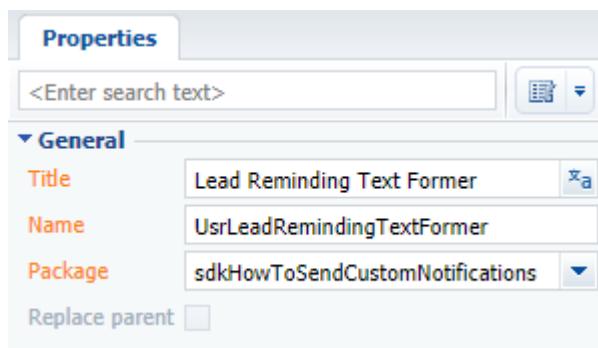
Case implementation algorithm

1. Create a class for generating the reminder text and the pop-up window.

1. Add a [Source code] schema in the custom package (see “**Creating the [Source code] schema**”). Set the following properties (Fig. 1):

- [Title] – “Lead Reminding Text Former”.
- [Name] – “UsrLeadRemindingTextFormer”

Fig. 1. The [Source code] schema properties



2. Use the context menu of the [Structure] tab to add two localized strings (Fig. 2). Properties of the localized strings are described in table 1.

Fig. 2. Adding localized strings to schema

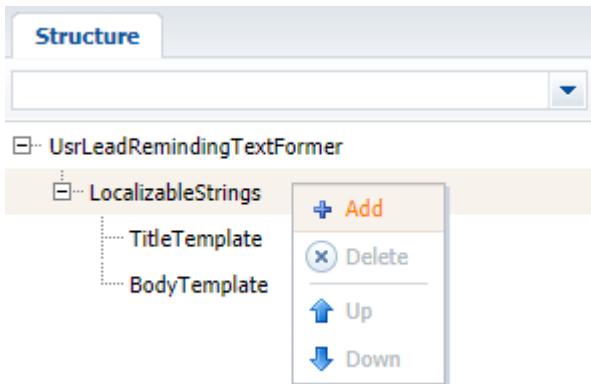


Table 1. Localized string properties

Name	Value
TitleTemplate	You need to update the sale
BodyTemplate	Lead {o} requires update of sales information

3. Add class implementation for forming reminder text and pop-up window:

```
namespace Terrasoft.Configuration
{
    using System.Collections.Generic;
    using Terrasoft.Common;
    using Terrasoft.Core;

    public class UsrLeadRemindingTextFormer : IRemindingTextFormer
    {
        private const string ClassName = nameof(UsrLeadRemindingTextFormer);
        protected readonly UserConnection UserConnection;

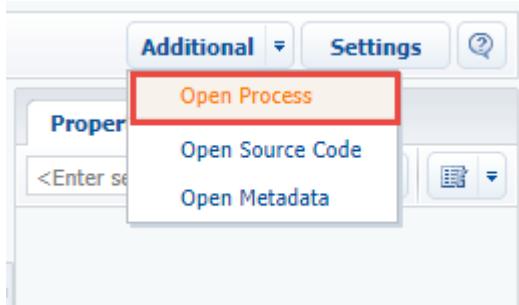
        public UsrLeadRemindingTextFormer(UserConnection userConnection)
        {
            UserConnection = userConnection;
        }
        // Generates reminder text from a collection of inbound parameters and
        BodyTemplate localized string.
        public string GetBody(IDictionary<string, object> formParameters)
        {
            formParameters.CheckArgumentNull("formParameters");
            var bodyTemplate = UserConnection.GetLocalizableString(ClassName,
            "BodyTemplate");
            var leadName = (string)formParameters["LeadName"];
            var body = string.Format(bodyTemplate, leadName);
            return body;
        }
        // Generates reminder title from the class name and TitleTemplate localize
        string.
        public string GetTitle(IDictionary<string, object> formParameters)
        {
            return UserConnection.GetLocalizableString(ClassName, "TitleTemplate");
        }
    }
}
```

4. Save and publish the schema.

2. Replace the [Lead] object and set the reminder logic in it.

1. Create a replacing schema of the [Lead] object (see “Crating a replacing object schema” section of the “**Creating the entity schema**” article).
2. Click [Additional] and select [Open process]. Built-in process of the replacing [Lead] object will open (Fig. 3).

Fig. 3. Opening built-in process



3. Using the context menu in the [Structure] tab, add process parameter *GenerateReminding* (Fig. 4) with the following properties (Fig. 5):

- [Title] – “Generate Reminding”.
- [Name] – "GenerateReminding".
- [Data type] – “Boolean”.

Fig. 4. Adding a process parameter

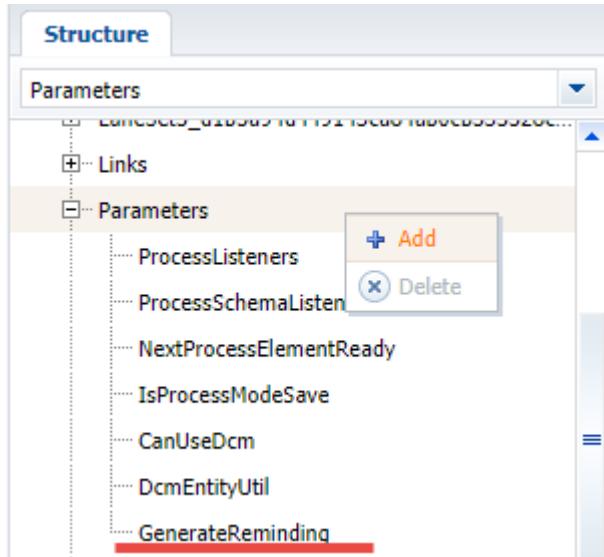
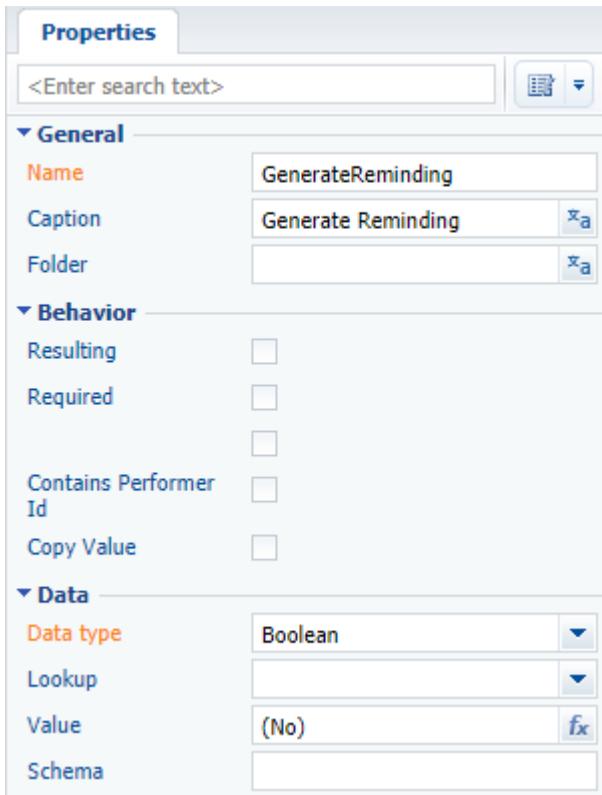


Fig. 5. The properties for the process parameter

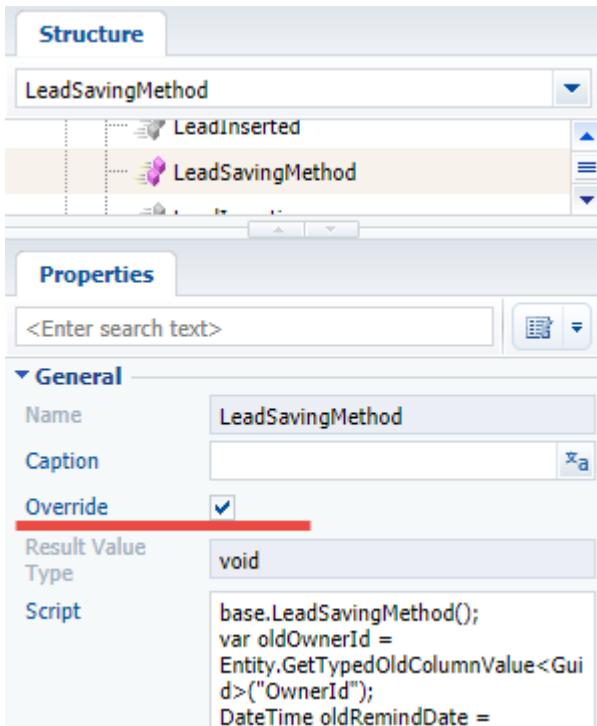


4. Select the *LeadSavingMethod()* (called before saving the object) in the [Methods] node on the [Structure] tab (Fig. 6). Select the [Override] checkbox and add the following code to the method body:

```
// Calling base implementation of the method.
base.LeadSavingMethod();
// Getting owner Id.
var oldOwnerId = Entity.GetTypedOldColumnValue<Guid>("OwnerId");
// Getting next actualization date.
DateTime oldRemindDate = Entity.GetTypedOldColumnValue<DateTime>
("NextActualizationDate");
// Comparing Id of original and current owner.
bool ownerChanged = !Entity.OwnerId.Equals(oldOwnerId);
// Comparing original and current actualization dates.
bool remindDateChanged = !Entity.NextActualizationDate.Equals(oldRemindDate);
// Set the GenerateReminding process parameter value to "true" if the owner
// or actualization date changed.
GenerateReminding = ownerChanged || remindDateChanged;
```

To open source code editor of a method, double-click the method name on the [Structure] tab.

Fig. 6. Properties of the LeadSavingMethod



5. Select the *LeadSavedMethod()* (called after saving the object) in the [Methods] node on the [Structure] tab (Fig. 7). Select the [Override] checkbox and add the following code to the method body:

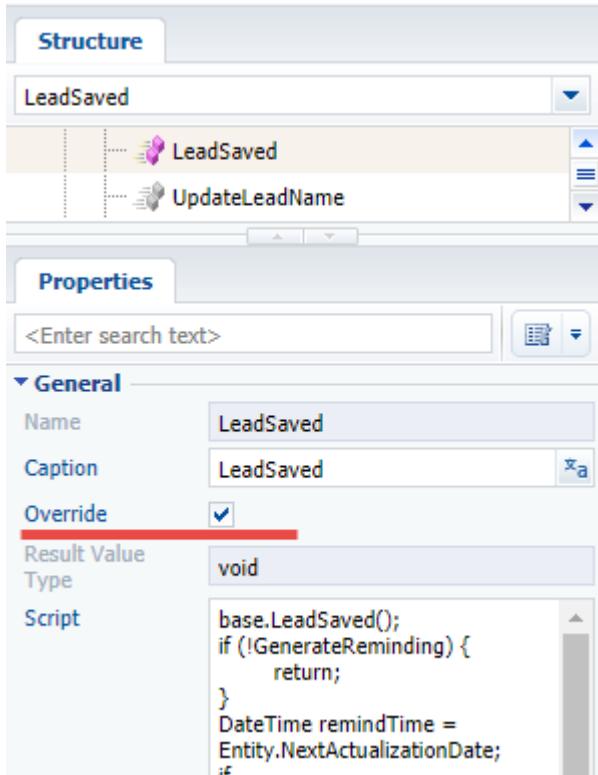
```
base.LeadSaved();
// Checks if the reminder must be generated.
if (!GenerateReminding) {
    return;
}
DateTime remindTime = Entity.NextActualizationDate;
if (Entity.OwnerId.Equals(Guid.Empty) || remindTime.Equals(default(DateTime))) {
    return;
}
// Instantiates the UsrLeadRemindingTextFormer class.
IRemindingTextFormer textFormer =
    ClassFactory.Get<UsrLeadRemindingTextFormer>(new
ConstructorArgument("userConnection", UserConnection));
// Gets lead name.
string leadName = Entity.LeadName;
// Gets the reminder text.
string subjectCaption = textFormer.GetBody(new Dictionary<string, object> {
    {"LeadName", leadName}
});
// Gets reminder title.
string popupTitle = textFormer.getTitle(null);
// Configures reminder.
var remindingConfig = new RemindingConfig(Entity);
// Message author - current contact.
remindingConfig.AuthorId = UserConnection.CurrentUser.ContactId;
// Target recipient - lead owner.
remindingConfig.ContactId = Entity.OwnerId;
// Type - reminder.
remindingConfig.NotificationTypeId = RemindingConsts.NotificationTypeRemindingId;
// Reminder date - next actualization date of an opportunity in lead.
remindingConfig.RemindTime = remindTime;
// Reminder text.
remindingConfig.Description = subjectCaption;
// Reminder title.
```

```

remindingConfig.PopupTitle = popupTitle;
// Creating reminder utility class.
var remindingUtilities = ClassFactory.Get<RemindingUtilities>();
// Creating reminder.
remindingUtilities.CreateReminding(UserConnection, remindingConfig);

```

Fig. 7. Properties of the LeadSavedMethod



To display reminders on the system notification tab , replace `remindingConfig.NotificationTypeId = RemindingConsts.NotificationTypeRemindingId;` with `remindingConfig.NotificationTypeId = RemindingConsts.NotificationTypeNotificationId;` in the code of the LeadSavedMethod.

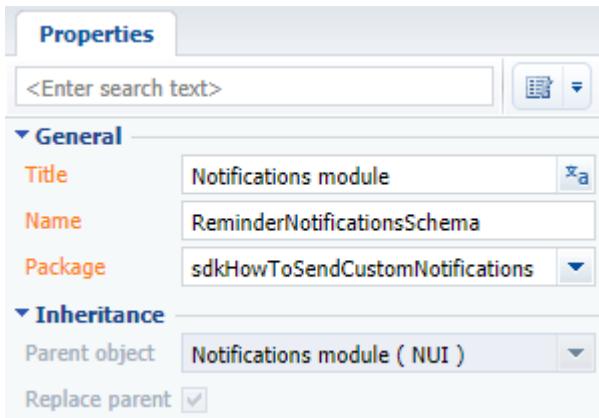
6. Save and publish built-in process schema of the [Lead] object. Publish the [Lead] object schema.

3. Replace the reminder tab schema ReminderNotificationsSchema.

1. To display reminders for a specific object, create a replacing `ReminderNotificationsSchema` in the **custom package** and add the needed logic to it. The procedure for creating a replacing client schema is covered in the “[Creating a custom client module schema](#)” article. Replacing schema properties (Fig. 8):

- [Title] – “Notifications module”.
- [Name] – “ReminderNotificationsSchema”.

Fig. 8. Properties of the ReminderNotificationsSchema schema



2. Add following source code on the [Source code] tab of the schema:

```
define("ReminderNotificationsSchema", ["ReminderNotificationsSchemaResources"],
  function() {
    return {
      entitySchemaName: "Reminding",
      methods: {
        // Determines of the reminder is related to the lead.
        getIsLeadNotification: function() {
          return this.get("SchemaName") === "Lead";
        },
        // Gets reminder title.
        getNotificationSubjectCaption: function() {
          var caption = this.get("Description");
          return caption;
        }
      },
      // Array of view model notifications.
      diff: [
        // Reminder primary container.
        {
          "operation": "insert",
          "name": "NotificationleadItemContainer",
          "parentName": "Notification",
          "propertyName": "items",
          "values": {
            "itemType": Terrasoft.ViewItemType.CONTAINER,
            "wrapClass": [
              "reminder-notification-item-container"
            ],
            // Displays only for leads.
            "visible": {"bindTo": "getIsLeadNotification"},
            "items": []
          }
        },
        // Title container.
        {
          "operation": "insert",
          "name": "NotificationItemleadTopContainer",
          "parentName": "NotificationleadItemContainer",
          "propertyName": "items",
          "values": {
            "itemType": Terrasoft.ViewItemType.CONTAINER,
            "wrapClass": ["reminder-notification-item-top-container"],
            "items": []
          }
        }
      ]
    }
  }
);
```

```

// Image.
{
    "operation": "insert",
    "name": "NotificationleadImage",
    "parentName": "NotificationItemleadTopContainer",
    "propertyName": "items",
    "values": {
        "itemType": Terrasoft.ViewItemType.BUTTON,
        "className": "Terrasoft.ImageView",
        "imageSrc": {"bindTo": "getNotificationImage"},
        "classes": {"wrapClass": ["reminder-notification-icon-class"]}}
},
},
// Date display.
{
    "operation": "insert",
    "name": "NotificationDate",
    "parentName": "NotificationItemleadTopContainer",
    "propertyName": "items",
    "values": {
        "itemType": Terrasoft.ViewItemType.LABEL,
        "caption": {"bindTo": "getNotificationDate"},
        "classes": {"labelClass": ["subject-text-labelClass"]}}
},
},
// Reminder text display.
{
    "operation": "insert",
    "name": "NotificationleadSubject",
    "parentName": "NotificationItemleadTopContainer",
    "propertyName": "items",
    "values": {
        "itemType": Terrasoft.ViewItemType.LABEL,
        "caption": {"bindTo": "getNotificationSubjectCaption"},
        "click": {"bindTo": "onNotificationSubjectClick"},
        "classes": {"labelClass": ["subject-text-labelClass", "label-link", "label-url"]}}}
},
]
}
);
});
```

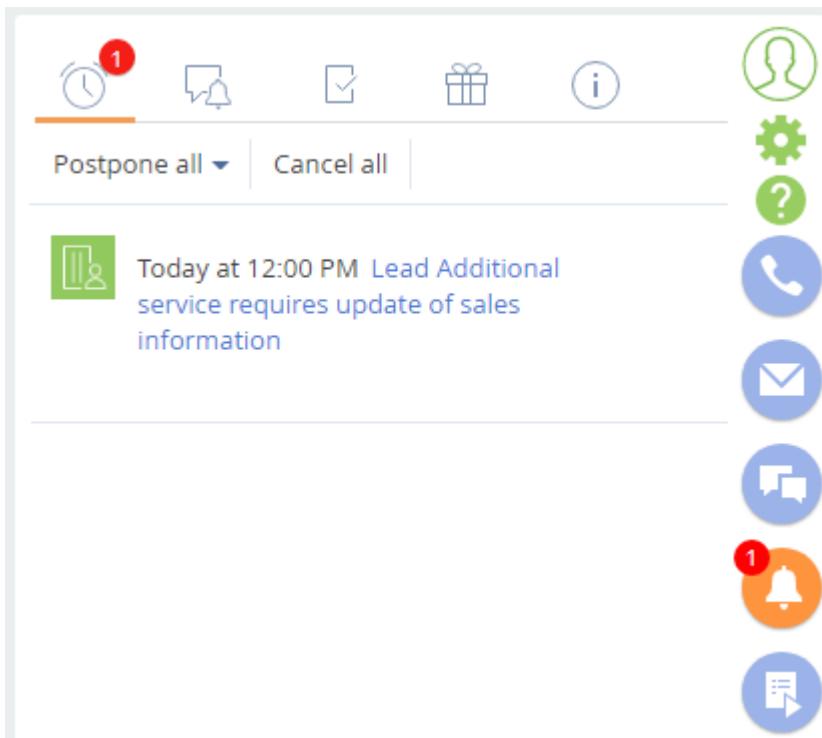
3. Save the schema.

As a result, Creatio will create reminders (Fig. 10) for all leads where the owner and actualization date are specified (Fig. 9).

Fig. 9. Lead page where owner and next actualization date fields populated

The screenshot shows a lead record for "Additional service". The left sidebar contains fields like Customer need*, Additional service, Registration method, Added manually, Budget (0.00), Created on (3/1/2018 at 12:24 PM), and Owner (Supervisor). The main area displays a timeline with stages: Qualification, Nurturing, Handoff to sales, Awaiting sale, and Satisfied. Below the timeline, there are "NEXT STEPS (0)" icons for phone, email, message, and flag. A horizontal navigation bar includes LEAD INFO, CUSTOMER NEED DETAILS, LEAD ENGAGEMENT, TIMELINE, and OPPORTUNITY INFO (which is selected). Under OPPORTUNITY INFO, it shows Opportunity information with Budget (0.00), Next actualization date (3/2/2018 at 12:00 PM), Opportunity/Order, and Deal owner.

Fig. 10. Lead custom reminder



Adding multi-language email templates to a custom section

Beginner **Easy** **Medium** **Advanced**

Introduction

You can set up custom logic for selecting languages of multi-language email templates. You can select email templates in the needed language using the action dashboard of a section record. The selection is based on special rules that can be specified for the section. If special rules are not defined, the selection is based on the contact, bound to the edited record (the [Contact] column). If a section object does not have a column for connecting with a contact, the *DefaultMessageLanguage* system setting value is used.

To add custom logic for selecting multi-language templates (localization):

1. Create a class or classes inherited from *BaseLanguageRule* and define the language selection rules (one class

defines one rule).

2. Create a class inherited from *BaseLanguageIterator*. Define the *LanguageRules* property in the class constructor as a class instance array created on the previous step. The sequence corresponds to the rule priority.
3. Create a class inherited from *AppEventListenerBase* that will bind the class defining the language selection rules to the section.
4. Add the necessary multi-language templates to the [Email templates] lookup.

Case description

Add logic of selecting an email template language to a custom section based on the *UsrContact* column of the primary section object. Use English and Spanish languages.

Source code

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

You can install the package for Creatio products, containing the *EmailTemplates* package. Make sure all the below described preliminary settings are performed after you install the package.

Preliminary settings

For correct case implementation:

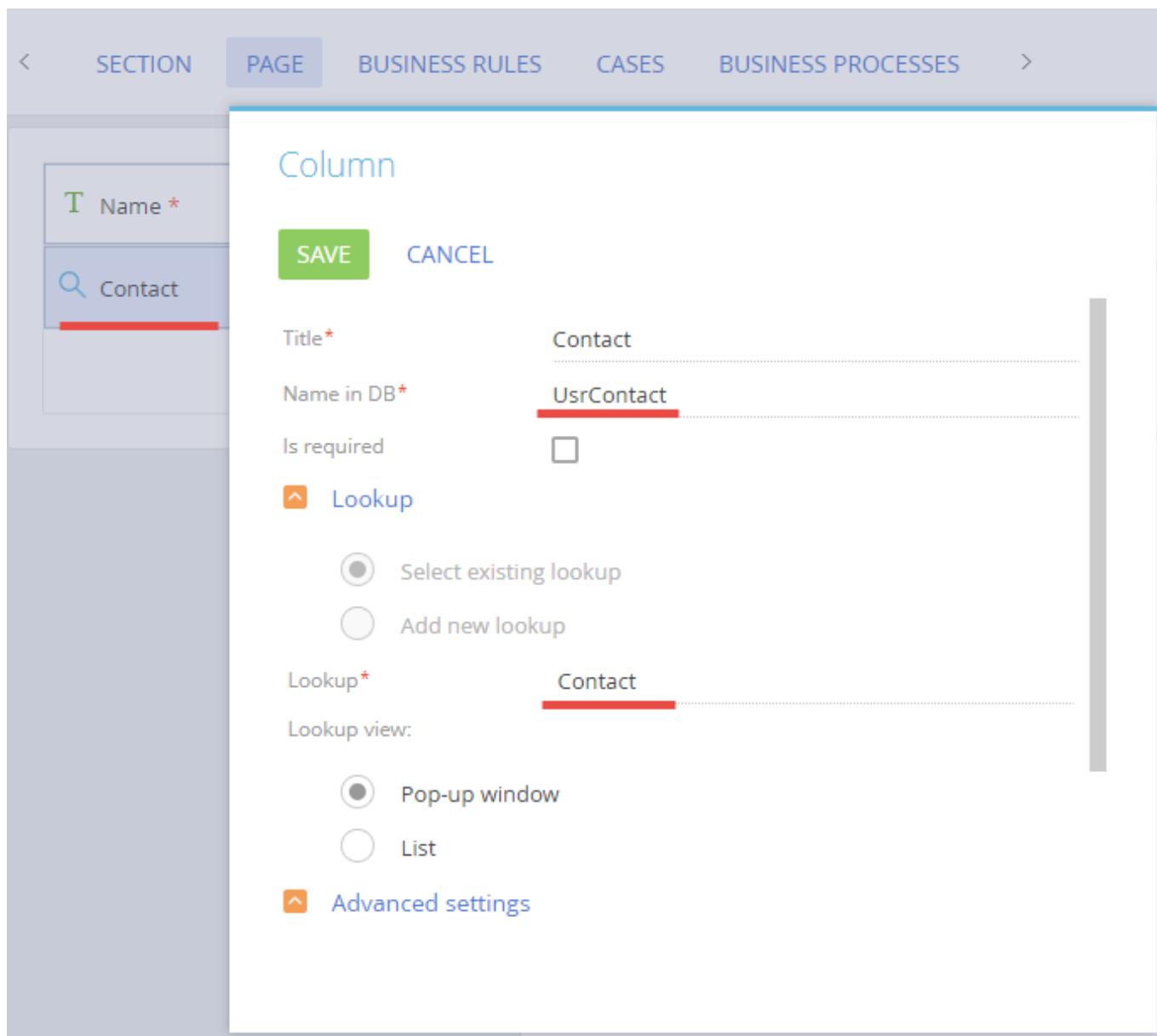
1. Make sure that the [Customer languages] lookup contains English and Spanish languages (Fig.1).

Fig. 1. [Customer languages] lookup

Customer languages			
Name	Description	Code	Is used
English (United St...)	English (United St...)	en-US	Yes
Spanish (Spain)	Español (España, ...)	es-ES	Yes

2. Use the section wizard to check that there is the *UsrContact* column bound to the [Contact] lookup on the edit page of the custom section record (Fig.2).

Fig. 2. The UsrContact column



Case implementation algorithm

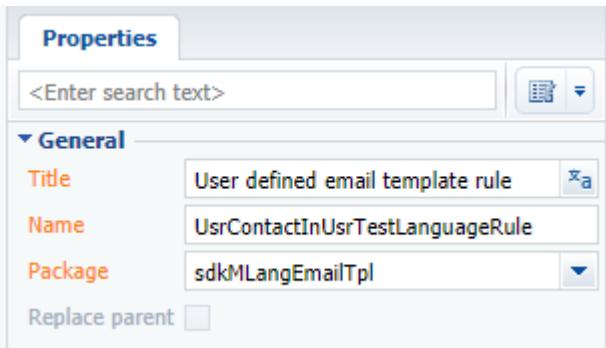
1. Adding a language selection rule

Create a [Source code] schema in the custom package (see “[Creating the \[Source code\] schema](#)”).

For the created schema specify (Fig. 3):

- [Name] – "UsrContactInUsrTestLanguageRule"
- [Title] – "User defined email template rule"

Fig. 3. The [Source code] schema properties



Add the following source code to the schema:

```
namespace Terrasoft.Configuration
{
    using System;
    using Terrasoft.Core;
    using Terrasoft.Core.Entities;
    public class ContactInUsrTestLanguageRule : BaseLanguageRule
    {
        public ContactInUsrTestLanguageRule (UserConnection userConnection) :
base(userConnection)
        {
        }
        // Defines the user preferred language identifier.
        // recId - current record identifier.
        public override Guid GetLanguageId(Guid recId)
        {
            // Creating the EntitySchemaQuery instance for the custom section primary
object.
            var esq = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"UsrMLangEmailTpl");
            // Defining the contact language column name.
            var languageColumnName = esq.AddColumn("UsrContact.Language.Id").Name;
            // Obtaining current record instance.
            Entity usrRecEntity = esq.GetEntity(UserConnection, recId);
            // Obtaining the value of user preferred language identifier.
            Guid languageId = usrRecEntity.GetTypedColumnValue<Guid>
(languageColumnName);
            return languageId;
        }
    }
}
```

Publish the schema.

2. Defining the order sequence of language selection rules

Create a [Source code] schema in the custom package (see “**Creating the [Source code] schema**”).

For the created schema specify (Fig. 3):

- [Name] – "UsrTestLanguageIterator"
- [Title] – "User defined language iterator"

Add the following source code to the schema:

```
namespace Terrasoft.Configuration
{
    using Terrasoft.Core;
    public class UsrTestLanguageIterator: BaseLanguageIterator
    {
```

```
public UsrTestLanguageIterator(UserConnection userConnection):  
base(userConnection)  
{  
    // Language selection rule array.  
    LanguageRules = new ILanguageRule[] {  
        // Custom rule.  
        new ContactInUsrTestLanguageRule (UserConnection),  
        // Default rule.  
        new DefaultLanguageRule(UserConnection),  
    };  
}  
}  
}
```

DefaultLanguageRule, is the second array element. The rules uses the *DefaultLanguage* system setting for obtaining the language and is used by default if the language was not detected by higher priority rules.

Publish the schema.

3. Binding language selection iterator to the section

Create a [Source code] schema in the custom package (see “[Creating the \[Source code\] schema](#)”).

For the created schema specify (Fig. 3):

- [Name] – "UsrTestMLangBinder"
 - [Title] – "UsrTestMLangBinder"

Add the following source code to the schema:

```
namespace Terrasoft.Configuration
{
    using Terrasoft.Core.Factories;
    using Terrasoft.Web.Common;
    public class UsrTestMLangBinder: AppEventListenerBase
    {
        public override void OnAppStart(AppEventArgs context)
        {
            // Calling the basic logics.
            base.OnAppStart(context);
            // Binding iterator to a custom section.
            // UsrMLangEmailTpl - name of the section primary object.
            ClassFactory.Bind<ILanguageIterator, UsrTestLanguageIterator>(
("UsrMLangEmailTpl"));
        }
    }
}
```

Publish the schema.

4. Adding the necessary multi-language templates

Add a new record (Fig.4) to the [Email Templates] lookup and define the email templates in the necessary languages (Fig.5).

Fig. 4. A new record in the [Email templates] lookup

Email templates

 Filters/folders ▾

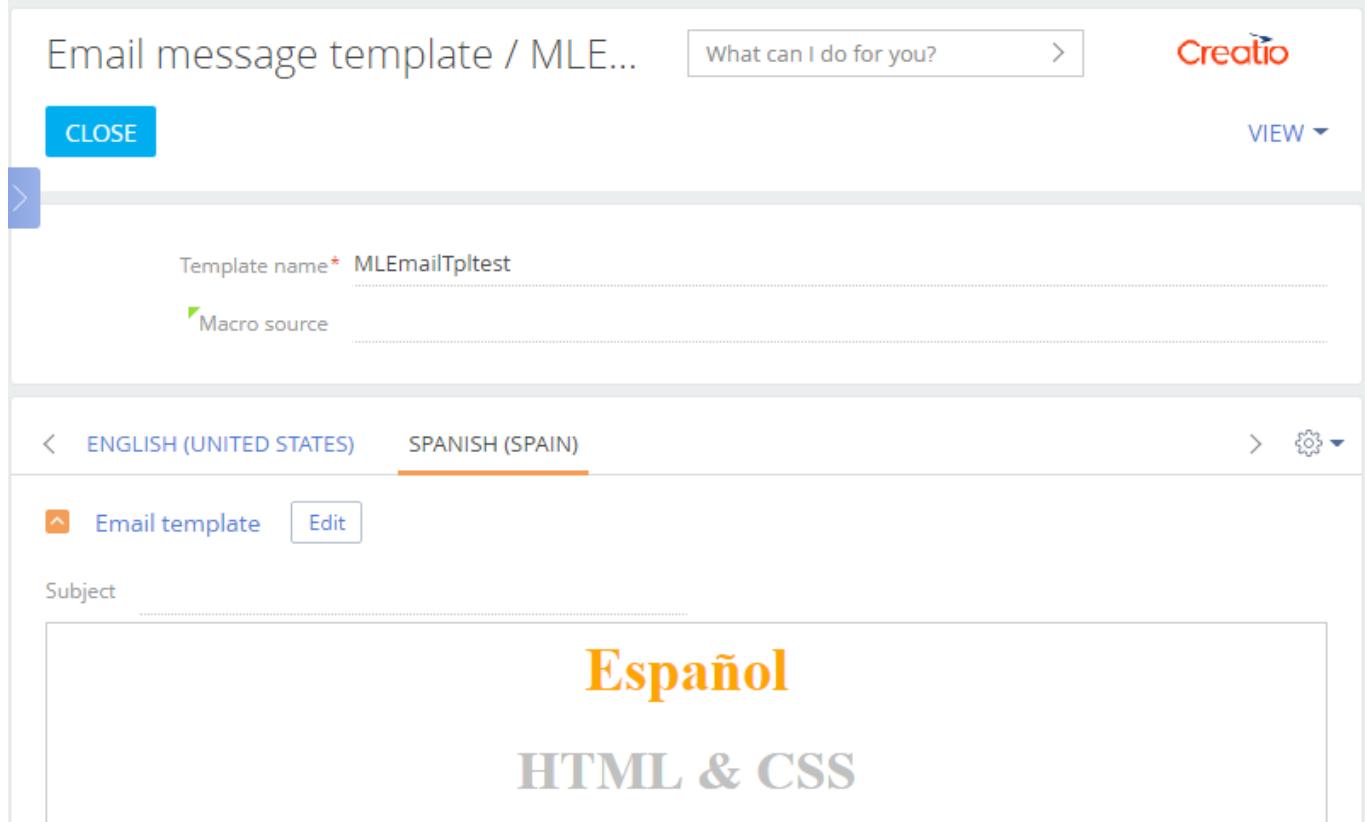
Case feedback request notification

Subject

New message on case #[#Number#]

MLEmailTpltest (US, ES)

Fig. 5. Adding templates in the necessary languages



Email message template / MLE...

What can I do for you? > **Creatio**

CLOSE VIEW ▾

Template name* **MLEmailTpltest**

Macro source

ENGLISH (UNITED STATES) SPANISH (SPAIN)

Email template Edit

Subject

Español

HTML & CSS

As a result of case implementation, in the action dashboard panel (Fig. 6. 1) of the custom section record edit page (Fig.6) the email templates (Fig. 6. 2) will be selected automatically in the language (Fig. 6. 3) specified as the contact's preferred language (Fig.7).

Fig. 6. Case result

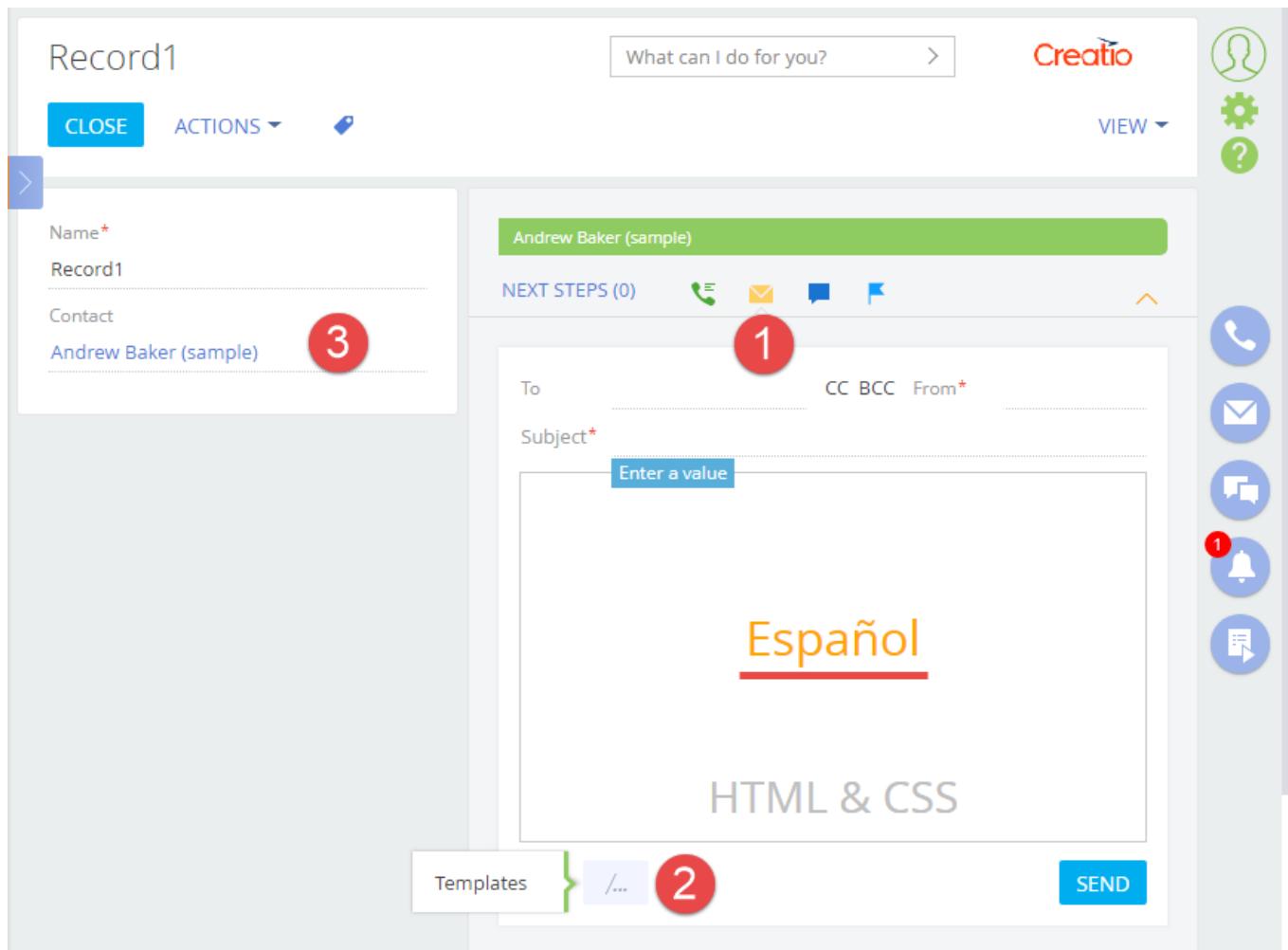


Fig. 7. Contact preferred language

Working with email threads

Beginner **Easy** **Medium** **Advanced**

Introduction

The mechanism for creating email threads is available in Creatio since version 7.10.0. The main purpose of this mechanism is to improve the email interface. Use this mechanism to easily find email connections, e.g. by copying the connections of the previous email.

The thread construction mechanism is based on the *In-Reply-To* email header data. According to generally accepted agreements, this header should contain the *Message-ID* email identifier. The Creatio application retrieves these headers from the synchronized Email service (IMAP / Exchange).

The mechanism can be divided into two logical parts – creating threads and populating activity connection fields.

Creating email threads

The *EmailMessageData* detail contains 3 fields:

- *MessageId* – a string of 500 characters in length;
- *InReplyTo* – a string of 500 characters in length;
- *ParentMessageId* – a lookup field that references the *EmailMessageData* detail.

The *MessageId* and *InReplyTo* fields are populated with the corresponding message header values during synchronization.

The *ParentMessageId* field is populated with the following values:

1. The *EmailMessageData* table searches for records where *MessageId* is identical to *InReplyTo*. The first found record is set as the current *ParentMessageId*.
2. The *ParentMessageId* field is populated with the current *Id* value in all *EmailMessageData* records if the *InReplyTo* field of these records is identical to the *MessageId* field.

Email thread connections are updated for every email.

Copying previous email connections in a thread

Spreading activity connections across the thread upon adding an email. The [Case] field is used in this case.

A parent record with the activity that includes the populated [Contact] field is searched for the current *EmailMessageData* record. The [Case] field values from this activity will be spread across the thread. Starting with the found *EmailMessageData* record, all child *EmailMessageData* records are found and their [Case] field value is updated.

A thread with 3 emails:

ActivityId	Title	CaseId	EmailMessageId	EmailMessageParentId
28BD6D59-B9D7-4FF9-89F5-FEE1DD003912	Re: relation	NULL	66812FBF-411B-4FE0-94C9-1E70FBEB2D3	F05B529D-C98C-4E26-BE00-21F8721AEF58
DCoA40D4-700A-40EB-B394-90E0376C3B5D	Re: relation	1C6E18E3-48B1-495E-8EF9-ACA35DB9DEoB	F05B529D-C98C-4E26-BE00-21F8721AEF58	E1A0120E-B7Co-4261-9DE0-C63341BF1EoB
D7A9B82C-ED46-437C-980A-B2650D4FF3DA	relation	906909E8-4D64-47FD-AF92-B65B0826AEC3	E1A0120E-B7Co-4261-9DE0-C63341BF1EoB	NULL

Another email is received in the thread:

ActivityId	Title	CaseId	EmailMessageId	EmailMessageParentId
6623B052-73AD-4AE5-AE61-A6F9BCD930Ao	Re: relation	NULL	60CooB01-DoBF-40F6-923E-1830E433AEA1	66812FBF-411B-4FE0-94C9-1E70FBEB2D3
28BD6D59-B9D7-4FF9-89F5-FEE1DD003912	Re: relation	NULL	66812FBF-411B-4FE0-94C9-1E70FBEB2D3	F05B529D-C98C-4E26-BE00-21F8721AEF58
DCoA40D4-700A-40EB-B394-90E0376C3B5D	Re: relation	1C6E18E3-48B1-495E-8EF9-ACA35DB9DEoB	F05B529D-C98C-4E26-BE00-21F8721AEF58	E1A0120E-B7Co-4261-9DE0-C63341BF1EoB
D7A9B82C-ED46-437C-980A-B2650D4FF3DA	relation	906909E8-4D64-47FD-AF92-B65B0826AEC3	E1A0120E-B7Co-4261-9DE0-C63341BF1EoB	NULL

Starting with the record in which *EmailMessageId* is "60CooB01-DoBF-40F6-923E-1830E433AEA1", a record with the populated *CaseId* column is searched (does not contain *NULL*). This is a record where *EmailMessageId* = "F05B529D-C98C-4E26-BE00-21F8721AEF58", and *CaseId* = "1C6E18E3-48B1-495E-8EF9-ACA35DB9DEoB".

Now, starting with the record in which *EmailMessageId* is "F05B529D-C98C-4E26-BEoo-21F8721AEF58", Creatio updates the value of the *CaseId* field for the connected records throughout the thread.

ActivityId	Title	CaseId	EmailMessageId	EmailMessageParentId
6623B052-73AD-4AE5-AE61-A6F9BCD930Ao	Re: relation	1C6E18E3-48B1-495E-8EF9-ACA35DB9DEoB	60C00B01-DoBF-40F6-923E-1830E433AEA1	66812FBF-411B-4FE0-94C9-1E70FBEB2D3
28BD6D59-B9D7-4FF9-89F5-FEE1DD003912	Re: relation	1C6E18E3-48B1-495E-8EF9-ACA35DB9DEoB	66812FBF-411B-4FE0-94C9-1E70FBEB2D3	F05B529D-C98C-4E26-BEoo-21F8721AEF58
DCoA40D4-700A-40EB-B394-90E0376C3B5D	Re: relation	1C6E18E3-48B1-495E-8EF9-ACA35DB9DEoB	F05B529D-C98C-4E26-BEoo-21F8721AEF58	E1A0120E-B7Co-4261-9DEo-C63341BF1EoB
D7A9B82C-ED46-437C-980A-B2650D4FF3DA	relation	906909E8-4D64-47FD-AF92-B65B0826AEC3	E1A0120E-B7Co-4261-9DEo-C63341BF1EoB	NULL

Mail servers sometimes send out letters in the wrong sequence, disregarding the way they were written in the thread (e.g. during synchronization, emails are received first from the inbox and then from the outbox). This complicates the mechanism. Building a thread for the emails that were not received consistently is impossible during synchronization. In that case, a thread can be built once the synchronization is complete. If certain mailbox folders are not loaded, or if the conversation was interrupted by other participants, the thread search logic may not work. However, the thread can be built when all emails are downloaded from the inbox in most cases.

Recommendations for adding a custom search process for all thread connections after downloading an email

Use the following guidelines to start working on a new email after thread formation:

1. The *Id* field of the synchronization session appears in the *EmailMessageData* table (the values are unique for all synchronization processes). This field is populated only for synchronized emails.
2. A record is added to the new *FinishedSyncSession* table if at least one email was received during synchronization.
3. Certain processes that responded to the signal of saving activities now respond to the (*Inserted*) insertion signal of the *FinishedSyncSession* object. Emails from *EmailMessageData* are selected based on the synchronization session *Id*. The *MailboxSyncSettings* field of the *EmailMessageData* object can be used to select Email messages from the service box.

Integration of third-party sites via iframe

Beginner

Easy

Medium

Advanced

Introduction

One way to integrate external solutions in the Creatio is to use the *iframe* HTML element.

The *iframe* HTML element is used to display third-party web page inside the web page where the element is placed. The *iframe* element is implemented in the HTML code of the page via the `<iframe>` and `</iframe>` tags. URL of the displayed page is set using the *src* attribute. More information about this element can be found in the [article](#).

The third-party web application can be implemented to Creatio with the *iframe* element. The advantage of this approach is the convenience of viewing the third-party web resources (pages, video, etc.) directly from the Creatio.

Note, that some sites prohibit uploading of their pages into the *iframe* element.

To exchange data between Creatio and third-party web applications it is recommended to use the **DataService** service or the **OData** protocol.

The `Terrasoft.controls.IframeControl` component is implemented in the client part of the Creatio core. This component is used to display custom HTML markup in the Creatio. For example, it is used, on the email templates

edit page of the [\[Email templates\]](#) lookup.

Integration case

Case description

Create a [WEB] tab on the record edit page in the [Accounts section]. The tab will contain a site which URL will be specified in the [Web] field.

Case implementation algorithm

Place the component on the record edit page of the [Accounts] section.

For this, create the [Account edit page] replacing schema in the custom package. The procedure for creating a replacing schema is covered in the “**Creating a custom client module schema**”. Add the following source code to the replacing schema:

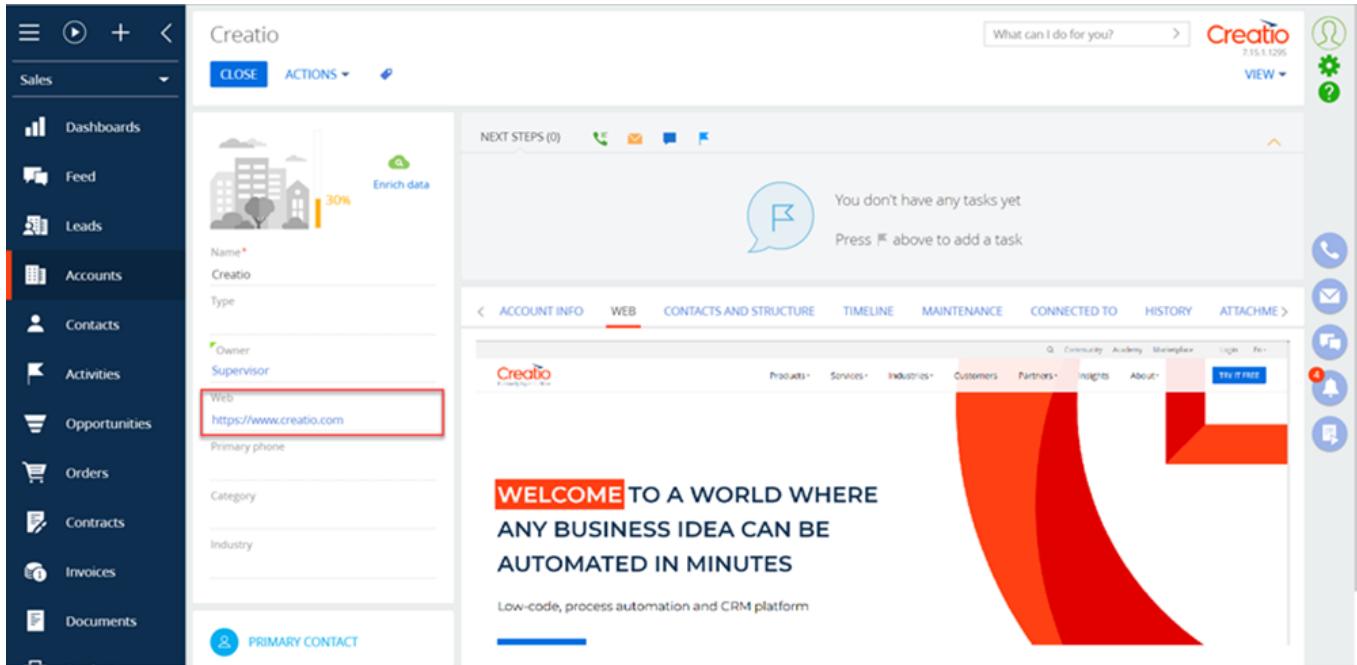
```
define("AccountPageV2", [], function() {
    return {
        entitySchemaName: "Account",
        diff: /**SCHEMA_DIFF*/ [
            // Adding the [WEB] tab.
            {
                "operation": "insert",
                "name": "WebTab",
                "values": {
                    "caption": "WEB",
                    "items": []
                },
                "parentName": "Tabs",
                "propertyName": "tabs",
                "index": 1
            },
            // Adding IFrameControl component.
            {
                "operation": "insert",
                "name": "UsrIframe",
                "parentName": "WebTab",
                "propertyName": "items",
                "values": {
                    "itemType": Terrasoft.ViewItemType.IFRAMECONTROL,
                    "src": {
                        "bindTo": "getSource"
                    }
                }
            }
        ]/**SCHEMA_DIFF*/,
        methods: {
            // Used to bind data.
            getSource: function() {
                return this.get("Web");
            }
        }
    };
});
```

Here, the configuration objects of the [WEB] tab and the component for displaying *Terrasoft.controls.IframeControl*. Binding the data of the [Web] column to the *src* property of the component is performed with the *getSource()* method.

After saving the schema and reloading the application page, the [WEB] tab will appear on the edit page of a section record. The tab will display a web page by the URL specified in the [Web] field (Fig. 1). If the [Web] field is empty

then the tab will be empty too.

Fig. 1. Case result



Analytics

Contents

- **Basic macros in the MS Word printables**
- **How to create macros for a custom report in Word**

Basic macros in the MS Word printables

Beginner Easy Medium Advanced

Introduction

Users can create MS Word reports in Creatio and configure them using the Creatio MS Word Report Designer plug-in. Learn more about creating and setting up MS Word reports in the "[Set up MS Word reports](#)" article.

Use macros to set up extra data output options for an MS Word report. You can use basic macros and create **custom ('How to create macros for a custom report in Word' in the on-line documentation)** macros.

The general procedure of creating an MS Word report using basic macros is as follows:

1. Install the MS Word Report Designer plug-in. This is a one-time procedure. Learn more in the "[Installing Creatio plug-in for MS Word](#)" article.
2. Add a new report record in the **[Report setup]** section.
3. Set up the report display parameters
4. Set up the report data fields and tables. Add a tag with the name of the macro in the **[#MacroName#]** format to the column when setting up column fields.
5. Set up the report template layout in the Creatio MS Word Report Designer plug-in and upload the template to Creatio.

The structure of macros in MS Word reports:

ColumnName [#Macro name | Arguments#]

Basic macros

The [#Date#] macro

Converts a date to a specified date format. The default date format is "dd.MM.yyyy". If the date format is not specified, the entered date value will be converted to the default format. A detailed description of date formats is available in the [Microsoft documentation](#). The argument is optional.

Cases:

ColumnName [#Date#]

If the entered value is "07/15/2020 11:48:24 AM", the macro will return "15.07.2020".

ColumnName [#Date|MM/dd/yyyy#]

If the entered value is "31/01/2019 08:25:48 AM", the macro will return "01/31/2019".

The [#Lower#] macro

Converts the value of a string to lowercase. The macro has no arguments.

An example:

ColumnName [#Lower#]

If the entered value is "ExaMpLe", the macro will return "example".

The [#Upper#] macro

Converts the value of a string to uppercase. The argument is optional. If the "FirstChar" argument is passed to the macro, only the first character in the string will be converted to uppercase.

Cases:

ColumnName [#Upper#]

If the entered value is "example", the macro will return "EXAMPLE".

ColumnName [#Upper|FirstChar#]

If the entered value is "example", the macro will return "Example".

The [#NumberDigit#] macro

Convert a raw number to a number with digit group separators. The default delimiter is the space character. The argument is optional.

Cases:

ColumnName [#NumberDigit#]

If the entered value is "345566777888.567", the macro will return "345 566 777 888.567".

ColumnName [#NumberDigit|, #]

If the entered value is "345566777888.567", the macro will return "345,566,777,888.567".

If the fractional part of the number equals zero, only the integer part will be returned. If the entered value is "345566777888.000", the macro will return "345,566,777,888".

The [#Boolean#] macro

Converts a boolean value to a custom representation. The argument is required. The following arguments are available:

- *CheckBox* – converts the entered value to "☒" or "☐".
- *Yes,No* – converts the entered value to "Yes" or "No".

Cases:

ColumnName [#Boolean|CheckBox#]

If the column contains the *true* value, the macro will return "☒".

ColumnName [#Boolean|Yes, No#]

If the column contains the *true* value, the macro will return "Yes".

Case description

Create an "Account Info" report for the **[Accounts]** section edit page to display the following information about the account:

- **[Name]**.
- **[Type]**.
- **[Primary contact]**.

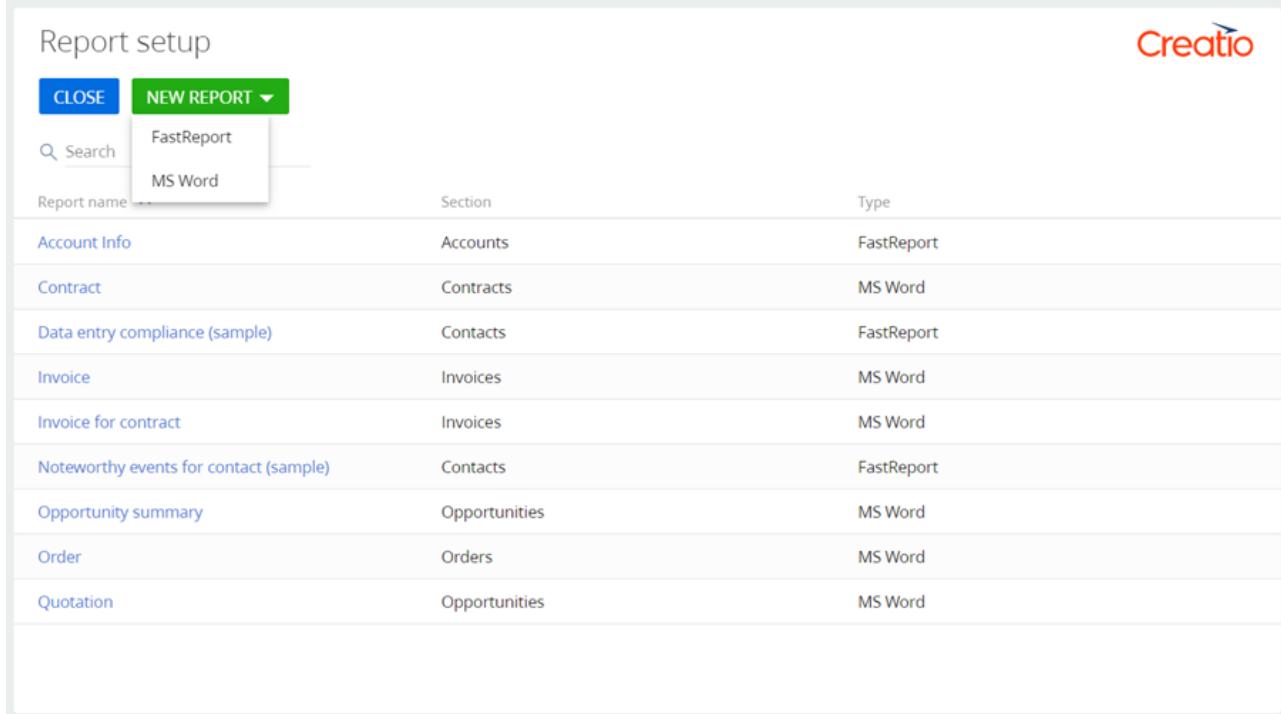
Case implementation algorithm

1. Create a new report

To do this:

1. Open the System Designer by clicking . In the **[System setup]** block, click the **[Report setup]** link.
2. Click **[New report] → [MS Word]** (Fig. 1).

Fig. 1. The [Report setup] section list



Report setup		
	Section	Type
Account Info	Accounts	FastReport
Contract	Contracts	MS Word
Data entry compliance (sample)	Contacts	FastReport
Invoice	Invoices	MS Word
Invoice for contract	Invoices	MS Word
Noteworthy events for contact (sample)	Contacts	FastReport
Opportunity summary	Opportunities	MS Word
Order	Orders	MS Word
Quotation	Opportunities	MS Word

2. Set up the report display parameters

Set the following values (Fig. 3) in the parameter setup area (Fig. 2, 2):

- **[Report title]** – "Account Info".
- **[Section]** – "Accounts".
- **[Show in the section list view]**.
- **[Show in the section record page]**.

Fig. 2. MS Word report setup page

New report

SAVE **CANCEL**

Report name*
New report

Section*
Type*
MS Word

Show in the section list view
 Show in the section record page

In order to download to a PC or upload a Word report template to Creatio, use the steps below:

Drag and drop *.docx template here or **Select file**

i Note

To setup report:
1. Install the Microsoft Word plugin.
2. Add the report's fields and table components.
This page contains all the necessary settings.
3. Open Microsoft Word and configure the

Fig. 3. – Setting up the report display parameters

Account Info

SAVE

CANCEL

Report name*

Account Info

Section*

Accounts

Type*

MS Word

Show in the section list view

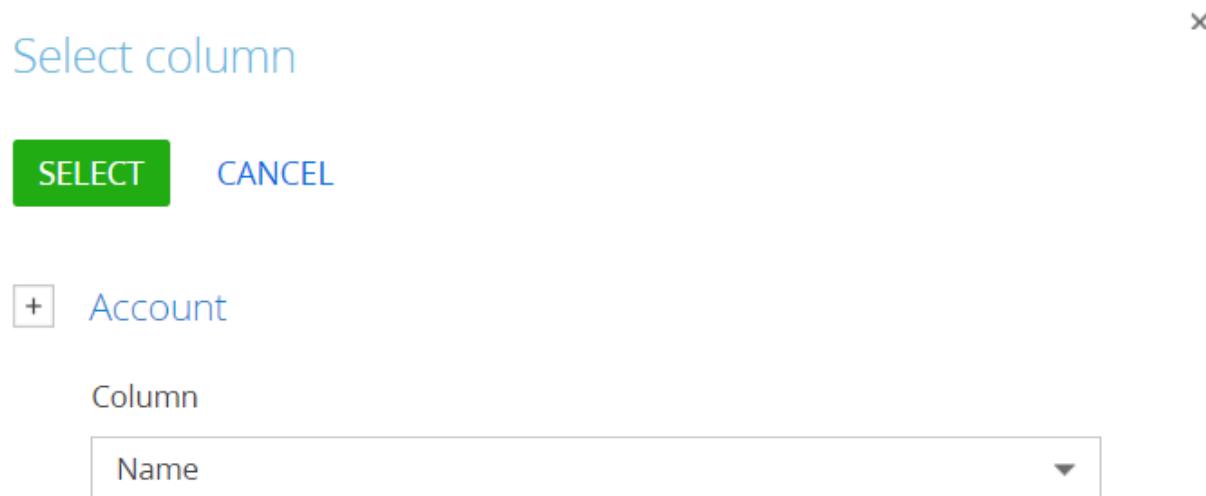
Show in the section record page

2. Set up the report fields

In the **[Set up report data]** block of the section working area (Fig. 2, 5), set up the fields to display in the report. To do

this, click  and select the **[Name]** column in the drop-down **[Column]** list (Fig. 4). A macro will be added to the column later.

Fig. 4. Selecting a column

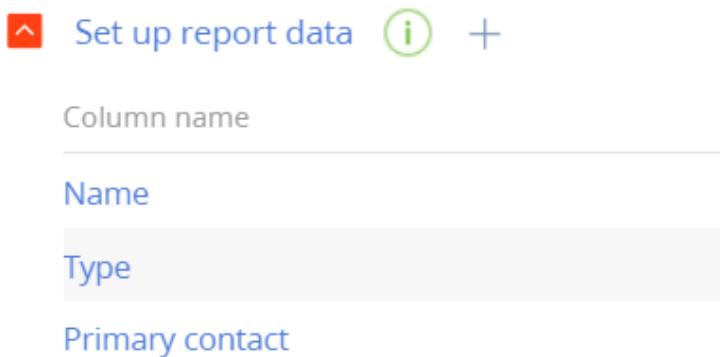


Click **[Select]**.

Using the same method, add the **[Type]** and **[Primary contact]** columns to the template.

The list of columns after this step is presented below (Fig. 5).

Fig. 5. The list of added columns



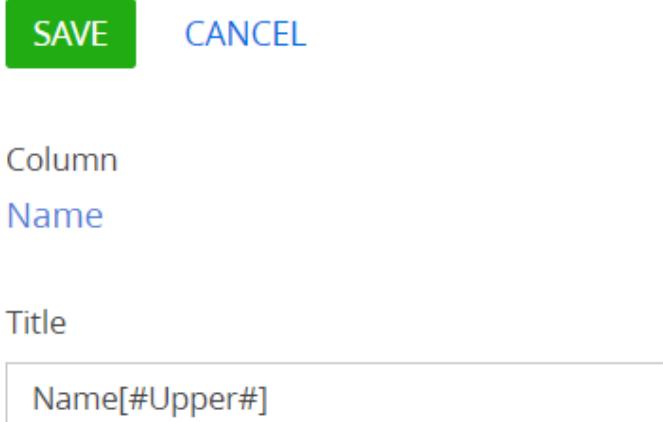
3. Add the macro tag to the column name

Change the **[Name]** column property. To do this, take the following steps:

1. In the **[Set up report data]** block of the section working area (Fig. 2, 5), double-click the title of the **[Name]** column or click  in the column title bar.

2. Change the [Name] value of the [Title] field to [Name[#Upper#]] (Fig. 6).

Fig. 6. Setting up a column property



Click [Save].

The list of columns after adding macro tags is presented below (Fig. 7).

Fig. 7. The list of added columns with macros

The screenshot shows the 'Set up report data' interface. At the top left is a red 'X' button, followed by 'Set up report data' and a help icon. Below is a table with three rows:

Column name	Type	Primary contact
Name[#Upper#]		

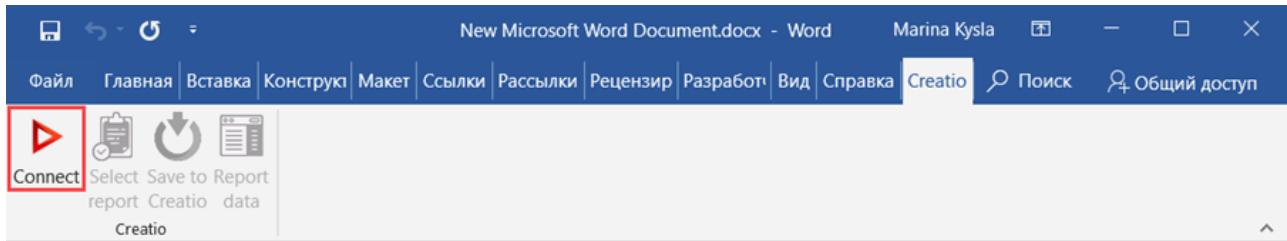
Click [Save].

4. Set up the report template layout and upload the template to Creatio

To set up the template:

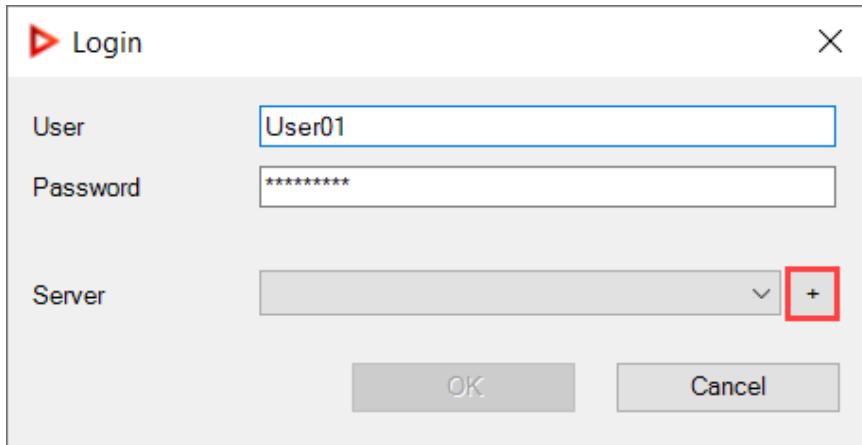
1. Open any MS Word file.
2. Click [Connect] on the Creatio plug-in toolbar (Fig. 8).

Fig. 8. The [Connect] button



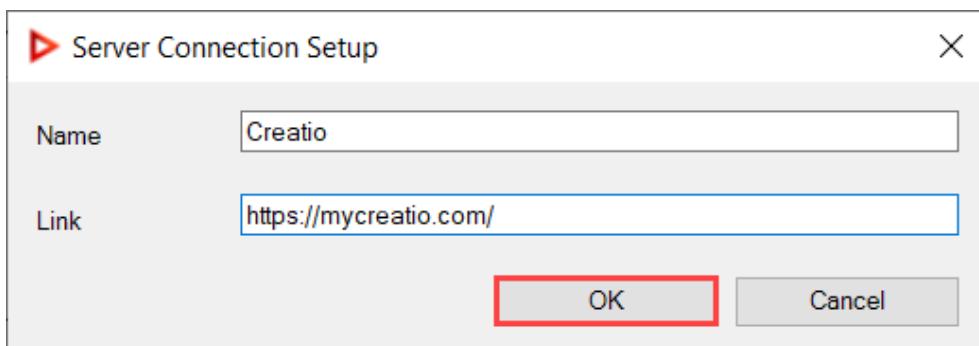
3. Enter the username and password of the Creatio user. Click next to the [Server] field (Fig. 9).

Fig. 9. Authentication



4. Click [New]. Enter the server parameters (Fig. 10).

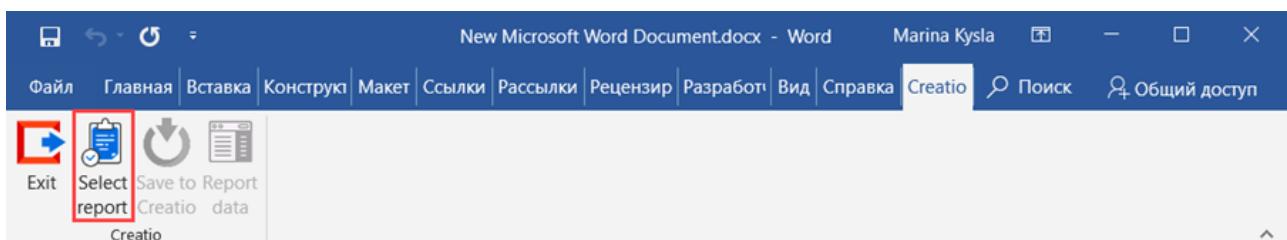
Fig. 10. Server settings



Click [OK].

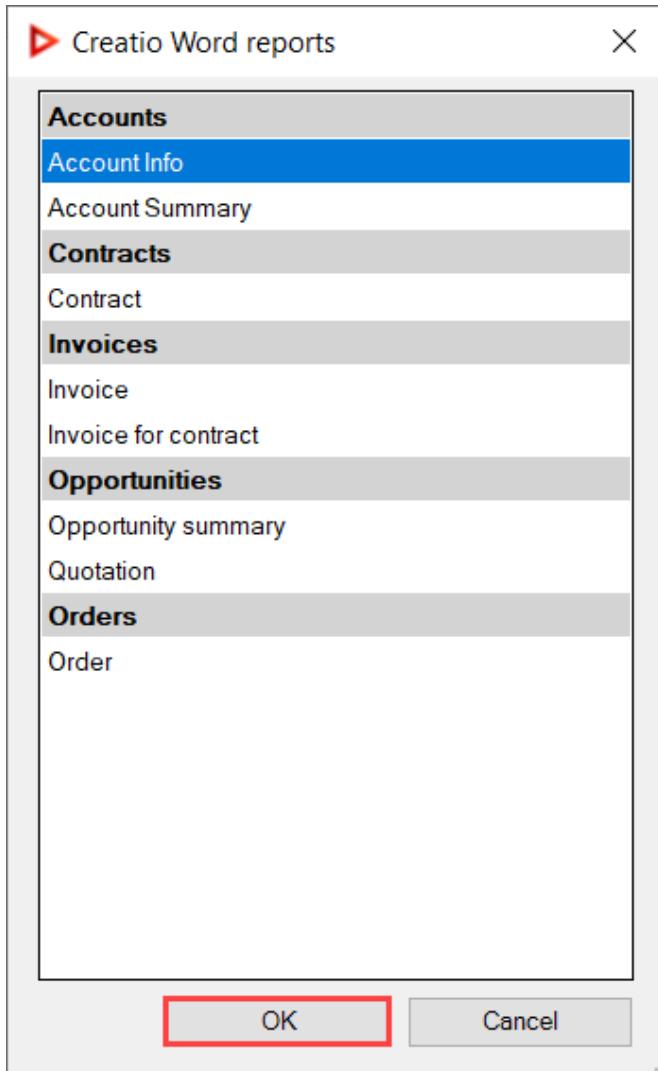
5. Click [Select report] on the Creatio plug-in toolbar (Fig. 11).

Fig. 11. The [Select report] button



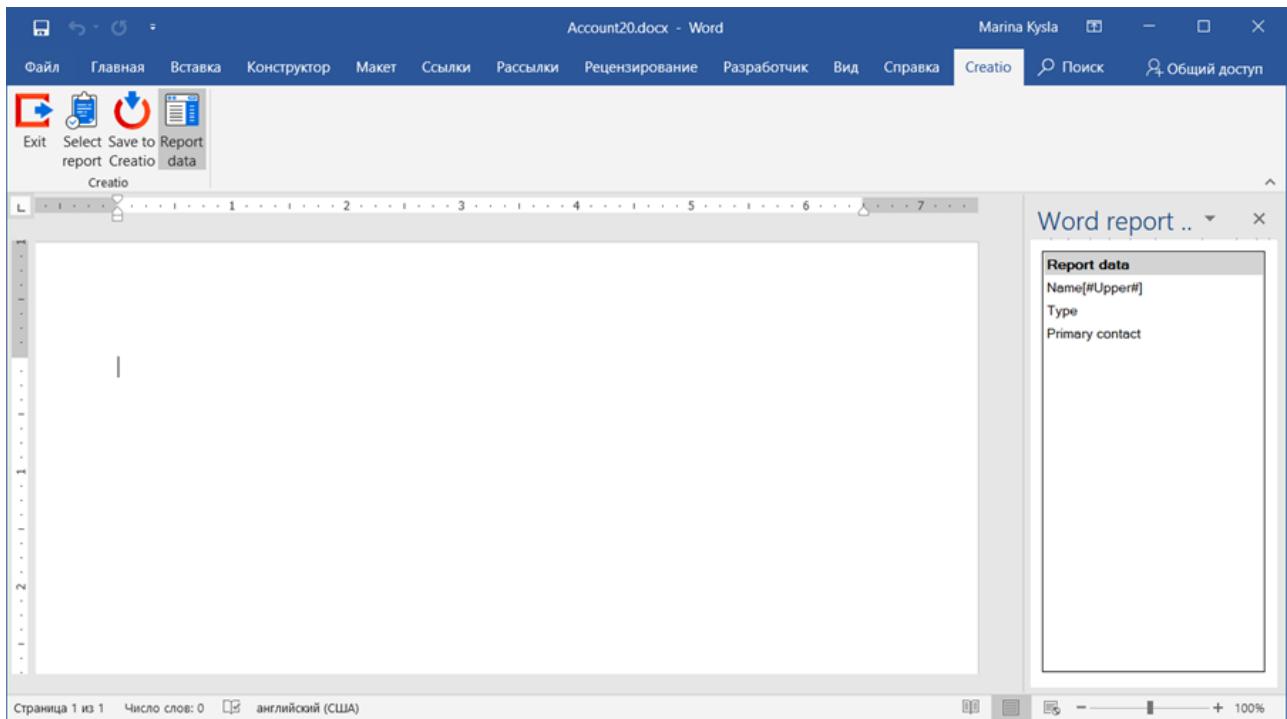
6. Select the "Account info" report and click [OK] (Fig. 12).

Fig. 12. Selecting a report



The report setup window looks as follows (Fig. 13):

Fig. 13. Setting up the report



- Set up the template layout. Learn more about setting up a report template in the "[Design report layout via the Creatio MS Word plug-in](#)" article.

After the setup, the report looks as follows (Fig. 14):

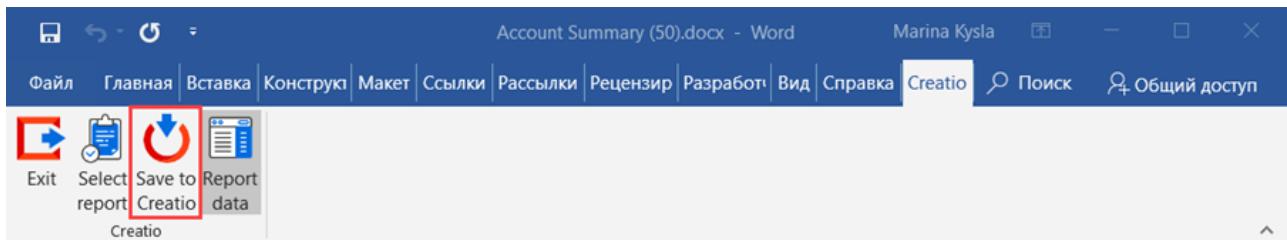
Fig. 14. Setting up a report template

Account Info

Name	«Name[#Upper#]»
Type	«Type»
Primary contact	«Primary contact»

- Click **[Save to Creatio]** to load the report template in Creatio (Fig. 15).

Fig. 15. The [Save to Creatio] button



As a result, the "Account Info" report will be available in the **[Accounts]** section list under **[Print]** (Fig. 16).

Fig. 16. Displaying the "Account Info" report in the dashboards of the [Accounts] section

Name	Web	Primary phone	Type
Vertigo Systems	www.vertigosys.com	+44 (20) 3427 1374	Customer
Sunrise Investments	www.sunrise-invest.co.uk	+44 (15) 1437 1598	Customer
RealWay	www.realway.co.uk	+44 (20) 3425 2139	Customer
FlashNet Consulting	www.flashnetconsulting.com	+1 206 429 1595	Customer

The report looks as follows (Fig. 17).

Fig. 17. – Example of the "Account Info" report

Account Info

Name	VERTIGO SYSTEMS
Type	Customer
Primary contact	Peter Moore

Transferring the package to another development environment

To transfer the package with the report to another development environment, go to the **[Configuration]** section -> the **[Data]** tab and **bind the data** of the following elements:

- *SysModuleReport_ReportName* – the report. To bind it, use the report Id from the **[dbo.SysModuleReport]** database table.
- *SysModuleReportTable_ReportName* – the tabular component of the report. To bind it, use the report Id from the **[dbo.SysModuleReportTable]** database table.

You can view the record Id in the database table even if you do not have access to the database. To do this, display the Id system column in the window of binding data to packages.

How to create macros for a custom report in Word

Beginner Easy Medium Advanced

Introduction

The **[Report setup]** section enables users to create MS Word reports using Creatio tools and configure them using the Creatio MS Word Report Designer plug-in. Learn more about creating and setting up MS Word reports in the "[Set up MS Word reports](#)" article.

The general procedure of creating an MS Word report using custom macros is as follows:

1. Install the MS Word Report Designer plug-in. This is a one-time procedure. Learn more in the "[Installing Creatio plug-in for MS Word](#)" article.
2. Add a new report record in the **[Report setup]** section.
3. Set up the report display parameters
4. Implement custom macros.

5. Set up the report data fields and tables.
6. Set up the report template layout in the Creatio MS Word Report Designer plug-in and upload the template to Creatio.

Working with macros

Use macros to set up extra data output options for an MS Word report. You can use **basic macros ('Basic macros in the MS Word reports' in the on-line documentation)** and create custom macros. Use custom macros to implement custom MS Word reports. A macro for setting up an MS Word report is a class implementing the *IExpressionConverter* interface (see the *ExpressionConverterHelper* schema of the *NUI* package).

To make a custom macro callable from the report template, mark the macro with the *ExpressionConverterAttribute* attribute containing the name of the macro. For example:

```
[ExpressionConverterAttribute("CurrentUser")]
```

The *Evaluate(object value, string arguments = "")* interface method must be implemented in the class. The method accepts an MS Word report template field value as an argument and returns the *string* type value that will be inserted instead of this field in the ready MS Word report.

The general procedure of creating a custom MS Word report macro is as follows:

1. Create a new report in the **[Report setup]** section.
2. Set up the report display parameters.
3. Set up the report data fields and tables.
4. Add the **[Id]** column to the list of report columns that will be the incoming parameter for the custom macro.
5. Add a schema of the **[Source Code]** type with a class implementing the *IExpressionConverter* interface to the custom package. The class must be marked by the *ExpressionConverterAttribute* attribute with the name of the macro. Implement the *Evaluate(object value, string arguments = "")* method in it.
6. Publish the **[Source Code]** type object schema.
7. Add a tag with the name of the custom macro in the **[#MacroName#]** format to the **[Id]** column when setting up column fields.

Case description

Create an "Account Summary" report for the **[Accounts]** section edit page to display the following information about the account:

- **[Name]**.
- **[Type]**.
- **[Primary contact]**.
- **[Additional info]**. The annual revenue should be displayed for **[Customer]** accounts and the number of employees for **[Partner]** accounts.

The report must contain information about the date of creation and the name of the employee who created it.

Source code

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

Case implementation algorithm

1. Create a new report

To do this:

1. Open the System Designer by clicking . In the **[System setup]** block, click the **[Report setup]** link.
2. Click **[New report] -> [MS Word]** (Fig. 1).

Fig. 1. The **[Report setup]** section list

The screenshot shows the 'Report setup' interface in Creatio. At the top left is a 'CLOSE' button and a 'NEW REPORT ▾' button. To the right is the 'Creatio' logo. A search bar labeled 'Search' is positioned next to the new report button. A dropdown menu from the 'NEW REPORT' button lists 'FastReport' and 'MS Word'. Below this is a table with the following data:

Report name	Section	Type
Account Info	Accounts	FastReport
Contract	Contracts	MS Word
Data entry compliance (sample)	Contacts	FastReport
Invoice	Invoices	MS Word
Invoice for contract	Invoices	MS Word
Noteworthy events for contact (sample)	Contacts	FastReport
Opportunity summary	Opportunities	MS Word
Order	Orders	MS Word
Quotation	Opportunities	MS Word

2. Set up the report display parameters

Set the following values (Fig. 3) in the parameter setup area (Fig. 2, 2):

- **[Report title]** – "Account Summary".
- **[Section]** – "Accounts".
- **[Show in the section list view].**
- **[Show in the section record page].**

Fig. 2. MS Word report setup page

The screenshot shows the 'New report' setup page. At the top left are 'SAVE' and 'CANCEL' buttons. A blue circle with the number '1' is above the 'Report name*' field, which contains 'New report'. Below it is a 'Section*' field with 'Type*' set to 'MS Word'. Two checkboxes, 'Show in the section list view' and 'Show in the section record page', are shown with blue circles containing '2' and '3' respectively. A note at the bottom says: 'In order to download to a PC or upload a Word report template to Creatio, use the steps below:'. A blue circle with '4' points to a 'Note' icon. Another blue circle with '5' points to the 'Set up report data' section, which currently shows 'No data'.

Fig. 3. – Setting up the report display parameters

Account Summary

SAVE **CANCEL**

Report name*
Account Summary

Section*
Accounts

Type*
MS Word

Show in the section list view
 Show in the section record page

3. Implement custom macros

Go to the [Advanced settings] section -> **[Configuration]** -> Custom package -> the **[Schemas]** tab. Click **[Add]** -> **[Source code]** (Fig. 4). Learn more about creating a schema of the **[Source Code]** type in the "**Creating the [Source code] schema**" article.

Fig. 4. Adding the [Source Code] schema

Schemas: All ▾ External Assemblies: All

Add ▾ Edit Delete

Standard

- Object
- Replacing Object
- Source Code
- Module
- Replacing Client Module
- Business Process

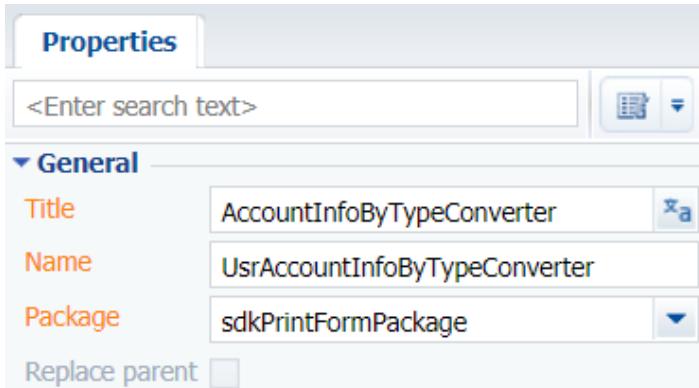
Additional

Specify the following parameters for the created object schema (Fig. 5):

- **[Title]** – "AccountInfoByTypeConverter".

- [Name] – "UsrAccountInfoByTypeConverter".

Fig. 5. – Setting up the [Source Code] type object schema



Implement a macro class for receiving additional information depending on the account type. The complete source code of the module is available below:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.CodeDom.Compiler;
    using System.Collections.Generic;
    using System.Data;
    using System.Linq;
    using System.Runtime.Serialization;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Web;
    using Terrasoft.Common;
    using Terrasoft.Core;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Entities;
    using Terrasoft.Core.Packages;
    using Terrasoft.Core.Factories;

    // An attribute with the [AccountInfoByType] macro name.
    [ExpressionConverterAttribute("AccountInfoByType")]
    // The class should implement the IExpressionConverter interface.
    class AccountInfoByTypeConverter : IExpressionConverter
    {
        private UserConnection _userConnection;
        private string _customerAdditional;
        private string _partnerAdditional;
        // Calling localizable string values
        private void SetResources() {
            string sourceCodeName = "UsrAccountInfoByTypeConverter";
            _customerAdditional = new LocalizableString(_userConnection.ResourceStorage,
sourceCodeName,
                "LocalizableStrings.CustomerAdditional.Value");
            _partnerAdditional = new LocalizableString(_userConnection.ResourceStorage,
sourceCodeName,
                "LocalizableStrings.PartnerAdditional.Value");
        }
        // Implementing the Evaluate method of the IExpressionConverter interface.
        public string Evaluate(object value, string arguments = "")
        {
            try
            {
                _userConnection =

```

```
(UserConnection) HttpContext.Current.Session["UserConnection"];
    Guid accountId = new Guid(value.ToString());
    return getAccountInfo(accountId);
}
catch (Exception err)
{
    return err.Message;
}
}
// The method for receiving additional information depending on the account type.
// As the Id input parameter of the account.
private string getAccountInfo(Guid accountId)
{
    SetResources();
    try
    {
        // Creating an EntitySchemaQuery class instance with the [Account] root
schema.
        EntitySchemaQuery esq = new
EntitySchemaQuery(_userConnection.EntitySchemaManager, "Account");
        // Adding the [Name] column from the [Type] lookup field.
        var columnType = esq.AddColumn("Type.Name").Name;
        // Adding the [Name] column from the [EmployeesNumber] lookup field.
        var columnNumber = esq.AddColumn("EmployeesNumber.Name").Name;
        // Adding the [Name] column from the [AnnualRevenue] lookup field.
        var columnRevenue = esq.AddColumn("AnnualRevenue.Name").Name;
        // The records are filtered by the account Id.
        var accountFilter = esq.CreateFilterWithParameters(
            FilterComparisonType.Equal,
            "Id",
            accountId
        );
        esq.Filters.Add(accountFilter);
        // Retrieving an entity collection.
        EntityCollection entities = esq.GetEntityCollection(_userConnection);
        // If the collection is not empty, the method will return the
corresponding
        // data depending on the account
        if (entities.Count > 0)
        {
            Entity entity = entities[0];
            var type = entity.GetTypedColumnValue<string>(columnType);
            switch (type)
            {
                case "Customer":
                    return String.Format(_customerAdditional,
entity.GetTypedColumnValue<string>(columnRevenue));
                case "Partner":
                    return String.Format(_partnerAdditional,
entity.GetTypedColumnValue<string>(columnNumber));
                default:
                    return String.Empty;
            }
        }
        return String.Empty;
    }
    catch (Exception err)
    {
        throw err;
    }
}
}
```

Populate the localizable strings of the report with the following values (table 1):

Table 1. Setting up the localizable strings

Name	English (United States)	Russian (Russia)
PartnerAdditional	Number of employees {o} persons	Number of employees {o} people
CustomerAdditional	Annual turnover {o}	Annual revenue {o}

Learn more about working with localizable strings in the "**Source code designer**" article.

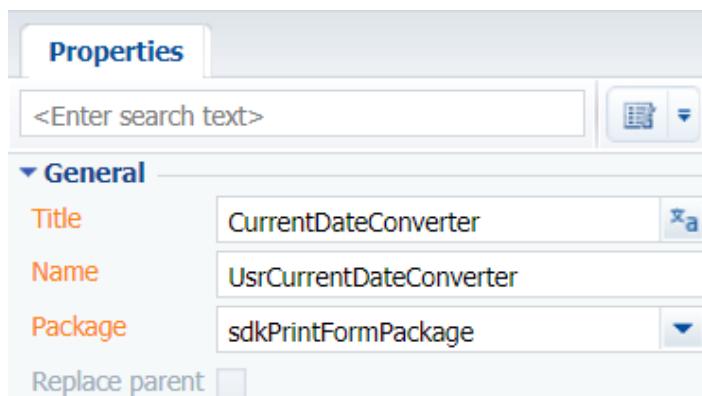
After making changes, save and publish the schema.

Go to the [Advanced settings] section → [Configuration] → Custom package → the [Schemas] tab. Click [Add] → [Source code] (Fig. 4). Learn more about creating a schema of the [Source Code] type in the "**Creating the [Source code] schema**" article.

Specify the following parameters for the created object schema (Fig. 6):

- [Title] – "CurrentDateConverter".
- [Name] – "UsrCurrentDateConverter".

Fig. 6. – Setting up the [Source Code] type object schema



Implement a macro class for retrieving the current date. The complete source code of the module is available below:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.CodeDom.Compiler;
    using System.Collections.Generic;
    using System.Data;
    using System.Linq;
    using System.Runtime.Serialization;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Web;
    using Terrasoft.Common;
    using Terrasoft.Core;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Entities;
    using Terrasoft.Core.Packages;
    using Terrasoft.Core.Factories;

    // An attribute with the [CurrentDate] macro name.
    [ExpressionConverterAttribute("CurrentDate")]
    // The class should implement the IExpressionConverter interface.
    class CurrentDateConverter : IExpressionConverter
    {
        private UserConnection _userConnection;

        // Implementing the Evaluate method of the IExpressionConverter interface.
        public string Evaluate(object value, string arguments = "")
```

```
        {
            try
            {
                _userConnection =
(HttpContext.Current.Session["UserConnection"]);
                // The method returns the current date.
                return _userConnection.CurrentUser.GetCurrentDateTime().Date.ToString("dd
MMM yyyy");
            }
            catch (Exception err)
            {
                return err.Message;
            }
        }
    }
}
```

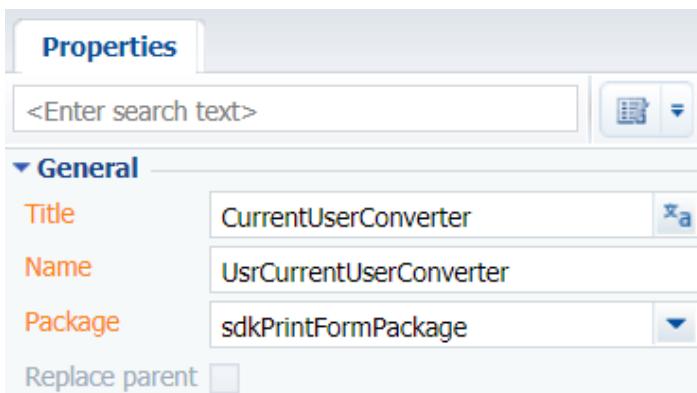
After making changes, save and publish the schema.

Go to the [Advanced settings] section → [Configuration] → Custom package → the [Schemas] tab. Click [Add] → [Source code] (Fig. 4). Learn more about creating a schema of the [Source Code] type in the "[Creating the \[Source code\] schema](#)" article.

Specify the following parameters for the created object schema (Fig. 7):

- **[Title]** – "CurrentUserConverter"
 - **[Name]** – "UsrCurrentUserConverter".

Fig. 7. – Setting up the [Source Code] type object schema



Implement a macro class for retrieving the current user. The complete source code of the module is available below:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.CodeDom.Compiler;
    using System.Collections.Generic;
    using System.Data;
    using System.Linq;
    using System.Runtime.Serialization;
    using System.ServiceModel;
    using System.ServiceModel.Web;
    using System.ServiceModel.Activation;
    using System.Text;
    using System.Text.RegularExpressions;
    using System.Web;
    using Terrasoft.Common;
    using Terrasoft.Core;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Entities;
    using Terrasoft.Core.Packages;
    using Terrasoft.Core.Factories;
```

```
// An attribute with the [CurrentUser] macro name.  
[ExpressionConverterAttribute("CurrentUser")]  
// The class should implement the IExpressionConverter interface.  
class CurrentUserConverter : IExpressionConverter  
{  
    private UserConnection _userConnection;  
    // Implementing the Evaluate method of the IExpressionConverter interface.  
    public string Evaluate(object value, string arguments = "")  
    {  
        try  
        {  
            _userConnection =  
(UserConnection)HttpContext.Current.Session["UserConnection"];  
            // The method returns the contact of the current user.  
            return _userConnection.CurrentUser.ContactName;  
        }  
        catch (Exception err)  
        {  
            return err.Message;  
        }  
    }  
}
```

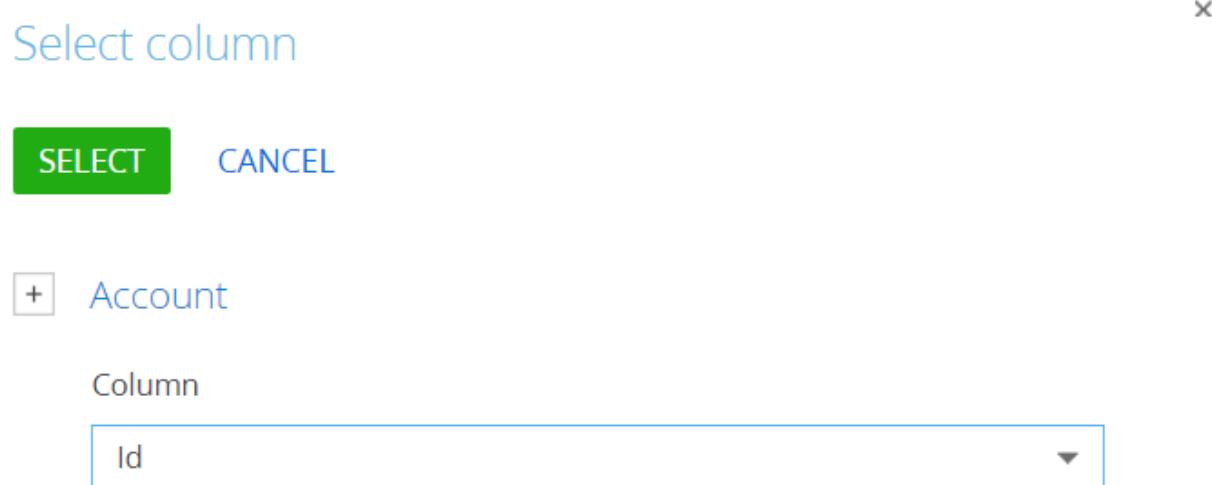
After making changes, save and publish the schema.

4. Set up the report fields

In the **[Set up report data]** block of the section working area (Fig. 2, 5), set up the fields to display in the report. To do this, click  and select the **[Id]** column in the drop-down **[Column]** list (Fig. 8). The current **[Id]** column will later be used in the custom macro to retrieve the current date.

Use the **[Id]** column as an input parameter for a custom macro.

Fig. 8. Selecting a column



Click **[Select]**.

Use the same procedure to add **[Id]** (the column will later be used in the custom macro for retrieving the current user), **[Name]**, **[Type]**, **[Primary contact]**, and **[Id]** (the column will later be used in the custom macro for receiving additional information depending on the account type) to the column template.

The list of columns after this step is presented below (Fig. 9).

Fig. 9. The list of added columns

The screenshot shows the 'Set up report data' interface. At the top left is a back arrow icon, followed by the text 'Set up report data' and an info icon (i). To the right is a '+' button. Below this is a table with five rows, each representing a column:

Column name
Id
Id
Name
Type

Below the table is another row labeled 'Primary contact' with a corresponding input field containing the value 'Id'.

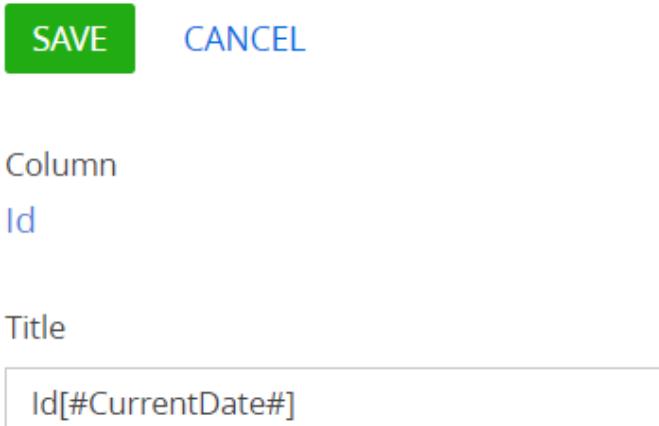
5. Attach custom macro tags to the column names

First, publish the [Source Code] type schema that implements a custom macro must. Then, add the name of the macro to the template layout. If you refresh the page in Creatio, the macro will not be printed.

Change the property of the [Id] column of an [Account] object. To do this, take the following steps:

1. In the [Set up report data] block of the section working area (Fig. 2, 5), double-click the title of the [Id] column or click  in the column title bar.
2. Change the [Id] value of the [Title] field to [Id[#CurrentDate#]] (Fig. 10). [#CurrentDate#] is a the custom macro tag for retrieving the current date.

Fig. 10. Setting up a column property



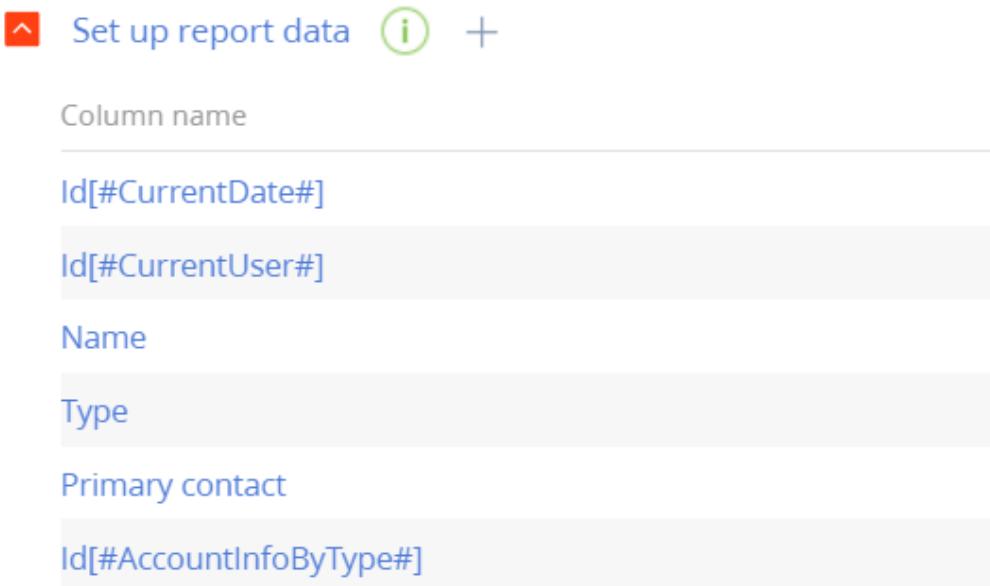
Click [Save].

Use the same procedure to add more custom macro tags to the names of other [Id] columns.

- [#CurrentUser#] – for receiving the current user.
- [#AccountInfoByType#] – for receiving additional information depending on the account type.

The list of columns after adding custom macro tags is presented below (Fig. 11).

Fig. 11. The list of added columns with macros



The screenshot shows the 'Set up report data' section. At the top are three buttons: a red arrow pointing up, 'Set up report data' (blue), and a green info icon. Below is a table with columns for 'Column name', 'Name', 'Type', and 'Primary contact'. The first row shows 'Id[#CurrentDate#]' in all four columns. The second row shows 'Id[#CurrentUser#]' in all four columns. The third row shows 'Name' in 'Name', 'Type' in 'Type', and 'Primary contact' in 'Primary contact'. The fourth row shows 'Id[#AccountInfoByType#]' in all four columns.

Column name	Name	Type	Primary contact
Id[#CurrentDate#]			
Id[#CurrentUser#]			
	Name	Type	Primary contact
			Id[#AccountInfoByType#]

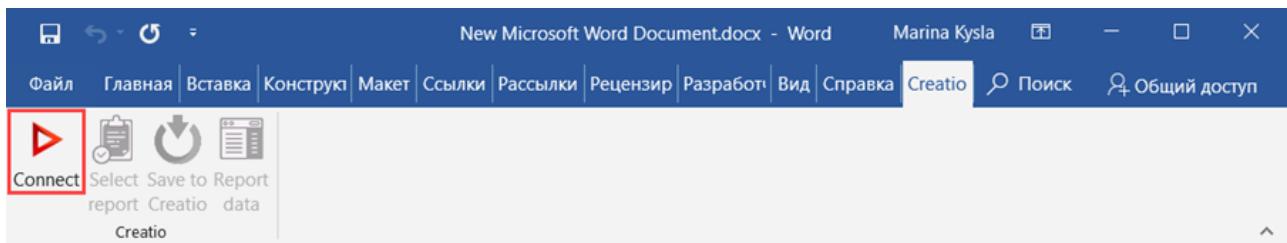
Click [Save].

6. Set up the report template layout and upload the template to Creatio

To set up the template:

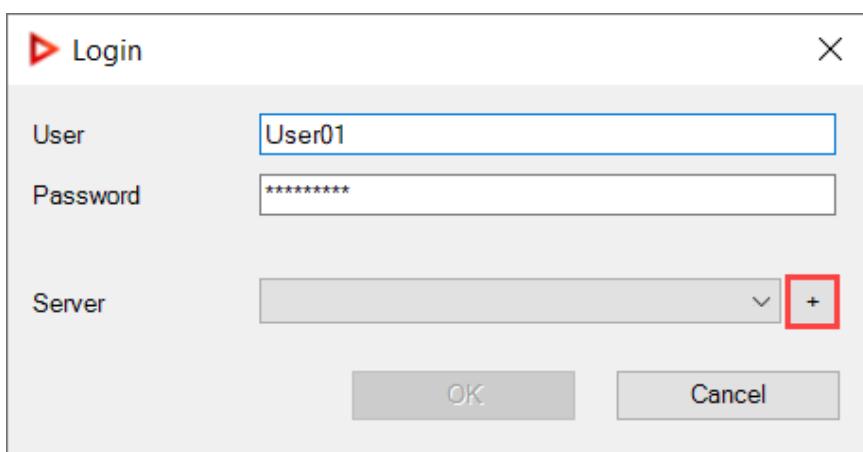
1. Open any MS Word file.
2. Click **[Connect]** on the Creatio plug-in toolbar (Fig. 12).

Fig. 12. The [Connect] button



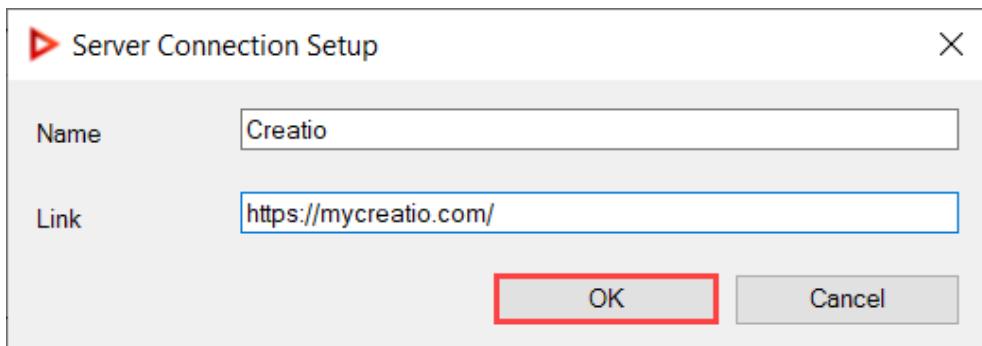
3. Enter the username and password of the Creatio user. Click **[+]** next to the **[Server]** field (Fig. 13).

Fig. 13. Authentication



4. Click **[New]**. Enter the server parameters (Fig. 14).

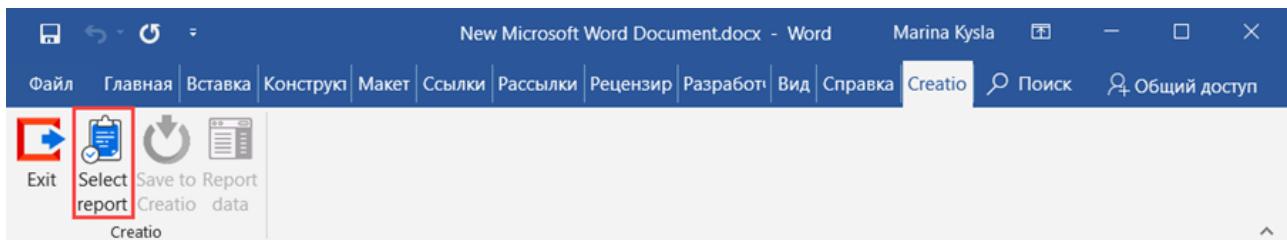
Fig. 14. Server settings



Click **[OK]**.

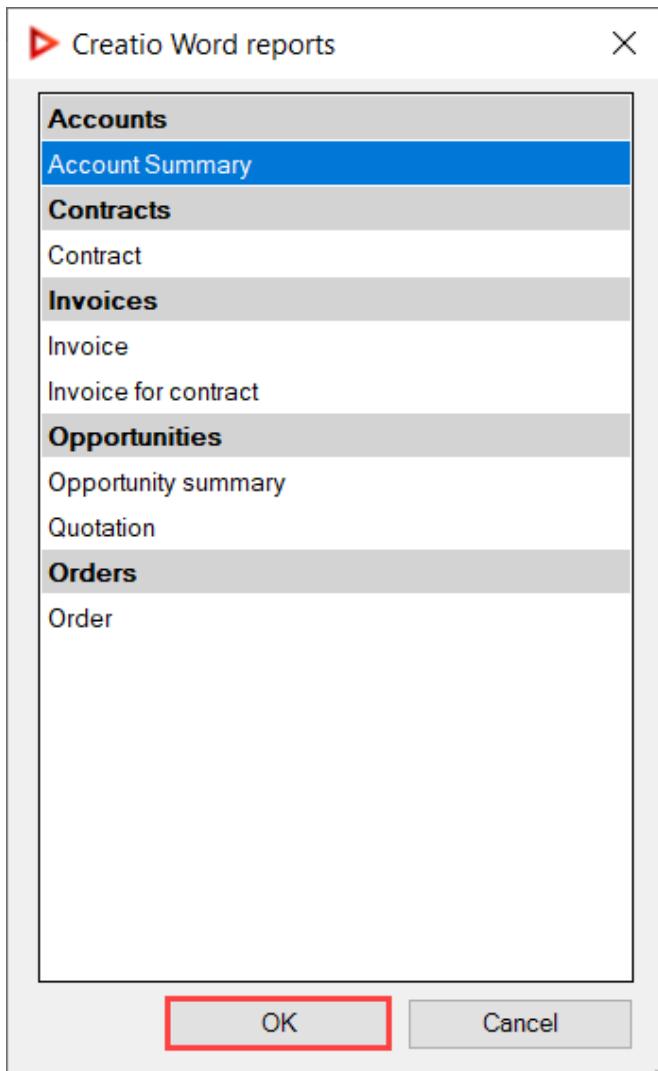
5. Click **[Select report]** on the Creatio plug-in toolbar (Fig. 15).

Fig. 15. The [Select report] button



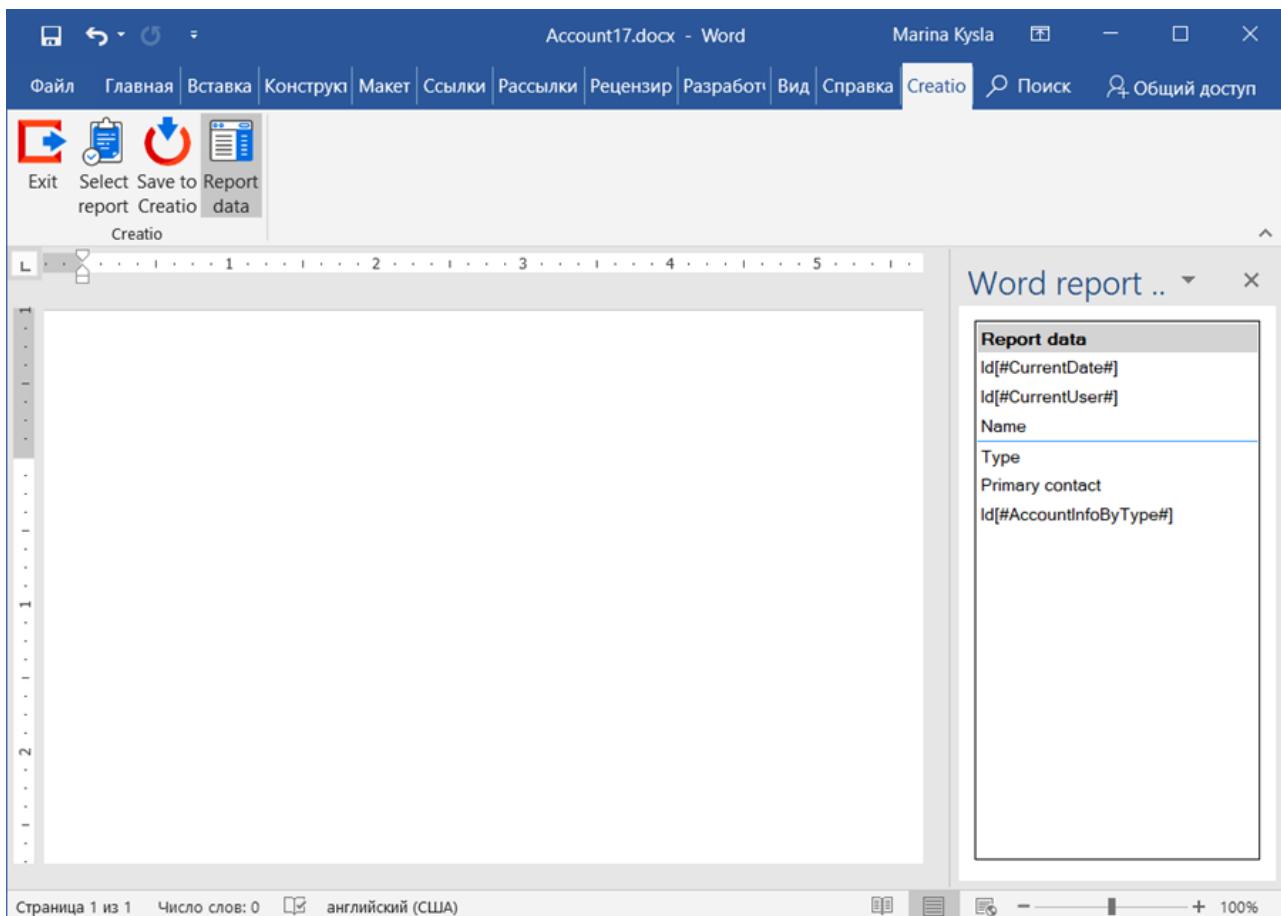
6. Select the "Account Summary" report and click **[OK]** (Fig. 16).

Fig. 16. Selecting a report



The report setup window looks as follows (Fig. 17):

Fig. 17. Setting up the report



7. Set up the template layout. Learn more about setting up a report template in the "[Design report layout via the Creatio MS Word plug-in](#)" article.

After the setup, the report looks as follows (Fig. 18):

Fig. 18. Setting up a report template

Account Summary

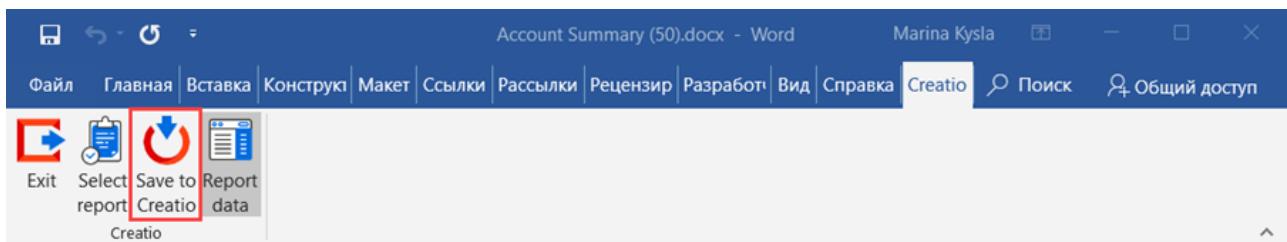
Name	«Name»
Type	«Type»
Primary contact	«Primary contact»
Additional info	«Id[#AccountInfoByType#]»

Date «Id[#CurrentDate#]»

Created by «Id[#CurrentUser#]»

8. Click **[Save to Creatio]** to load the configured report template in Creatio (Fig. 19).

Fig. 19. The [Save to Creatio] button



As a result, the "Account Summary" report will be available on the contact page under [Print] (Fig. 20).

Fig. 20. – Displaying the "Account Summary" report on a record page of the [Accounts] section

A report for accounts of the [Customer] type looks as follows (Fig. 21).

Fig. 21. A sample "Account Summary" report for [Customer] accounts

Account Summary

Name	Sunrise Investments
Type	Customer
Primary contact	Sarah M. Richards
Additional info	Annual turnover 21 – 30 million

Date

14 Jul 2020

Created by

User01

A report for accounts of the [Partner] type looks as follows (Fig. 22).

Fig. 22. A sample "Account Summary" report for [Partner] accounts

Account Summary

Name	ClearSoft Systems
Type	Partner
Primary contact	William Clarks
Additional info	Number of employees 51-100 persons

Date

14 Jul 2020

Created by

User01

Transferring the package to another development environment

To transfer the package with the report to another development environment, go to the **[Configuration]** section -> the **[Data]** tab and **bind the data** of the following elements:

- *SysModuleReport_ReportName* – the report. To bind it, use the report Id from the **[dbo.SysModuleReport]** database table.
- *SysModuleReportTable_ReportName* – the tabular component of the report. To bind it, use the report Id from the **[dbo.SysModuleReportTable]** database table.

You can view the record Id in the database table even if you do not have access to the database. To do this, display the Id system column in the window of binding data to packages.

Sales Creatio customization

Contents

- **How to change the calculation for the "Closed" column in the [Forecasts] section.**
- **Configuration of the editable columns on the product selection page**

How to change the calculation for the "Closed" column in the [Forecasts] section.

Beginner

Easy

Medium

Advanced

Introduction

Use Creatio to plan your company's sales turnover and analyze the targets. In the [Forecasts] section, you can generate forecasts using different units of measure registered in the system, and calculate the actual values. This enables to specify the time period for which you want to analyze the sales performance and monitor the overall performance of your department using the summary tables provided in the [Forecasts] section.

More information about the section can be found in the "[\[Forecasts\] section](#)" article.

Case description

Change the logic of calculation of the "Closed" column in the [Forecasts] section: the calculation should be based on invoices instead of sales.

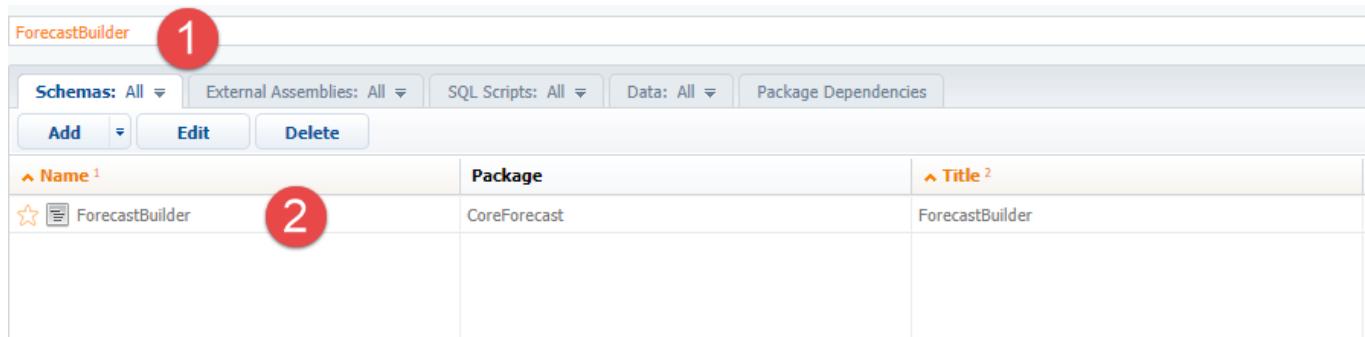
The resulting source codes of the module and stored procedure are available [by the link below](#).

Case implementation algorithm

1. Copy the source code of the forecast building module

To do this, search for the *ForecastBuilder* schema name in the [Configuration] section (Fig.1.1). Double click the found schema (Fig. 1.2) to open the module schema in the module designer (“**Module designer**”).

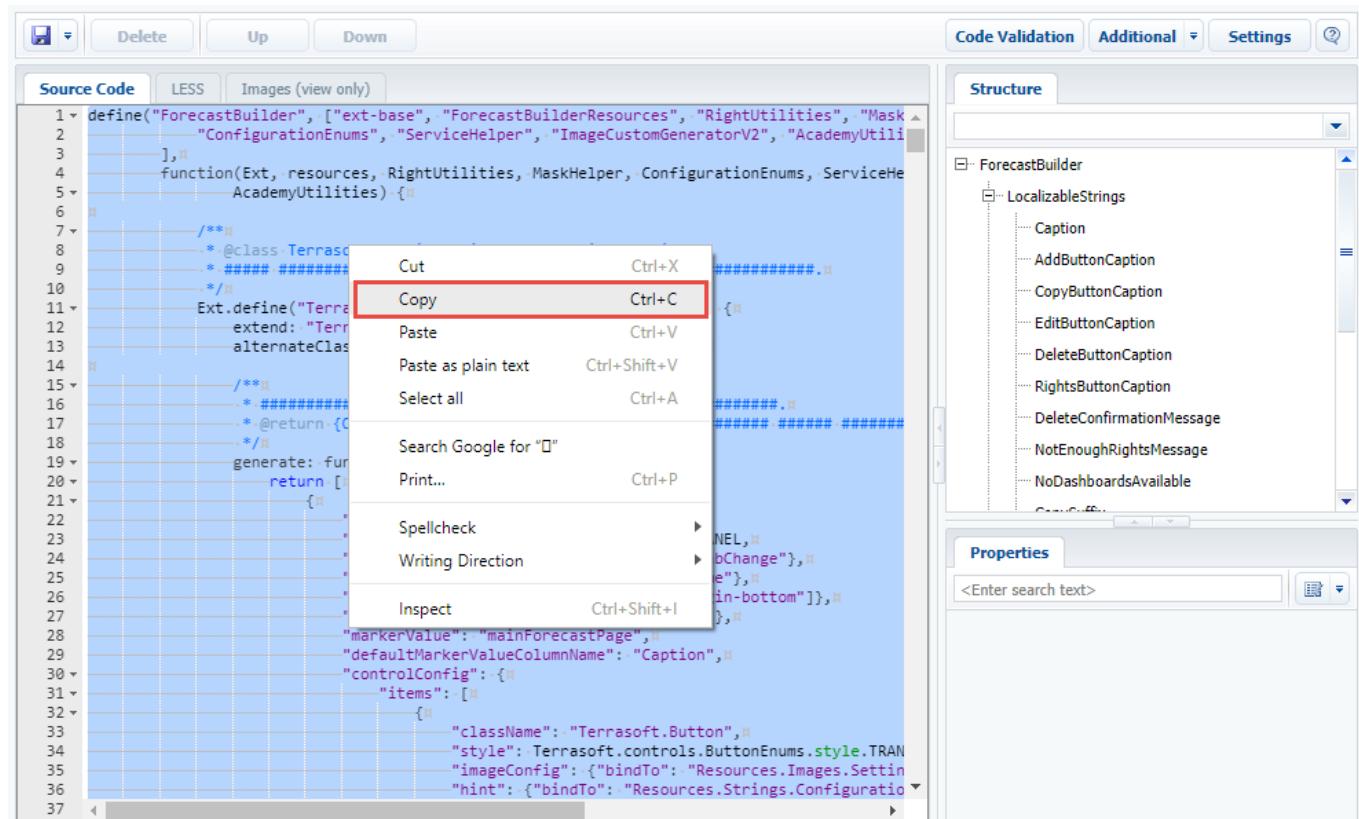
Fig. 1. Search field in the [Configuration] section



The schema is located in the pre-installed package. You will get the message that changes for this item could not be saved. The pre-installed packages are described in detail in the "**Package structure and contents**" article.

Copy all content from the [Source code] area (Fig. 2) to a text file. .

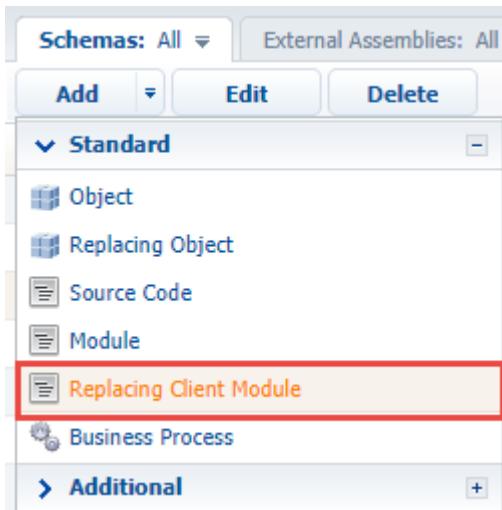
Fig. 2. Copying the source code of the schema



2. Create a replacing forecast building module.

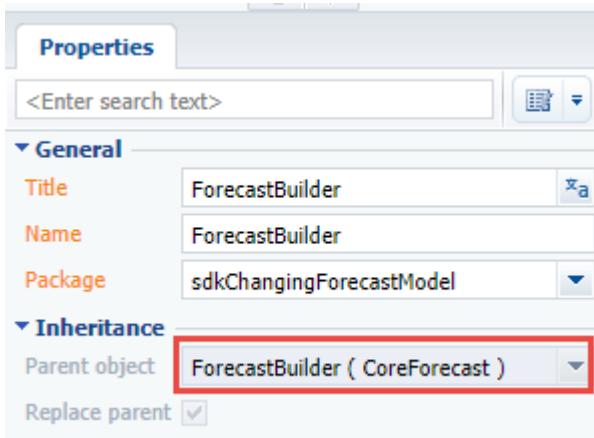
To do this, select a custom package in the [Configuration] section and execute the [Add] -> [Replacing Client Module] action on the [Schemas] tab (Fig. 3). The procedure for creating a replacing custom module is covered in the “[Creating a custom client module schema](#)”.

Fig. 3. Creating a replacing module



For the created module schema, set the “ForecastBuilder” as a [Parent object] (Fig.4). After that, all schema properties from the parent module will be applied automatically.

Fig. 4. Properties of the module schema



Add the source code of the parent schema (that was copied at the previous step) on the [Source Code] tab and save the schema.

3. Change the method of the forecast page opening

In the source code of the ForecastBuilder module schema, change the values of the *valuePairs* array in the *openForecastPage()* method of the *Terrasoft.configuration.BaseForecastsViewModel* to the values corresponding the schema of the [Invoice] object. The source code for the changes (previous values are commented out):

```
...
openForecastPage: function(moduleId, operation) {
    ...
    var valuePairs = [
        {
            name: "EntitySchemaUID",
            value: "bfb313dd-bb55-4e1b-8e42-3d346e0da7c5" //value: "ae46fb87-c02c-
4ae8-ad31-a923cdd994cf"
        },
        {
            name: "EntitySchemaName",
            value: "Invoice" //value: "Opportunity"
        }
    ];
    ...
},
```

You can get the value of the *EntitySchemaUid* Id by executing following SQL query to the Creatio database (Fig. 5):

```
select lower(UId), Name from SysSchema
Where name = 'Invoice'
and ExtendParent = 0
```

Fig. 5. Request result

	(No column name)	Name
1	bf8313dd-bb55-4e1b-8e42-3d346e0da7c5	Invoice

Save the schema to apply changes.

5. Modify the *tsp_RecalculateForecastFact* stored procedure

The *tsp_RecalculateForecastFact* stored procedure recalculates the values of the “Closed” column for selected time period. Executing and applying changes in the stored procedures is described in the [“How to: Modify a Stored Procedure \(SQL Server Management Studio\)”](#) MSDN article.

Pay high attention when creating and executing the SQL query. Executing an incorrect SQL query can damage existing data and disrupt the system.

To use invoices for calculation, make following modification to the procedure (previous values are commented out).

1. Change the value stored in the @CompletedId variable.

```
--SET @CompletedId = '{60D5310C-5BE6-DF11-971B-001D60E938C6}'
SET @CompletedId = '{698D39FD-52E6-DF11-971B-001D60E938C6}'
```

This variable stores the Id of the status of a paid invoice. You can get the value of the variable by executing following SQL query to the Creatio database (Fig. 6):

```
select Id, Name from InvoicePaymentStatus
where Name = 'Paid'
```

Fig. 6. Request result

	Id	Name
1	698D39FD-52E6-DF11-971B-001D60E938C6	Paid

2. Change the query which result is stored in the @MaxDueDate variable.

```
--SET @MaxDueDate = (SELECT Convert(Date, MAX(DueDate), 104) FROM Opportunity o WHERE
o.StageId = @CompletedId)
SET @MaxDueDate = (SELECT Convert(Date, MAX(StartDate), 104) FROM Invoice o WHERE
o.PaymentStatusId = @CompletedId)
```

The query searches for the newest paid invoice.

3. Change the subquery expression stored in the @SQLText variable. In this subquery the logic of calculating the “Closed” and “Pipeline” columns is implemented.

```
--Initial value
/*SET @SQLText = N'
    SELECT
        (SELECT SUM(ISNULL(fiv.[Value], 0))
        FROM [ForecastItemValue] fiv
        WHERE fiv.[ForecastItemId] = @P5
        AND fiv.[PeriodId] = @P6
        AND fiv.[ForecastIndicatorId] = @P7
        ) PlanAmount,
        (SELECT SUM(ISNULL(o.[Amount], 0))
        FROM [Opportunity] o
        WHERE o.[StageId] = @P1
        AND o.[DueDate] >= @P2
        AND o.[DueDate] < @P3
        AND o.' + @ColumnName + N' = @P4
        ) FactAmount,
        (SELECT SUM(ISNULL(o.[Amount], 0) * ISNULL(o.[Probability], 0) / 100)
        FROM [Opportunity] o
        INNER JOIN [OpportunityInStage] ois ON ois.[OpportunityId] = o.[Id]
        INNER JOIN [OpportunityStage] os ON os.[Id] = ois.[StageId]
        WHERE os.[End] = 1
        AND ois.[DueDate] >= @P2
        AND ois.[DueDate] < @P3
        AND ois.[Historical] = 0
        AND o.' + @ColumnName + N' = @P4
        ) PotentialAmount*/
--New value
SET @SQLText = N'
    SELECT
        (SELECT SUM(ISNULL(fiv.[Value], 0))
        FROM [ForecastItemValue] fiv
        WHERE fiv.[ForecastItemId] = @P5
        AND fiv.[PeriodId] = @P6
        AND fiv.[ForecastIndicatorId] = @P7
        ) PlanAmount,
        (SELECT SUM(ISNULL(o.[Amount], 0))
        FROM [Invoice] o
        WHERE o.[PaymentStatusId] = @P1
        AND o.[StartDate] >= @P2
        AND o.[StartDate] < @P3
        AND o.' + @ColumnName + N' = @P4
        ) FactAmount,
        (SELECT SUM(ISNULL(o.[Amount], 0))
        FROM [Invoice] o
        INNER JOIN [InvoicePaymentStatus] os ON os.[Id] = o.[PaymentStatusId]
        WHERE os.[FinalStatus] = 0
        AND o.[StartDate] >= @P2
        AND o.[StartDate] < @P3
        AND o.' + @ColumnName + N' = @P4
        ) PotentialAmount'
```

Run the SQL script (F5 key) to apply the changes.

As a result, the calculation of “Closed” and “Potential” columns will be based on invoices (Fig. 8) instead of sales (Fig. 7).

Fig. 7. Calculating the “Closed” columns by sales

		1 Expected	April 2017 Actual	2 Closed, %	Pipeline
Alpha Business		10,000	12,520	125	0
Axiom		5,000	6,200	124	0
Fast Works		0	0	0	0

Fig. 8. Calculating the “Closed” columns by invoices

		1 Expected	April 2017 Actual	2 Closed, %	Pipeline
Alpha Business		10,000	11,300	113	19,502
Axiom		5,000	0	0	15,350
Fast Works		0	0	0	0

The resulting source codes of the module and stored procedure can be downloaded from the [link](#).

Configuration of the editable columns on the product selection page

Beginner Easy Medium Advanced

Introduction

In the Creatio version 7.11.2 or higher the editable columns are available on the [product selection page](#). By default the [Quantity], [Unit of measure] and [Price] columns are available for edit. You can also make other columns editable.

Case description

Make editable the [Discount, %] column on the product selection page in the [Orders] section. Add and make editable the [Custom price] column.

The case can be also done for the product selection page in the [Invoices] section.

Case implementation algorithm

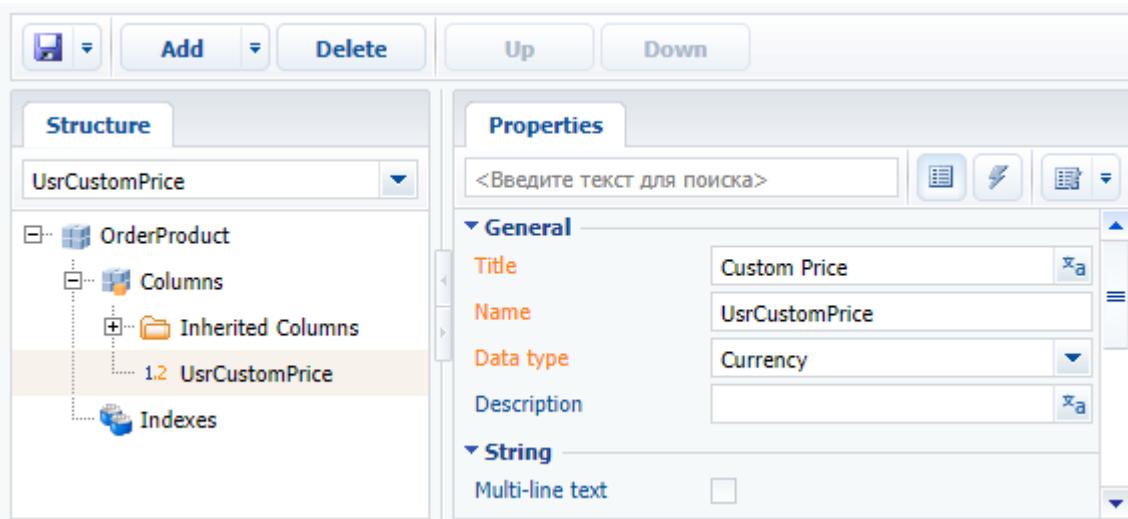
1. Add a custom column to the [Product in order] object

For this, create the [Product in order] replacing object and add a column to it. Creating replacing object and adding a custom column is described in the “[Creating the entity schema](#)”.

Set following properties for the added column (Fig. 1):

- [Title] – “Custom Price”
- [Name] – "UsrCustomPrice"
- [Data type] – “Currency.”

Fig. 1. Properties of the custom column



Publish the object schema to apply changes.

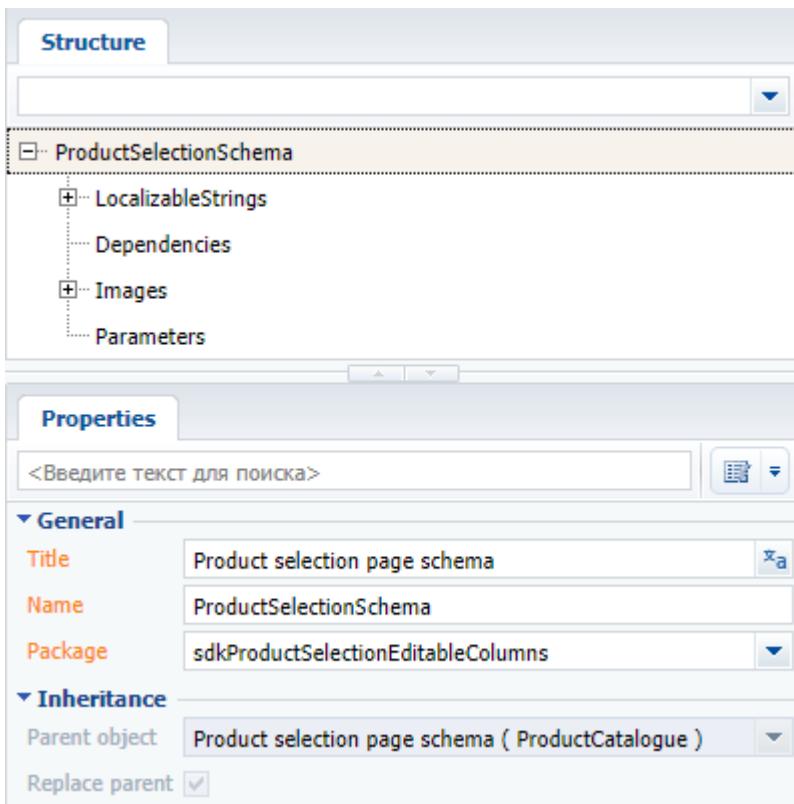
To implement case in the [Invoices] section, perform above steps for the [Product in invoice] object.

2. Create a replacing custom module for the product selection page schema

The procedure of creating a replacing custom module is covered in the “[Creating client schema](#)”. Set following properties for the for the created module (Fig. 2):

- [Title] – [Product selection page schema]
- [Name] – "ProductSelectionSchema"
- [Parent object] – [Product selection page schema].

Fig. 2. Properties of the replacing client module



Add the following source code on the [Source Code] tab of the schema designer:

```
define("ProductSelectionSchema", [],
  function() {
    return {
      methods: {
        getEditableColumns: function() {
          // Getting an array of editable columns.
          var columns = this.callParent(arguments);
          // Adding the [Discount, %] column to the array of editable
          columns.push("DiscountPercent");
          // Adding a custom column.
          columns.push("UsrCustomPrice");
          return columns;
        },
        setColumnHandlers: function(item) {
          this.callParent(arguments);
          // Bind the event handler of the user column change event.
          item.on("change:UsrCustomPrice", this.onCustomPriceChanged,
this);
        },
        // A handler method that will be called when the field value is
        changed.
        onCustomPriceChanged: function(item, value) {
          window.console.log("Changed: ", item, value);
        }
      };
    });
});
```

Save the schema to apply changes.

3. Configure columns display on the product selection page

Configure the columns to display them on the product selection page, (see "[Setting up columns](#)"). In title view configuration of the list, add the [Discount, %] and [Custom price] columns.

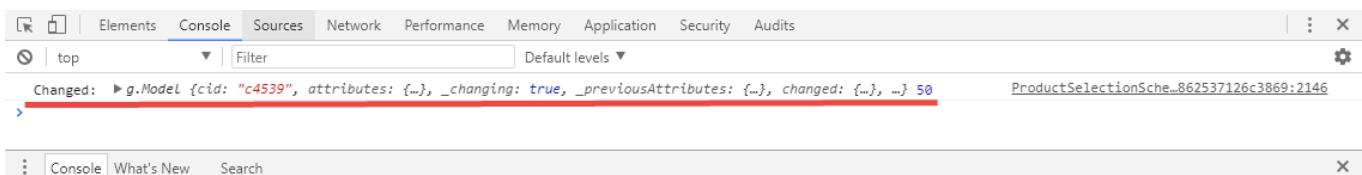
As a result, two editable columns will be displayed on the product selection page (Fig. 3).

Fig. 3. Case result

Code	Name	Price, rub.	Quantity	Unit of measure	Discount, %	Custom Price
116652	Laptop UZ155VN-P529C	80,000.00	2.000	pieces	3.00	1 50.00 2

After modification of the value in the [Custom price] column, the corresponding message will be displayed in the browser console (Fig. 4).

Fig. 4. The result in the browser console



Service Creatio customization

Contents

- **Adding a new rule for calculating case deadline**
- **Adding the macro handler to the email template**
- **How to hide feed area in the agent desktop**
- **Adding floating icons for internal case feed posts**

Adding a new rule for calculating case deadline

Beginner Easy Medium **Advanced**

Introduction

Creatio enables implementing custom logic of receiving parameters for calculating case deadline. When calculating or recalculating a case deadline, a developer implemented strategy is used instead of one of the base calculation strategies.

You can select a specific calculation rule in the [Case deadline calculation rules] lookup. Follow these steps to add a new calculation rule:

1. Create an object schema and add columns necessary for storage of response and resolution deadlines, links to the calendar, service agreement and service.
2. Based on the created object schema, add a lookup and populate it with values needed to calculate the deadline parameters.
3. Add the source code schema and declare the class inherited from the *BaseTermStrategy* abstract class. Implement custom mechanism of receiving response and resolution deadline parameters in the class.
4. Add a new rule.

Case description

Add a custom rule for calculating case deadline parameters for the [Lost data recovery] service as per the [78 – Elite Systems] agreement. Set the following values for the new rule:

- response time – 2 working hours

- resolution time – 1 working day
- used calendar – [Default calendar]

Source code of the case:

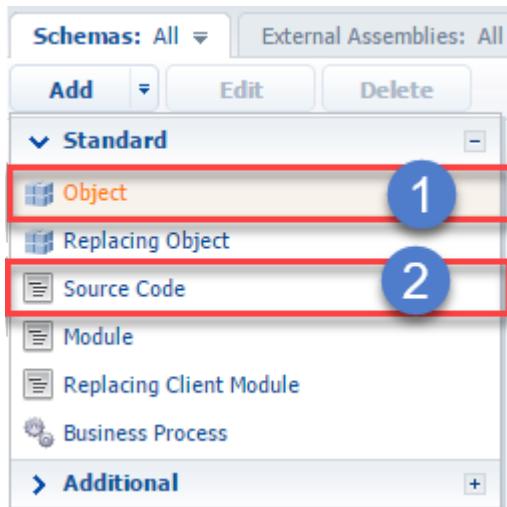
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Creating an object schema containing the necessary columns for calculation

Perform the [Add] – [Object] action on the [Schemas] tab of the [Configuration] section.

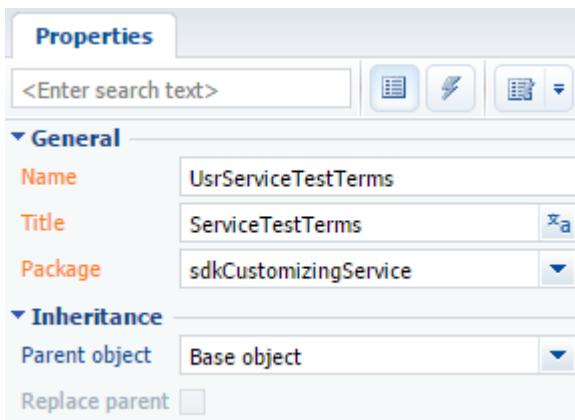
Fig. 1. Adding the schema



Set the following properties for the created object schema (Fig. 2):

- [Name] – “UsrServiceTestTerms”
- [Title] – “ServiceTestTerms”
- [Parent object] – the [Base object] schema

Fig. 2. Properties of the added object schema



In the created schema, create a number of columns, whose primary properties are listed in table 1.

Table 1. Properties of the added columns

Name	Title	Type	Description
UsrReactionTimeUnit	Response time unit	The [Time unit] lookup	Specifies the time unit (calendar days, hours, etc.) that will be used for calculating the [Response time] parameter.

Name	Title	Type	Description
<i>UsrReactionTimeValue</i>	Response time value	Integer	A column for storage the response time value.
<i>UsrSolutionTimeUnit</i>	Response time unit	The [Time unit] lookup	Specifies the time unit (calendar days, hours, etc.) that will be used for calculating the [Response time] parameter.
<i>UsrSolutionTimeValue</i>	Resolution time	Integer	A column for storage the response time value.
<i>UsrCalendarId</i>	Calendar that is used	The [Calendar] lookup	The calendar used for calculating the case deadline.
<i>UsrServicePactId</i>	Service agreement	The [Service agreement] lookup	Link to the [Service agreement] object. Added for enabling filtration.
<i>UsrServiceItemId</i>	Service	The [Service] lookup	Link to the [Service] object. Added for enabling filtration.

Publish the schema after adding the columns.

2. Adding a lookup and populating it with values needed to calculate the deadline parameters

Provide specific values to calculate the case response and resolution deadline. To do this, add a lookup with the following values based on the added schema (fig.3):

- [Name] – “Custom response and resolution deadlines”
- [Object] – ServiceTestTerms

Fig. 3. Properties of the added lookup

Name * Custom response and resolution deadlines

Object * ServiceTestTerms

List page

Description

Add a record with the following data to the added lookup (as per the case conditions) (fig.4):

Fig. 4. A record in the created lookup that meets the case conditions

Response time un...	Respo...	Resolve time unit	Resolu...	Service	Calendar that is us...	Service agreement
Calendar hours	2	Calendar days	1	Lost data recovery	Default calendar	78 — Elite Syst...

3. Implementing a class with the mechanism of receiving deadline parameters

Add the source code schema (fig.1, 2) Add the class inherited from the *BaseTermStrategy* abstract class (declared in the *Calendar* package) to the schema source code. Implement a parameterized constructor with the following parameters in the class:

- *UserConnection userConnection* – user current connection
- *Dictionary<string, object> args* – arguments that are the base of performing calculation

Implement the *GetTermInterval()* abstract method declared in the base class. This method accepts the mask of populated values as the incoming parameter, which is the base of taking a decision about populating the specific deadline parameters of the *TermInterval* returned class implementing the *ITermInterval<TMask>* interface.

The complete schema source code:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
    using Terrasoft.Common;
    using Terrasoft.Configuration.Calendars;
    using Terrasoft.Core;
    using Terrasoft.Core.Entities;
    using CalendarsTimeUnit = Calendars.TimeUnit;
    using SystemSettings = Terrasoft.Core.Configuration.SysSettings;
    public class ServiceTestTermsStrategy: BaseTermStrategy<CaseTermInterval,
CaseTermStates>
    {
        // Container class for storage of data received from the entrance point.
        protected class StrategyData
        {
            public Guid ServiceItemId {
                get;
                set;
            }
            public Guid ServicePactId {
                get;
                set;
            }
        }
        // The field for storage of data received from the entrance point.
        protected StrategyData _strategyData;
        // Parameterized constructor necessary for the correct
        // initialization by selector class.
        public ServiceTestTermsStrategy(UserConnection userConnection,
Dictionary<string, object> args)
            : base(userConnection) {
            _strategyData = args.ToObject<StrategyData>();
        }
        // Method that receives data and returns them in the CaseTermInterval class
instance.
        public override CaseTermInterval GetTermInterval(CaseTermStates mask) {
            var result = new CaseTermInterval();
            // Creating the EntitySchemaQuery query.
            var esq = new EntitySchemaQuery(UserConnection.EntitySchemaManager,
"UserServiceTestTerms");
            // Adding columns to the query.
            string reactionTimeUnitColumnName =
esq.AddColumn("UsrReactionTimeUnit.Code").Name;
            string reactionTimeValueColumnName =
esq.AddColumn("UsrReactionTimeValue").Name;
            string solutionTimeUnitColumnName =
esq.AddColumn("UsrSolutionTimeUnit.Code").Name;
```

```

        string solutionTimeValueColumnName =
esq.AddColumn("UsrSolutionTimeValue").Name;
        string calendarColumnName = esq.AddColumn("UsrCalendarId.Id").Name;
        // Adding filters to the query.
        esq.CreateFilterWithParameters(FilterComparisonType.Equal,
"UserServiceItemId", _strategyData.ServiceItemId);
        esq.CreateFilterWithParameters(FilterComparisonType.Equal,
"UserServicePactId", _strategyData.ServicePactId);
        // Execution and processing of query results.
        EntityCollection entityCollection =
esq.GetEntityCollection(UserConnection);
        if (entityCollection.IsEmpty()) {
            // Adding response time to the nurtured value.
            if (!mask.HasFlag(CaseTermStates.ContainsResponse)) {
                result.ResponseTerm = new TimeTerm {
                    Type =
entityCollection[0].GetTypedColumnValue<CalendarsTimeUnit>
(reactionTimeUnitColumnName),
                    Value = entityCollection[0].GetTypedColumnValue<int>
(reactionTimeValueColumnName),
                    CalendarId = entityCollection[0].GetTypedColumnValue<Guid>
(calendarColumnName)
                };
            }
            // Adding resolution time to the nurtured value.
            if (!mask.HasFlag(CaseTermStates.ContainsResolve)) {
                result.ResolveTerm = new TimeTerm {
                    Type =
entityCollection[0].GetTypedColumnValue<CalendarsTimeUnit>
(solutionTimeUnitColumnName),
                    Value = entityCollection[0].GetTypedColumnValue<int>
(solutionTimeValueColumnName),
                    CalendarId = entityCollection[0].GetTypedColumnValue<Guid>
(calendarColumnName)
                };
            }
        }
        return result;
    }
}
}

```

Publish the schema after adding the source code.

4. Adding the new rule

Add a value to the [Case deadline calculation schemas] lookup. In the [Handler] column, specify the full qualified name of the created class (specifying the namespaces).

In the [Alternative schema] column you may specify the rule for calculating the deadline in case calculation by current rule is not possible. Take into considerations that if any of deadline parameters is not calculated by strategy class, a class instance of an alternative strategy will be created. In case the alternative strategy cannot calculate the deadline either, another alternative strategy will be created, thus forming a rule queue.

Select the [Default] checkbox for the added record.

See an example of an added record to the [Case deadline calculation schemas] lookup in fig.5.

Fig. 5. A record of a custom deadline calculation rule

Lookups

What can I do for you? >

Actions		New	Close	ACTIONS ▾
Case deadline calculation schemas				
Filter ▾				
Name	Description	Handler	Default	Alternative schema
Strategy for 78 – Elite systems		Terrasoft.Configuration.ServiceTestTermsStrategy	<input checked="" type="checkbox"/>	
By priority		Terrasoft.Configuration.CaseTermStrategyByPriority	No	
By service		Terrasoft.Configuration.CaseTermStrategyByService	No	

As a result, new response and resolution deadline calculation rules will be applied for cases per the [78 – Elite Systems] agreement for the [Lost data recovery] service.

Fig. 6. Case result

The screenshot shows the Creatio Case #SR00000004 interface. On the left, there's a sidebar with various filters like Resolution time (4/12/2017, 12:22 PM, 1d 00:00), Priority (Medium), Contact, Account (Elite Systems), SLA (78 – Elite Systems), Category, Incident, Service (selected), Configuration item, and Assignees group. The main area has tabs: PROCESSING, CLOSURE AND FEEDBACK (highlighted), CASE INFORMATION, ATTACHMENTS, and FEED. The CASE INFORMATION tab shows fields: Subject (Lost data), Description, Source (Call), Support line (1st-line support). Under Terms, it shows Registration date (4/11/2017, 12:22 PM) and Response time (4/11/2017, 2:22 PM). It also shows Resolution time (4/12/2017, 12:22 PM), First resolution time, Actual response time, Actual resolution time, Remaining time (02:00), and Remaining duration (1d 00:00). The Response time and Resolution time fields are highlighted with red boxes.

Adding the macro handler to the email template

Beginner Easy Medium Advanced

Introduction

The email templates can be used for communication with a service team. They are available in the [Email templates] lookup. More information about email templates setup can be found in the “[Automatic emailing setup](#)”.

For example, the [Case closure notification] template is used to notify the user that the case has been closed. A number of pre-set macros can be used to display the values of some system object columns in the emails (for example contact's title or job).

Creatio enables to implement a custom logic of adding values that is returned by the macro handler. During the

processing of the macro the Creatio will execute the algorithm implemented by the developer instead of the basic logic.

The `@Invoke` tag points on the specific handler of a class. After that, the name of the class that implements the `IMacrosInvokable` interface with the `GetMacrosValue()` method should be specified after “.”. This method will return the string that replaces the macro string.

To implement custom macro handler:

1. Create class that implements the `IMacrosInvokable` interface.
2. Register a macro in the `EmailTemplateMacros` table by specifying the `ParentId` (base template with the `@Invoke` tag) and the `ColumnPath` (class name).
3. Add a macro to the email template

Case description

Add the email macro handler that will return the “Test” string.

Source code

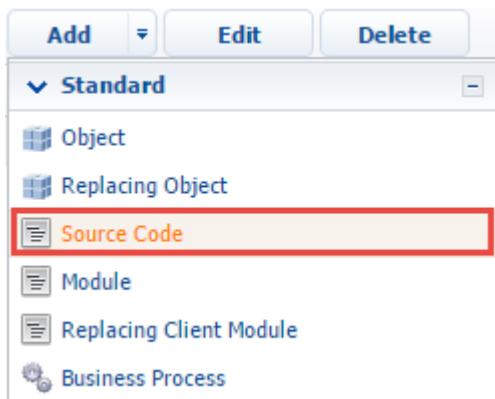
You can download the package with case implementation using the following [link](#).

Case implementation algorithm

1. Creating the class which implements the `IMacrosInvokable` interface.

To create class implementing the `IMacrosInvokable` interface, add the [Source Code] schema to the development package. For this, execute the [Add] – [Source Code] menu command on the [Schemas] tab in **the [Configuration]** section (Fig.1).

Fig. 1. Adding the [Source Code] schema



For the selected schema specify:

- [Title] – “Text string generator”.
- [Name] – “UsrTestStringGenerator”.

Source code of the schema:

```
namespace Terrasoft.Configuration
{
    using System;
    using Terrasoft.Core;
    // The class of the macro handler for the Email template.
    public class UsrTestStringGenerator : IMacrosInvokable
    {
        // A user connection.
        public UserConnection UserConnection {
            get;
            set;
        }
    }
}
```

```
// A method that returns an substitution value.
public string GetMacrosValue(object arguments) {
    return "Test";
}
}
```

Publish the schema.

2. Macro registration in the EmailTemplateMacros table

To register macro in the *EmailTemplateMacros*, execute the following SQL request:

```
INSERT INTO EmailTemplateMacros (Name, Parentid, ColumnPath)
VALUES (
    'UsrTestStringGenerator',
    (SELECT TOP 1 Id
     FROM EmailTemplateMacros
     WHERE Name = '@Invoke'),
    'Terrasoft.Configuration.UsrTestStringGenerator'
)
```

3. Adding a macro to the email template

After registering the macro you can use it in the email templates. For this, modify one or several records of the [Email templates] lookup (Fig. 2).

Fig. 2. The [Email templates] lookup

Record	Subject
Case resolution notification	Case #[#Number#] "[#Subject#]" is resolved
Case assigned to group	Case #[#Number#] "[#Subject#]" assigned to group
"Offer of the day" template	Offer of the day
Template for new order approval notification	Approval request

For example to modify the content of the emails sent at case resolution you need to change the [Case resolution notification] record. If you add the `[#@Invoke.UsrTestStringGenerator#]` macro to the template (Fig. 3), the “Test” value will be displayed instead of the macro in the email sent to the customer.

Fig. 3. Macro in the email template

Hello, [#Contact.Name#!]
 Your [#Category.Name#] No.#[#Number#] "[#Subject#]" has been resolved.
 [#Solution#]
Actual resolution time: [#SolutionProvidedOn#].
 If the provided resolution is not good enough, please respond to this email.
 Please rate the quality of customer support service
 [#@Invoke.UsrTestStringGenerator#]
 Best regards,
 [#Owner.Name#]

How to hide feed area in the agent desktop

Beginner **Easy** **Medium** **Advanced**

The feed area of the [Agent desktop] section used to notify helpdesk or contact center agents about noteworthy events of the company (Fig. 1).

Fig. 1. Agent desktop feed area

Number	Registration date	Resolution time
SR_206	7/6/2017 12:14 PM	7/6/2017 5:14 PM
SR_199	7/5/2017 7:35 PM	7/7/2017 8:00 PM
SR_202	7/5/2017 11:35 AM	7/7/2017 8:00 PM
SR_203	7/4/2017 6:30 PM	7/7/2017 4:00 PM
SR_205	7/4/2017 11:30 AM	7/6/2017 8:00 PM
SR_192	7/1/2017 12:15 PM	7/3/2017 2:00 AM

To hide feed area:

1. Create the [Agent desktop page] replacing schema in the custom package. The procedure for creating a replacing client schema is covered in the “**Creating a custom client module schema**” article.
2. Add the following source code to the schema:

```
define("OperatorSingleWindowPage", [],
  function() {
    return {
      methods: {
        // Replacing the base method to exclude the ENSFeedModule feed module
      }
    }
  }
)
```

from the loaded modules.

```

        loadContent: function() {
            // the ESNFeedModule module does not need to be loaded because
            The centerContainer container is removed
            //this.loadModule("ESNFeedModule", "centerContainer");
            this.loadModule("SectionDashboardsModule", "rightContainer");
            this.loadModule("OperatorQueuesModule", "leftContainer");
        }
    },
    diff: /**SCHEMA_DIFF*/ [
    {
        "operation": "remove",
        "name": "centerContainer"
    }
] /**SCHEMA_DIFF*/
};

}

);

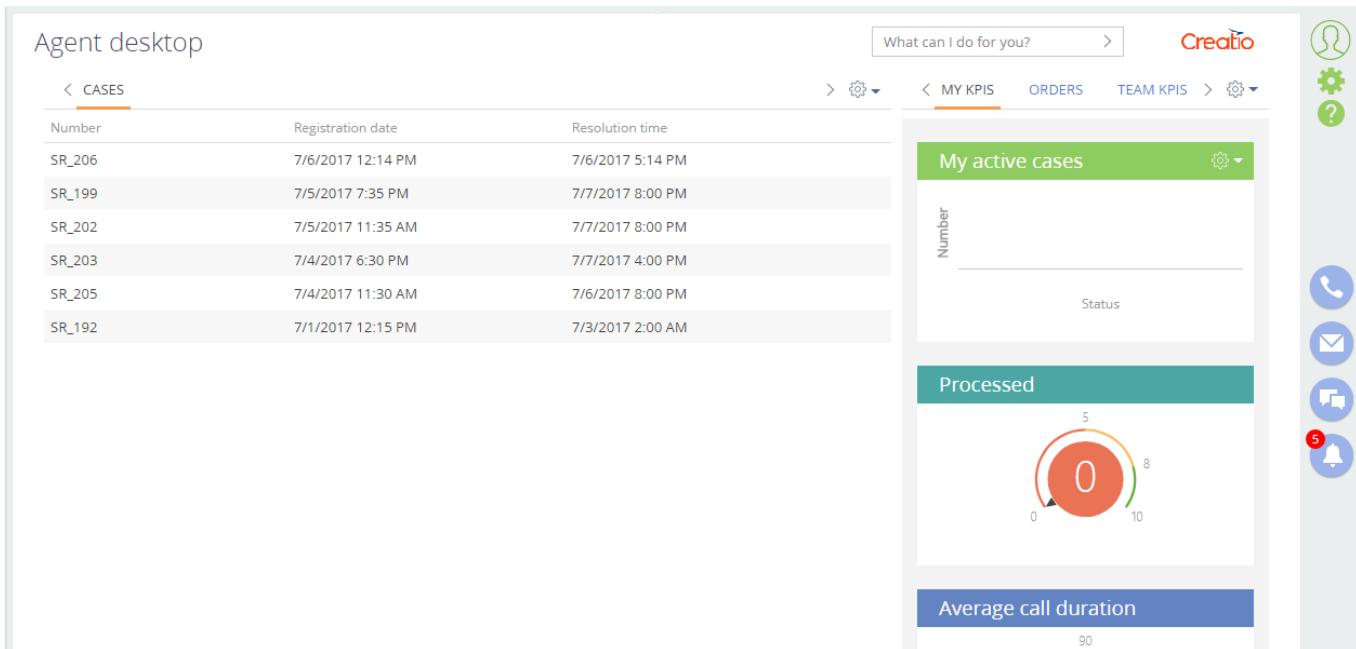
```

3. Save the changes.

4. Refresh the browser page.

As a result the feed area in the agent desktop will be hidden (Fig. 2).

Fig. 2. [Agent desktop] section without feed area



Adding floating icons for internal case feed posts

Beginner

Easy

Medium

Advanced

Introduction

In Creatio 7.12.0 you can quickly add a new case based on existing case communication email thread. Select a text in

an email from the case message history and click the button. The values of the source message fields whose [Make copy] checkbox is selected in the section wizard will be copied to the new case. Creatio will automatically reply to the email with a standard case registration notification. Adding cases based on an email text works both for emails and portal posts.

This function was implemented via the *SelectionHandlerMultiLineLabel* control element located in the *Message* package.

To process the [Processing] tab posts added to the internal feed:

1. Create the SocialMessageHistoryItemPage replacing schema.
2. Implement the selected text processing logic via the *selectedTextChanged* and *selectedTextButtonClick* events of the *SelectionHandlerMultiLineLabel* control element.
3. Add a configuration object with the *SelectionHandlerMultiLineLabel* element settings to the diff array.

Case description

Add the floating icon function when selecting a text from the internal case feed posts on the [Processing] tab of the [Cases] section. A new case with automatically populated fields should be added upon clicking the floating icon.

Source code

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

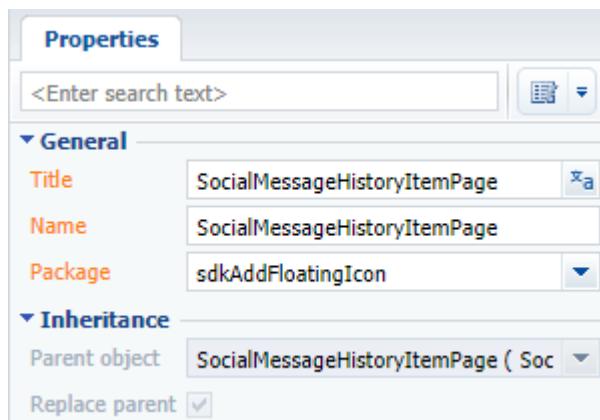
You can set up the package only for the *Service* line products or for product lines containing the *Message* and *SocialMessage* packages.

Case implementation algorithm

1. Create the SocialMessageHistoryItemPage replacing schema.

Create a replacing client module and specify the *SocialMessageHistoryItemPage* as parent object (Fig. 1). Creating a replacing page is covered in the “[Creating a custom client module schema](#)” article.

Fig. 1. Properties of the replacing module



2. Implement the selected text processing logic

Add the following methods to the created schema method collection (the source code is provided below):

- *getMessageFromHistory()* – an overridden base method. Receives the selected post subject.
- *onSelectedTextChanged()* – sets the selected text value to the *HighlightedHistoryMessage* attribute. Triggered upon text selection.
- *onSelectedTextButtonClick()* – adds a case whose subject is received from the previously installed *HighlightedHistoryMessage* attribute. The logic of adding a case is defined in the *BaseMessageHistory* parent schema. Triggered upon clicking the floating icon.

3. Set up the SelectionHandlerMultiLineLabel element.

Add the configuration object with the *SelectionHandlerMultiLineLabel* element settings to the *diff* array of the created schema. The replacing schema source code is as follows:

```
define("SocialMessageHistoryItemPage", ["SocialMessageConstants",
"css!SocialMessageHistoryItemStyle"],
```

```
function(socialMessageConstants) {
    return {
        // Name of the edit page object schema.
        entitySchemaName: "BaseMessageHistory",
        details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
        // Edit page view model methods.
        methods: {
            // Overridden base method. Receives subject for the selected post.
            getMessageFromHistory: function() {
                var message = this.get("HighlightedHistoryMessage");
                if (this.isHistoryMessageEmpty(message)) {
                    message = this.get("[Activity:Id:RecordId].Body");
                }
                return message;
            },
            // Text selection event handler.
            onSelectedTextChanged: function(text) {
                this.set("HighlightedHistoryMessage", text);
            },
            // Floating icon clicking handler.
            onSelectedTextButtonClick: function() {
                // Preparing case data from histroy.
                this.prepareCaseDataFromHistory();
            }
        },
        diff: /**SCHEMA_DIFF*/[
            {
                // Change the existing "MessageText" component.
                "operation": "merge",
                // Component name.
                "name": "MessageText",
                // Object properties.
                "values": {
                    // View generator properties.
                    "generator": function() {
                        return {
                            // HTML-tag id value.
                            "id": "MessageText",
                            // Marker value.
                            "markerValue": "MessageText",
                            // Component class name.
                            "className":
                                "Terrasoft.SelectionHandlerMultilineLabel",
                            // CSS-style setup.
                            "classes": {
                                "multilineLabelClass": ["messageText"]
                            },
                            // Caption.
                            "caption": {
                                "bindTo": "Message"
                            },
                            "showLinks": true,
                            // Binding of the selected text modification event to
                            // the handler method.
                            "selectedTextChanged": {"bindTo":
                                "onSelectedTextChanged"}, // Binding of the selected text floating icon
                            // clicking event to the handler method.
                            "selectedTextHandlerButtonClick": {"bindTo":
                                "onSelectedTextButtonClick"}, // Floating icon display checkbox.
                            "showFloatButton": true
                        }
                    }
                }
            }
        ]
    }
}
```

```

        } ;
    }
}
] /***SCHEMA_DIFF*/
);
)
;
)
```

A floating icon will be displayed on the [Processing] tab of the case page when you select a text from the internal case feed post upon your saving the schema and updating the page (Fig.2). A new case with automatically populated fields will be added upon clicking the floating icon (Fig.3).

Fig. 2. Floating icon upon selecting a text

The screenshot shows a Creatio Case page for Case #SR00000040. The left sidebar lists categories like Category, Incident, Service, and Assignee. The main area shows a history feed with a message from John Best. A floating orange icon with a plus sign appears over the text "I have a couple of questions". On the right, a vertical toolbar has icons for user, gear, question, mail, message, bell, and file.

Fig. 3. New case

The screenshot shows a Creatio Case creation page for Case #SR00000005. The left sidebar lists resolution time, priority (Medium), contact (Bruce Clayton), account (Axiom), and SLA. The main area shows a timeline with status steps: New, In prog..., Waiting..., Resolved, Closed. Below the timeline, the CASE INFORMATION tab is selected, showing a red-bordered box containing the subject and description fields, both set to "I have a couple of questions". On the right, a vertical toolbar has icons for user, gear, question, mail, message, bell, and file.

Contents

- **How to create custom verification action page**
- **Using the EntityMapper schema**

How to create custom verification action page

Beginner

Easy

Medium

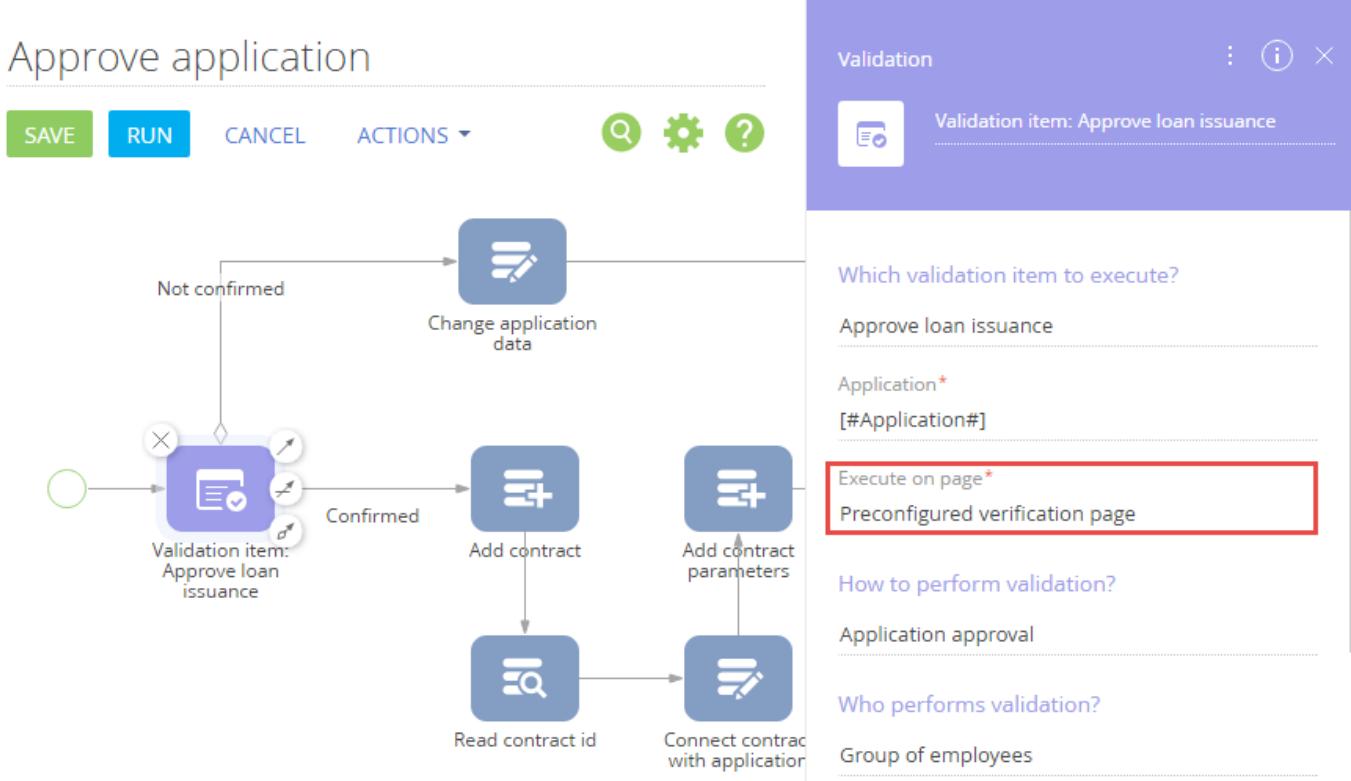
Advanced

Introduction

Verification action is a confirmation that the data in the application form corresponds to the requirements of the application. The verification action is performed to check the data provided by the clients when they fill out their application forms.

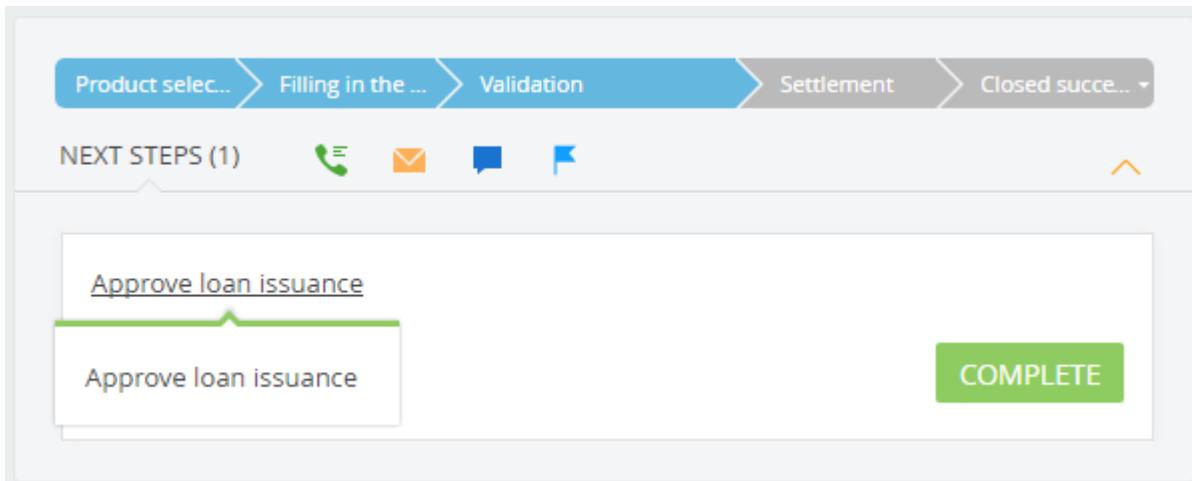
When creating a custom verification action page, for example, in the *Approve application* business process, you can select the [Preconfigured verification page].

Fig. 1 Selecting the verification page



The page is displayed after clicking the [Complete] button of the [Approve loan issuance] activity that is created when the application is moved to the [Validation] stage (fig. 2).

Fig. 2 Activity on the [Validation] stage

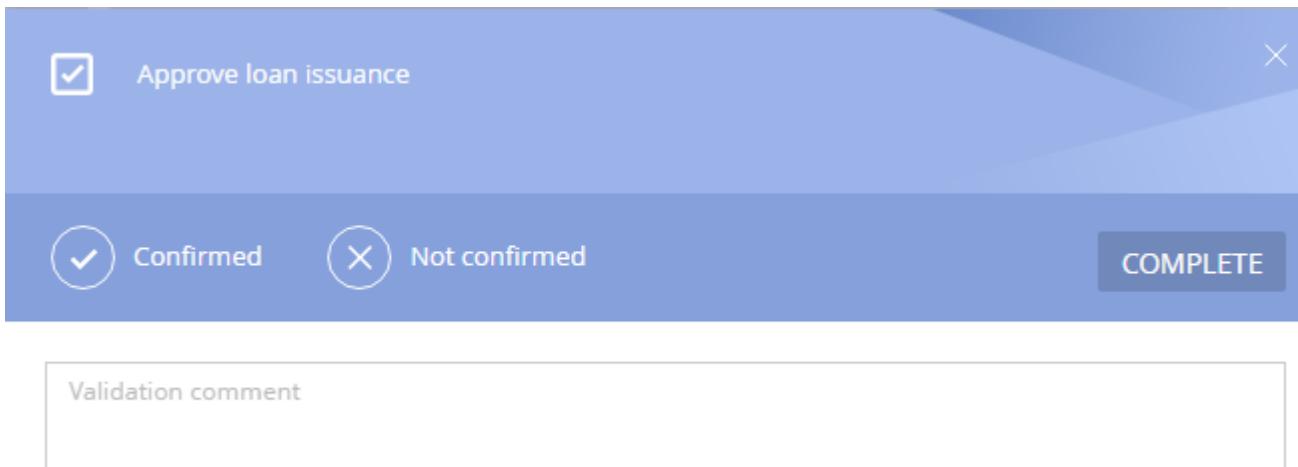


Preconfigured verification page contains (Fig. 3):

1. Buttons for selecting the result of the verification action.
2. The [Comment] field – comments to the verification action.
3. The [Conversation script] detail – contains a hints for the verifiers who call customers in the process of verification. Read only.
4. The [Attachments] detail – contains files and links attached to the validation stage. Read only.
5. The [Checklist] detail – contains control questions and answers to them.

If a detail has no attached data, it will not be displayed to save page space.

Fig. 3 Verification page



You can create custom verification pages, inheriting them from the preconfigured page. To create a custom page:

1. Create a custom schema of the verification action page.
2. Use the schema created in the business process.

Case description

Create a verification action page where the [Comment] field is hidden.

Case implementation algorithm

1. Create a schema of the verification action page

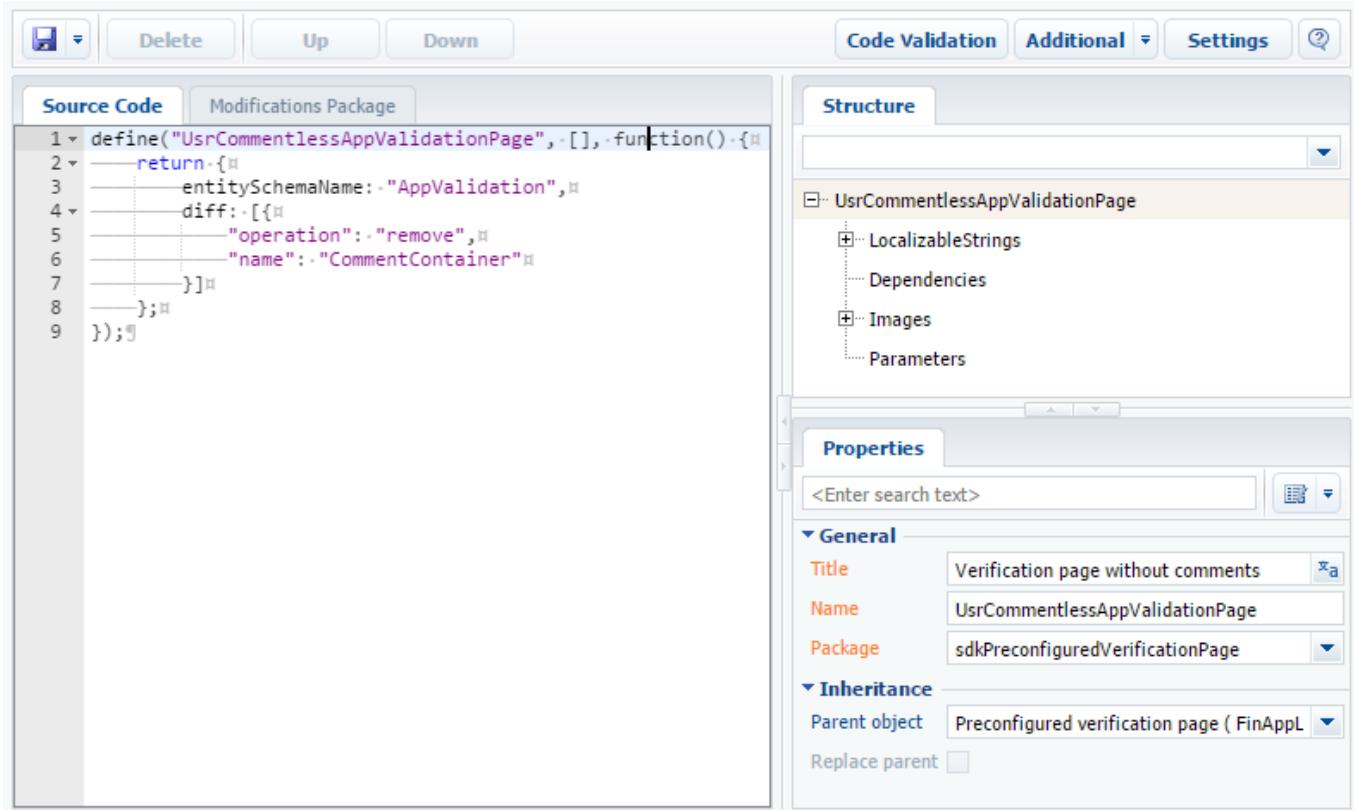
To do this, go to the [Configuration] section and select a custom package. Then execute the [Add] > [Schema of the Edit Page View Model] command. The process of creating custom schema of the view model is covered in the

"Creating a custom client module schema" article.

You need to assign the following properties for the created schema (Fig. 4):

- [Title] – Verification page without comments.
- [Name] – UsrCommentlessAppValidationPage.
- [Package] – Custom (or another custom package).
- [Parent object] – Preconfigured verification page of the *FinAppLending* package.

Fig. 4 Schema properties of the view model page



Add the following source code to the [Source code] tab:

```

define("UsrCommentlessAppValidationPage", [], function() {
  return {
    entitySchemaName: "AppValidation",
    diff: [
      {
        "operation": "remove",
        "name": "CommentContainer"
      }
    ]
  };
});

```

The [Comment] field is removed from the parent element in the **diff array**.

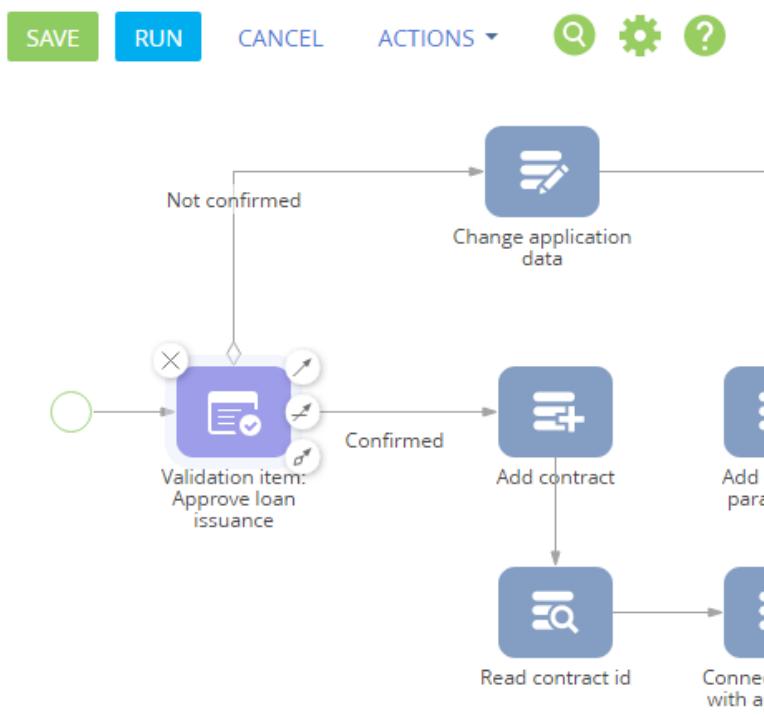
Save the schema to apply the changes.

2. Use the schema created in the business process.

To use the created schema, specify it in the [Execute on page] field of the [Validation item] item of the business process. This schema can be used in both new and existing business processes, such as *Approve application* (Fig. 5).

Fig. 5 Specifying the custom verification page

Approve application



Validation

Validation item: Approve loan issuance

Which validation item to execute?

Approve loan issuance

Application*
[#Application#]

Execute on page*
Verification page without comments

How to perform validation?

Application approval

Who performs validation?

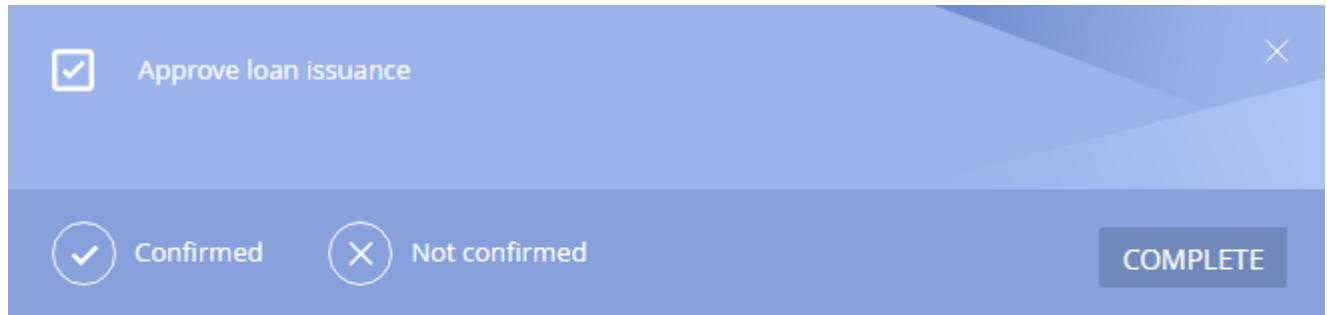
Group of employees

Save the business process to apply the changes.

Restart the application in IIS for the changes to take effect.

After the changes are applied, the previous verification page (Fig. 3) will be replaced with a custom page that does not contain the [Comment] field (Fig. 6).

Fig. 6 Verification page without the [Comment] field.



Using the EntityMapper schema

Beginner

Easy

Medium

Advanced

Introduction

Terrasoft.Configuration.EntityMapper ('Terrasoft.Configuration.EntityMapper class' in the on-line documentation) is the utility configuration class, implemented in the *EntityMapper* schema of the [FinAppLending] package in Financial Services Creatio, lending edition. *EntityMapper* enables you to match the data of one *entity* with another according to the rules defined in the configuration file. This approach prevents the creation of monotonous code.

Creatio lending features two objects with identical columns – [Contact] and [AppForm]. There are several details related to the [Contact] object and having similar details pertaining to [AppForm]. When the application is filled, there should be a possibility to get a list of all columns and values by the [Id] column of the [Contact] object, as well as a list of necessary details with their columns and values, and match this data with the application form data. After that, you can automatically fill out the fields of the application form with mapped data. This enables you to reduce manual data input.

Case description

Create a custom *UsrEntityMapperConfigsContainer* class to check the data matching mechanism through *Terrasoft.Configuration.EntityMapper*. Implement the data matching logic for the [Contact] and [AppForm] objects in this class. Implement a custom configuration service for data matching on the client side of the application. Add a button that will launch the custom configuration service on the edit page of the application form. The result has to be displayed in the browser's console.

Case implementation algorithm

1. Create a custom UsrEntityMapperConfigsContainer class for data matching.

Learn more about the process of creating the [Source code] schema in the “[Creating the \[Source code\] schema](#)” article.

Property values for the created schema:

- [Title] – "UsrEntityMapperConfigsContainer".
- [Name] – "UsrEntityMapperConfigsContainer".
- [Package] – "Custom" (or a different custom package).

Add the following source code on the [Source Code] tab of the schema designer:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
    // This class contains mapping settings.
    public class UsrEntityMapperConfigsContainer
    {
        // Settings for contact and application form mapping.
        public MapConfig ContactToAppFormConfig { get; protected set; }

        public UsrEntityMapperConfigsContainer() {
            this.InitContactToAppFormConfig();
        }
        // Configures the mapping of contact and application form objects.
        protected virtual void InitContactToAppFormConfig() {
            var columns = new Dictionary<string, string>();
            // In this case, the column names of the contact and the application form
            coincided.
            columns.Add("Surname", "Surname");
            columns.Add("GivenName", "GivenName");
            columns.Add("MiddleName", "MiddleName");
            columns.Add("INN", "INN");
            columns.Add("SpouseSurname", "SpouseSurname");
            columns.Add("SpouseGivenName", "SpouseGivenName");
            columns.Add("SpouseMiddleName", "SpouseMiddleName");
            columns.Add("Spouse", "Spouse");
            var config = new MapConfig {
                ContactToAppFormConfig = ContactToAppFormConfig,
                Columns = columns
            };
            config.AddMapping("INN", "INN");
            config.AddMapping("Spouse", "Spouse");
            config.AddMapping("SpouseGivenName", "SpouseGivenName");
            config.AddMapping("SpouseMiddleName", "SpouseMiddleName");
            config.AddMapping("SpouseSurname", "SpouseSurname");
            config.AddMapping("GivenName", "GivenName");
            config.AddMapping("MiddleName", "MiddleName");
            config.AddMapping("Surname", "Surname");
        }
    }
}
```

```

        SourceEntityName = "Contact",
        Columns = columns,
        RelationEntities = new List<RelationEntityMapConfig>() {
            new RelationEntityMapConfig() {
                SourceEntityName = "Contact",
                ParentColumnName = "Spouse",
                Columns = new Dictionary<string, string>() {
                    { "Surname", "SpouseSurname" },
                    { "BirthDate", "SpouseBirthDate" }
                }
            }
        },
        DetailsConfig = new List<DetailMapConfig>() {
            new DetailMapConfig() {
                SourceEntityName = "ContactAddress",
                DetailName = "RegistrationAddressFieldsDetail",
                Columns = new Dictionary<string, string>() {
                    { "AddressType", "AddressType" },
                    { "Country", "Country" },
                    { "Region", "Region" }
                },
                Filters = new List<EntityFilterMap>() {
                    new EntityFilterMap() {
                        ColumnName = "AddressType",
                        Value = BaseFinanceConst.RegistrationAddressTypeId
                    }
                }
            }
        },
        CleanDetails = new List<string>() {
            "AppFormIncomeDetail"
        }
    };
    this.ContactToAppFormConfig = config;
}
}

```

Publish the schema to apply changes.

2. Create a custom configuration service for data matching

The process of creating a custom configuration service is described in the **[“Creating a user configuration service \(on-line documentation\)”](#)** article.

Create the [Source Code] schema in a custom package. Property values for the created schema:

- [Title]— "UsrEntityMappingService".
 - [Name] — "UsrEntityMappingService".
 - [Package] — "Custom" (or a different custom package).

Add the following source code on the [Source Code] tab of the schema designer:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Linq;
    using System.Collections.Generic;
    using System.Runtime.Serialization;
    using System.ServiceModel;
    using System.ServiceModel.Activation;
    using System.ServiceModel.Web;
    using Terrasoft.Core;
```

```
using Terrasoft.Core.Factories;
using Terrasoft.Core.Entities;
using Terrasoft.Common;
using System.Web;
using Terrasoft.Web.Common;
using Terrasoft.Nui.ServiceModel.DataContract;
using Terrasoft.Common.Json;
using Terrasoft.Core.Configuration;

/// Service class for mapping entities and their details.
[ServiceContract]
[AspNetCompatibilityRequirements(RequirementsMode =
AspNetCompatibilityRequirementsMode.Required)]
public class UsrEntityMappingService: BaseService
{
    private EntityMapper _entityMapper;
    // Returns an EntityMapper instance.
    protected virtual EntityMapper EntityMapper {
        get {
            return _entityMapper ?? (_entityMapper =
ClassFactory.Get<EntityMapper>(
                new ConstructorArgument("userConnection", this.UserConnection)));
        }
    }
    // Returns mapping settings.
    protected virtual MapConfig GetConfig() {
        UsrEntityMapperConfigsContainer mapperConfigsContainer = new
UsrEntityMapperConfigsContainer();
        return mapperConfigsContainer.ContactToAppFormConfig;
    }
    // Performs the mapping and returns the result. The main method of service.
    [OperationContract]
    [return: MessageParameter(Name = "result")]
    [WebInvoke(Method = "POST", BodyStyle = WebMessageBodyStyle.Wrapped,
    RequestFormat = WebMessageFormat.Json, ResponseFormat =
WebMessageFormat.Json)]
    public EntityMappingResult GetMappedEntity(string id)
    {
        EntityMappingResult result = new EntityMappingResult();
        try {
            Guid recordId;
            MapConfig config = this.GetConfig();
            EntityResult entityResult = new EntityResult(config);
            result.columns = entityResult.Columns;
            result.details = entityResult.Details;
            if (!Guid.TryParse(id, out recordId)) {
                return result;
            }
            entityResult = EntityMapper.GetMappedEntity(recordId, config);
            result.columns = entityResult.Columns;
            result.details = entityResult.Details;
            result.Success = true;
        } catch (Exception e){
            result.Success = false;
            result.Exception = e;
        }
        return result;
    }
}
```

Publish the schema to apply changes.

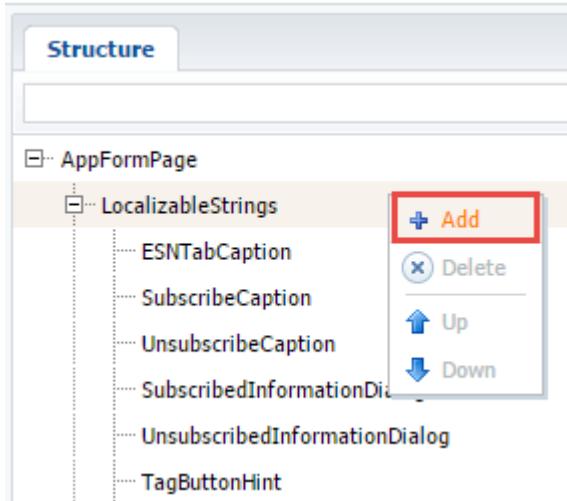
3. Adding a data mapping button to the application form edit page

To add a data mapping button, replace the existing [Application Form Edit Page] schema. The procedure for creating a replacing client schema is covered in the “**Creating a custom client module schema**”.

Add a localizable string to a replacing schema (Fig. 1) with the following properties:

- [Title] – “Call service”.
- [Name] – “EntityMappingButtonCaption”.

Fig. 1. Adding a localizable string



Add the following source code on the [Source Code] tab of the schema designer:

```
define("AppFormPage", [], function() {
    return {
        entitySchemaName: "AppForm",
        methods: {
            // User service request function.
            requestContactData2: function() {
                var data = {
                    id: this.get("Contact").value
                };
                // A configuration object for passing parameters to the service.
                var config = {
                    serviceName: "UsrEntityMappingService",
                    methodName: "GetMappedEntity",
                    data: data
                };
                // Calling a service.
                this.callService(config, this.parseMappedEntityResponse2, this);
            },
            //Callback-function for outputting the service response to the
            //console.
            parseMappedEntityResponse2: function(response) {
                window.console.log("Response from UsrEntityMappingService",
response);
            }
        },
        diff: /**SCHEMA_DIFF*/ [
            {
                "operation": "insert",
                "parentName": "ProfileContainer",
                "propertyName": "items",
            }
        ]
    }
});
```

```
        "name": "EntityMappingButton",
        "values": {
            itemType: Terrasoft.ViewItemType.BUTTON,
            "style": Terrasoft.controls.ButtonEnums.style.GREEN,
            // Bind the button's title to the localized string of the schema.
            caption: { bindTo: "Resources.Strings.EntityMappingButtonCaption"
},
            // Bind the button click method-handler.
            click: { bindTo: "requestContactData2" },
            "layout": {
                "column": 0,
                "row": 2,
                "colSpan": 24
            }
        }
    }
]/**SCHEMA_DIFF*/
};
});
```

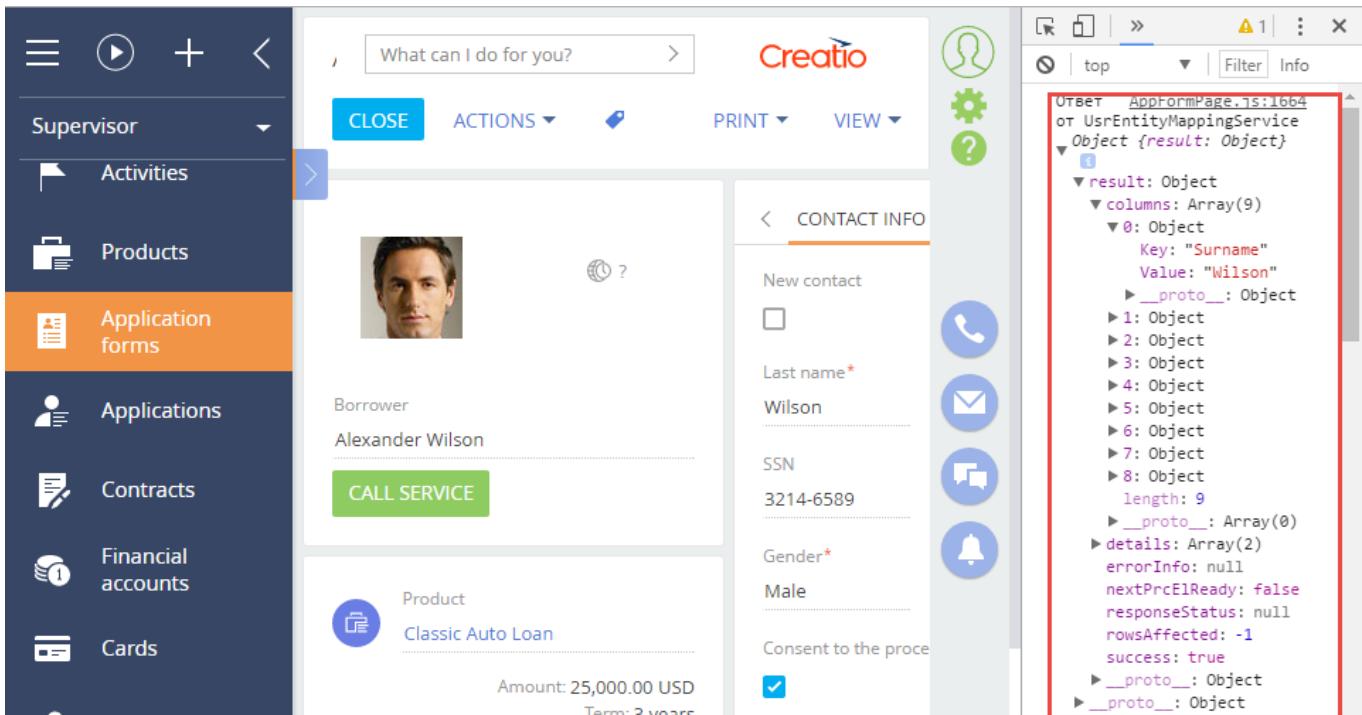
Add the configuration object to the **diff array**. The object will be used to add the data matching button to the application form edit page. The *requestContactData2()* method is called when the button is pressed, and the *UsrEntityMappingService* configuration service is called in the method with all necessary parameters. The *parseMappedEntityResponse2()* callback-function will display the service response in the browser console.

Instead of browser console output, you can implement the mechanism of auto completing fields of the application form edit page with the matched values. However, this functionality is already implemented in the `requestContactData()` and `parseMappedEntityResponse()` methods of the `AppFormPage` parent schema.

Save the schema to apply changes.

As a result, the button will appear on the application form edit page. When you press the button, the object with mapped data will be displayed in the browser console.

Fig. 2. Case result



If the profile edit page is open in the new record creation mode, you must first select or create a contact connected to the created application form. If a contact is not selected, an exception will occur, because `This.get ("Contact")` returns `null`.

Marketing Creatio customization

Contents

- **Adding a custom campaign element**
- **Configuring campaign elements for working with triggers**
- **Adding a custom transition (flow) to a new campaign element**
- **Creating Web-to-Case landing pages**
- **Web-To-Object. Integration via landings and web-forms**
- **Setting up web forms for a custom object**

Adding a custom campaign element

Beginner Easy Medium **Advanced**

Introduction

Use the Campaign designer to set up marketing campaigns. Using this designer, you can create a campaign diagram that consists of interconnected elements. In addition to default campaign elements you can create custom ones.

The general procedure for adding a custom campaign element is as follows:

1. Create a new element for the Campaign designer.
2. Create the element's edit page.
3. Expand the Campaign designer menu with a new element.
4. Create the element's server part.
5. Create executable element for the new campaign element.
6. Add custom logic for processing campaign events.

Case description

Create a new campaign element for sending text messages (SMS) for users.

Case implementation algorithm

1. Creating a new element for the Campaign designer

To display the element in the Campaign designer UI, add a new module schema for the campaign element. The procedure for creating a module schema is covered in the “**Creating a custom client module schema**” article. Set the following properties for the created schema:

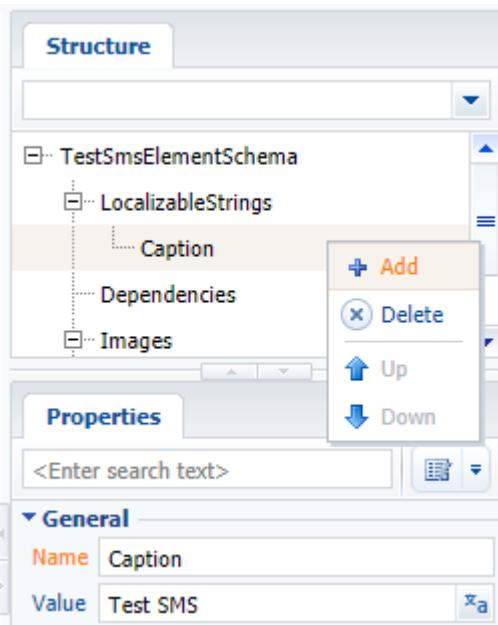
- [Title] – "Test SMS Element Schema".
- [Name] – "TestSmsElementSchema".

The schema names in the case below do not contain the *Usr* prefix. You can change the default prefix in the [Prefix for object name] (*SchemaNamePrefix*) system setting.

Add a localized string (Fig. 1) to the schema:

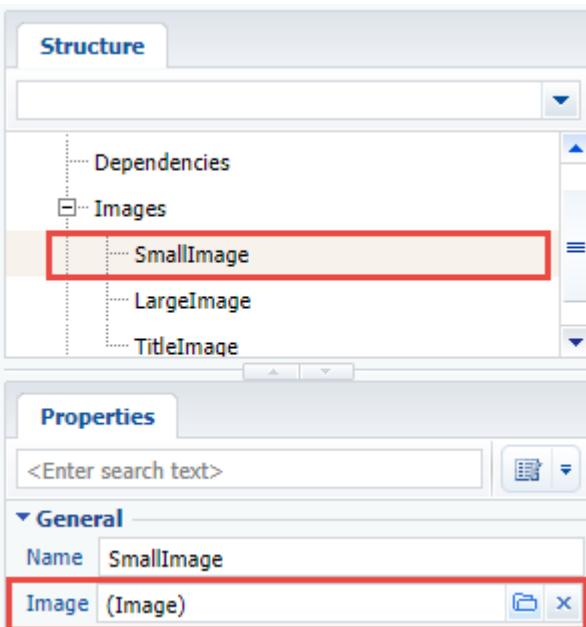
- [Name] – "Caption".
- [Value] – "Test SMS".

Fig. 1. Adding localized string to the schema



Add images that will represent the campaign element in the Campaign designer. Use the *SmallImage*, *LargeImage* and *TitleImage* (Fig. 2) properties to add the images.

Fig. 2. Adding a campaign element image



In this example we used a scalable vector graphics (SVG) image available [here](#).

Add following source code on the [Source Code] section of the schema":

```
define("TestSmsElementSchema", ["TestSmsElementSchemaResources",
"CampaignBaseCommunicationSchema"],
function(resources) {
    Ext.define("Terrasoft.manager.TestSmsElementSchema", {
        // Parent schema.
        extend: "Terrasoft.CampaignBaseCommunicationSchema",
        alternateClassName: "Terrasoft.TestSmsElementSchema",
        // Manager Id. Must be unique.
        managerItemId: "a1226f93-f3e3-4baa-89a6-11f2a9ab2d71",
        // Plugged mixins.
        mixins: {
```

```

        campaignElementMixin: "Terrasoft.CampaignElementMixin"
    },
    // Element name.
    name: "TestSms",
    // Resource binding.
    caption: resources.localizableStrings.Caption,
    titleImage: resources.localizableImages.TitleImage,
    largeImage: resources.localizableImages.LargeImage,
    smallImage: resources.localizableImages.SmallImage,
    // Schema name of the edit page.
    editPageSchemaName: "TestSmsElementPropertiesPage",
    // Element type.
    elementType: "TestSms",
    // Full name of the class that corresponds to the current schema.
    typeName: "Terrasoft.Configuration.TestSmsElement",
    Terrasoft.Configuration",
    // Overriding the properties of visual styles.
    color: "rgba(249, 160, 27, 1)",
    width: 69,
    height: 55,
    // Setting up element-specific properties.
    smsText: null,
    phoneNumber: null,
    // Determining the types of the elemen's outbound connections.
    getConnectionUserHandles: function() {
        return ["CampaignSequenceFlow", "CampaignConditionalSequenceFlow"];
    },
    // Expnding the properties for serialization.
    getSerializableProperties: function() {
        var baseSerializableProperties = this.callParent(arguments);
        return Ext.Array.push(baseSerializableProperties, ["smsText",
    "phoneNumber"]);
    },
    // Setting up the icons that are displayed on the campaign diagram.
    getSmallImage: function() {
        return this.mixins.campaignElementMixin.getImage(this.smallImage);
    },
    getLargeImage: function() {
        return this.mixins.campaignElementMixin.getImage(this.largeImage);
    },
    getTitleImage: function() {
        return this.mixins.campaignElementMixin.getImage(this.titleImage);
    }
});
return Terrasoft.TestSmsElementSchema;
});

```

Specifics:

- The *managerItemId* property value must be unique for the new element and not repeat the value of the other elements.
- The *typeName* property contains the name of the C# class that corresponds to the campaign element name. This class will be saving and reading the element's properties from the schema metadata.

Save the schema to apply changes.

Adding a group of elements

If a new group of elements, such as [Scripts] must be created for the campaign element, the schema source code must be supplemented with the following code:

```
// Name of the new group.
```

```

group: "Scripts",

constructor: function() {
    if (!Terrasoft.CampaignElementGroups.Items.contains("Scripts")) {
        Terrasoft.CampaignElementGroups.Items.add("Scripts", {
            name: "Scripts",
            caption: resources.localizableStrings.ScriptsElementGroupCaption
        });
    }
    this.callParent(arguments);
}

```

Also, add a localized string with the following properties:

- [Name] – "ScriptsElementGroupCaption".
- [Name] – "Scripts".

Save the schema to apply changes.

2. Creating the element's edit page

Create the campaign element's edit page in the custom package to enable the users to view and edit the element's properties. To do this, create a schema that expands *BaseCampaignSchemaElementPage* (*CampaignDesigner* package). The procedure for creating a replacing client schema is covered in the "[Creating a custom client module schema](#)" article.

Set the following properties for the created schema:

- [Title] – "TestSmsElementPropertiesPage".
- [Name] – "TestSmsElementPropertiesPage".
- [Parent object] – "BaseCampaignSchemaElementPage".

Add localized strings to the created schema (Fig. 1) with properties given in the table 1.

Table 1. Localized string properties

Name	Value
PhoneNumberCaption	Sender phone number
SmsTextCaption	Message
TestSmsText	Which text message should be sent? (Which SMS text to send?)

Add following source code on the [Source Code] section of the schema":

```

define("TestSmsElementPropertiesPage", [],
    function() {
        return {
            attributes: {
                // Attributes that correspond to specific properties of element
                schema.
                "PhoneNumber": {
                    "dataValueType": this.Terrasoft.DataValueType.TEXT,
                    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN
                },
                "SmsText": {
                    "dataValueType": this.Terrasoft.DataValueType.TEXT,
                    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN
                }
            },
            methods: {
                init: function() {
                    this.callParent(arguments);
                    this.initAcademyUrl(this.onAcademyUrlInitialized, this);
                }
            }
        };
    }
);

```

```
        },
        // Element code for generating a contextual help link.
        getContextHelpCode: function() {
            return "CampaignTestSmsElement";
        },
        // Initialization of attributes with the current schema property
values.
        initParameters: function(element) {
            this.callParent(arguments);
            this.set("SmsText", element.smsText);
            this.set("PhoneNumber", element.phoneNumber);
        },
        // Saving schema properties.
        saveValues: function() {
            this.callParent(arguments);
            var element = this.get("ProcessElement");
            element.smsText = this.getSmsText();
            element.phoneNumber = this.getPhoneNumber();
        },
        // Reading current attribute values.
        getPhoneNumber: function() {
            var number = this.get("PhoneNumber");
            return number ? number : "";
        },
        getSmsText: function() {
            var smsText = this.get("SmsText");
            return smsText ? smsText : "";
        }
    },
    diff: [
        // UI container.
        {
            "operation": "insert",
            "name": "ContentContainer",
            "propertyName": "items",
            "parentName": "EditorsContainer",
            "className": "Terrasoft.GridLayoutEdit",
            "values": {
                "itemType": Terrasoft.ViewItemType.GRID_LAYOUT,
                "items": []
            }
        },
        // Element primary signature.
        {
            "operation": "insert",
            "name": "TestSmsLabel",
            "parentName": "ContentContainer",
            "propertyName": "items",
            "values": {
                "layout": {
                    "column": 0,
                    "row": 0,
                    "colSpan": 24
                },
                "itemType": this.Terrasoft.ViewItemType.LABEL,
                "caption": {
                    "bindTo": "Resources.Strings.TestSmsText"
                },
                "classes": {
                    "labelClass": ["t-title-label-proc"]
                }
            }
        }
    ]
}
```

```
        }
    },
    // Caption for the text field where sender name is entered.
{
    "operation": "insert",
    "name": "PhoneNumberLabel",
    "parentName": "ContentContainer",
    "propertyName": "items",
    "values": {
        "layout": {
            "column": 0,
            "row": 1,
            "colSpan": 24
        },
        "itemType": this.Terrasoft.ViewItemType.LABEL,
        "caption": {
            "bindTo": "Resources.Strings.PhoneNumberCaption"
        },
        "classes": {
            "labelClass": ["label-small"]
        }
    }
},
// Text field for entering phone number.
{
    "operation": "insert",
    "name": "PhoneNumber",
    "parentName": "ContentContainer",
    "propertyName": "items",
    "values": {
        "labelConfig": {
            "visible": false
        },
        "layout": {
            "column": 0,
            "row": 2,
            "colSpan": 24
        },
        "itemType": this.Terrasoft.ViewItemType.TEXT,
        "classes": {
            "labelClass": ["feature-item-label"]
        },
        "controlConfig": { "tag": "PhoneNumber" }
    }
},
// Caption for text field for entering message text.
{
    "operation": "insert",
    "name": "SmsTextLabel",
    "parentName": "ContentContainer",
    "propertyName": "items",
    "values": {
        "layout": {
            "column": 0,
            "row": 3,
            "colSpan": 24
        },
        "classes": {
            "labelClass": ["label-small"]
        },
        "itemType": this.Terrasoft.ViewItemType.LABEL,
        "caption": {
```

```
        "bindTo": "Resources.Strings.SmsTextCaption"
    }
}
},
// Text field for entering message text.
{
    "operation": "insert",
    "name": "SmsText",
    "parentName": "ContentContainer",
    "propertyName": "items",
    "values": {
        "labelConfig": {
            "visible": false
        },
        "layout": {
            "column": 0,
            "row": 4,
            "colSpan": 24
        },
        "itemType": this.Terrasoft.ViewItemType.TEXT,
        "classes": {
            "labelClass": ["feature-item-label"]
        },
        "controlConfig": { "tag": "SmsText" }
    }
}
];
};

} i
```

Save the schema to apply changes.

3. Expanding the Campaign designer menu with a new element

To display the new element in the Campaign designer menu, expand the campaign element base schema manager. Add a schema that expands *CampaignElementSchemaManagerEx* (the *CampaignDesigner* package) to the custom package. The procedure for creating a replacing client schema is covered in the “[Creating a custom client module schema](#)” article.

Set the following properties for the created schema:

- [Title] – "TestSmsCampaignElementSchemaManagerEx".
 - [Name] – "CampaignElementSchemaManagerEx".
 - [Parent object] – "CampaignElementSchemaManagerEx".

Add following source code on the [Source Code] section of the schema":

```
require(["CampaignElementSchemaManager", "TestSmsElementSchema"],  
    function() {  
        // Adding a new schema to the list of available element schemas in the  
        Campaign designer.  
        var coreElementClassNames =  
Terrasoft.CampaignElementSchemaManager.coreElementClassNames;  
        coreElementClassNames.push({  
            itemType: "Terrasoft.TestSmsElementSchema"  
        });  
   });
```

Save the schema to apply changes.

4. Creating server part of the custom campaign element

To implement saving the campaign element properties, create a class that interacts with the application server part. The class must inherit *CampaignSchemaElement* and override the *ApplyMetaAttributeValue()* and *WriteMetaData()* methods.

Create a source code schema with the following properties:

- [Title] – "TestSmsElement".
- [Name] – "TestSmsElement".

For more information on creating source code schemas, please see the **Creating the [Source code] schema** article.

Add the following source code on the [Source Code] section of the schema":

```
namespace Terrasoft.Configuration
{
    using System;
    using Terrasoft.Common;
    using Terrasoft.Core;
    using Terrasoft.Core.Campaign;
    using Terrasoft.Core.Process;

    [DesignModeProperty(Name = "PhoneNumber",
        UsageType = DesignModeUsageType.NotVisible, MetaPropertyName =
PhoneNumberPropertyName)]
    [DesignModeProperty(Name = "SmsText",
        UsageType = DesignModeUsageType.NotVisible, MetaPropertyName =
SmsTextPropertyName)]
    public class TestSmsElement : CampaignSchemaElement
    {
        private const string PhoneNumberPropertyName = "PhoneNumber";
        private const string SmsTextPropertyName = "SmsText";
        // Default constructor.
        public TestSmsElement() {
            ElementType = CampaignSchemaElementType.AsyncTask;
        }
        // Constructor with parameter.
        public TestSmsElement(TestSmsElement source)
            : base(source) {
            ElementType = CampaignSchemaElementType.AsyncTask;
            PhoneNumber = source.PhoneNumber;
            SmsText = source.SmsText;
        }

        // Instance action Id.
        protected override Guid Action {
            get {
                return CampaignConsts.CampaignLogTypeMailing;
            }
        }

        // Phone number.
        [MetaTypeProperty("{A67950E7-FFD7-483D-9E67-3C9A30A733C0}")]
        public string PhoneNumber {
            get;
            set;
        }
        // Text message.
        [MetaTypeProperty("{05F86DF2-B9FB-4487-B7BE-F3955703527C}")]
        public string SmsText {
            get;
            set;
        }
    }
}
```

```
// Applies metadata values.
protected override void ApplyMetaDataValue(DataReader reader) {
    base.ApplyMetaDataValue(reader);
    switch (reader.CurrentName) {
        case PhoneNumberPropertyName:
            PhoneNumber = reader.GetValue<string>();
            break;
        case SmsTextPropertyName:
            SmsText = reader.GetValue<string>();
            break;
    }
}

// Records metadata values.
public override void WriteMetaData(DataWriter writer) {
    base.WriteMetaData(writer);
    writer.WriteValue(PhoneNumberPropertyName, PhoneNumber, string.Empty);
    writer.WriteValue(SmsTextPropertyName, SmsText, string.Empty);
}

// Copies element.
public override object Clone() {
    return new TestSmsElement(this);
}

// Creates a specific ProcessFlowElement instance.
public override ProcessFlowElement CreateProcessFlowElement(UserConnection
userConnection) {
    var executableElement = new TestSmsCampaignProcessElement {
        UserConnection = userConnection,
        SmsText = SmsText,
        PhoneNumber = PhoneNumber
    };
    InitializeCampaignProcessFlowElement(executableElement);
    return executableElement;
}
}
```

Publish the source code schema.

5. Creating executable element for the new campaign element

For the custom campaign element to execute the needed logic, add an executable element. It is a class that inherits the `CampaignProcessFlowElement` class, where the `SafeExecute()` method is implemented.

To create an executable element, add a source code schema element with the following properties in the custom package:

- [Title] – "TestSmsCampaignProcessElement".
 - [Name] – "TestSmsCampaignProcessElement".

Add following source code on the [Source Code] section of the schema":

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
    using Terrasoft.Core.Campaign;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Process;

    public class TestSmsCampaignProcessElement : CampaignProcessFlowElement
    {
```

```
public const string ContactTableName = "Contact";

public TestSmsCampaignProcessElement(ICampaignAudience campaignAudience) {
    CampaignAudience = campaignAudience;
}

public TestSmsCampaignProcessElement() {
}

// Audiences for whom to send texts on the current step.
private ICampaignAudience _campaignAudience;
private ICampaignAudience CampaignAudience {
    get {
        return _campaignAudience ??
            (_campaignAudience = new CampaignAudience(UserConnection,
CampaignId));
    }
    set {
        _campaignAudience = value;
    }
}

// SMS-specific properties. Passed from an instance of the TestSmsElement
class.
public string PhoneNumber {
    get;
    set;
}
public string SmsText {
    get;
    set;
}

// Implementation of the element execution method
protected override int SafeExecute(ProcessExecutingContext context) {
    // TODO: Implement sending SMS messages.
    //
    // Current step for audiences is set as completed.
    return CampaignAudience.SetItemCompleted(SchemaElementUID);
}
}
}
```

Publish the source code schema.

6. Adding custom logic for processing campaign events

Use the event handler mechanism to implement custom logic on saving, copying, deleting, running and stopping campaigns. Create a public *sealed* handler class that inherits *CampaignEventHandlerBase*. Implement interfaces that describe specific event handler signatures. This class must not be generic. It must have a constructor available by default.

The following interfaces are supported in the current version:

- *IOnCampaignBeforeSave* – contains method that will be called before saving the campaign.
- *IOnCampaignAfterSave* – contains method that will be called after saving the campaign.
- *IOnCampaignDelete* – contains method that will be called before deleting the campaign.
- *IOnCampaignStart* – contains method that will be called before running the campaign.
- *IOnCampaignStop* – contains method that will be called before stopping the campaign.
- *IOnCampaignValidate* – contains method that will be called on validating the campaign.
- *IOnCampaignCopy* – contains method that will be called after copying the campaign.

If an exception occurs during the event processing, the call chain is stopped, and campaign status is reverted to the previous one in DB.

When implementing the *IOnCampaignValidate* interface, save errors in the campaign schema using the *AddValidationInfo(string)* method.

Additional case conditions

In order for the new custom campaign element to work, SMS gateway connection is required. The connection, account status and other parameters must be checked during campaign validation. The messages must be sent when campaign starts.

To implement these conditions, add a source code schema element with the following properties in the custom package:

- [Title] – "TestSmsEventHandler".
- [Name] – "TestSmsEventHandler".

Add following source code on the [Source Code] section of the schema":

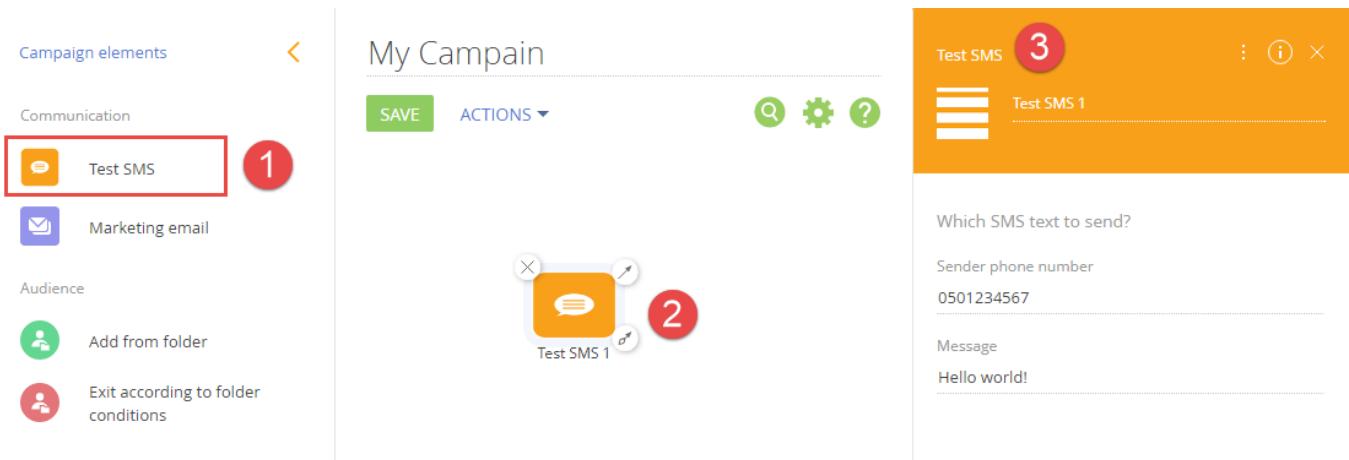
```
namespace Terrasoft.Configuration
{
    using System;
    using Terrasoft.Core.Campaign.EventHandler;

    public sealed class TestSmsEventHandler : CampaignEventHandlerBase,
        IOnCampaignValidate, IOnCampaignStart
    {
        // Implementing handler for the campaign start event.
        public void OnStart() {
            // TODO: Text SMS message sending logic...
            //
        }

        // Implementing event handler for campaign validation.
        public void OnValidate() {
            try {
                // TODO: SMS gateway connection validation logic...
                //
            } catch (Exception ex) {
                // If errors are found, add information to the campaign schema.
                CampaignSchema.AddValidationInfo(ex.Message);
            }
        }
    }
}
```

After making the changes, publish the schema. Compile the application and clear the cache.

As a result, a new [TestSMS] element will be added in the campaign element menu (Fig. 3, 1) that the users can add to the campaign diagram (Fig. 3, 2). When an added element is selected, its edit page will be displayed (Fig. 3, 3).



When saving the campaign, the “Parameter ‘type’ cannot be null” may occur. The error indicates that the configuration library was not updated after the compilation and therefore does not contain the new types.

Recompile the project and clear all possible storages with cached data. You may also need to clear the application pool and restart the website in IIS on the application server.

Configuring campaign elements for working with triggers

[Beginner](#) [Easy](#) [Medium](#) [Advanced](#)

Introduction

Starting from version 7.12.4, a new [Triggered adding] campaign element has been added to Creatio. The way the element works with campaign audience has been changed: as soon as the trigger is run, the synchronous campaign element is launched.

Thus, new and existing campaign elements must be configured for working with this element.

To include the custom element into the synchronous fragment and make sure it works correctly both for the scheduled execution and triggers, make the following changes on the server side:

- Specify the additional *CampaignSchemaElementType.Sessioned* type for the element.
- The executed element for the campaign custom element (the inheritor of the *CampaignProcessFlowElement* class) must work using the *CampaignAudience* base property. Perform all operations (specifying the audience, selecting the [Step complete] checkbox) via the object in the *CampaignAudience* property of the *ICampaignAudience* type.

Case description

Configure the marketing campaign element for sending SMS messages to users. Learn how to create this element in the “[Adding a custom campaign element](#)” article.

Case implementation algorithm

1. Modifying the class that interacts with the server side of the application

Perform steps 1-6 from the “[Adding a custom campaign element](#)” article before you implement the case.

Change the *TestSmsElement* source code schema class (the inheritor of the *CampaignSchemaElement* class). Specify additional *ElementType* – *CampaignSchemaElementType.Sessioned* in the class constructor.

```
public TestSmsElement() {
    // TestSmsElement is asynchronous and session element that can be triggered.
    ElementType = CampaignSchemaElementType.AsyncTask |
    CampaignSchemaElementType.Sessioned;
}
```

2. Modifying the executed element for the new campaign element

Modify the *TestSmsCampaignProcessElement* source code schema class (the inheritor of the *CampaignProcessFlowElement* class). Add the audience reading operation to the *SafeExecute()* method implementation via the object in the *CampaignAudience* property of the *ICampaignAudience* type.

```
// Method implementation of the element performance.
protected override int SafeExecute(ProcessExecutingContext context) {
    // TODO: Implement sending SMS-messages.
    //
    //
    // Receive the audience that is available for processing by the element at
    the moment..
    var audienceSelect = CampaignAudience.GetItemAudienceSelect(CampaignItemId);
    //
    // Specify the current audience step as [Complete].
    return CampaignAudience.SetItemCompleted(SchemaElementUID);
}
```

Adding a custom transition (flow) to a new campaign element

Beginner Easy Medium **Advanced**

Introduction

Use [Campaign designer] to set up your marketing campaigns. You can create a visual campaign diagram that would consist of interconnected pre-configured elements. You can also add custom campaign elements.

General algorithm of adding a custom flow (an arrow):

1. Create a new schema for the flow element.
2. Create the edit page for the flow element properties.
3. Create the server part for the flow element.
4. Create the executed element for the flow transition.
5. Create the *CampaignConnectorManager* replacing module for adding the flow operation logic.
6. Connect the *CampaignConnectorManager* replacing module.

Case description

Create a custom flow (an arrow) from the **new campaign element for sending SMS-messages to a user** and add a possibility to select the bulk sms response condition. On the setup page, a user can select an option to either ignore the responses or take them into consideration based on the list of possible bulk sms response types. If no response type is selected, the flow is enabled for any response.

In this case, the *Usr* prefix is not used in schema names. You can change the prefix used by default in the [Prefix for object name] system setting (the *SchemaNamePrefix* code).

Perform steps 1-6 from the “**Adding a custom campaign element**” article before you implement the case.

Source code

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

Case implementation algorithm

1. Creating the [TestSmsTarget] and [TestSmsResponseType] objects

Create schemas for the [TestSmsTarget] (fig. 1) and [TestSmsResponseType] (fig. 2) objects in the development package.

Learn more about creating object schemas in the “**Creating the entity schema**” article.

Add a column with the following properties to the [TestSmsResponseType] schema:

- [Title] – “Name”
- [Name] – “Name”
- [Data type] – “Text (50 characters)”

Add columns with the following properties to the [TestSmsTarget] schema:

Table 1. Primary column properties of the [TestSmsTarget] object schema

[Title]	[Name]	[Data type]
Phone number	PhoneNumber	“Text (50 characters)”
SMS text	SmsText	“Text (50 characters)”
Contact	Contact	“Lookup” – “Contact”
Test SMS response	TestSmsResponse	“Lookup” – “TestSmsResponseType”

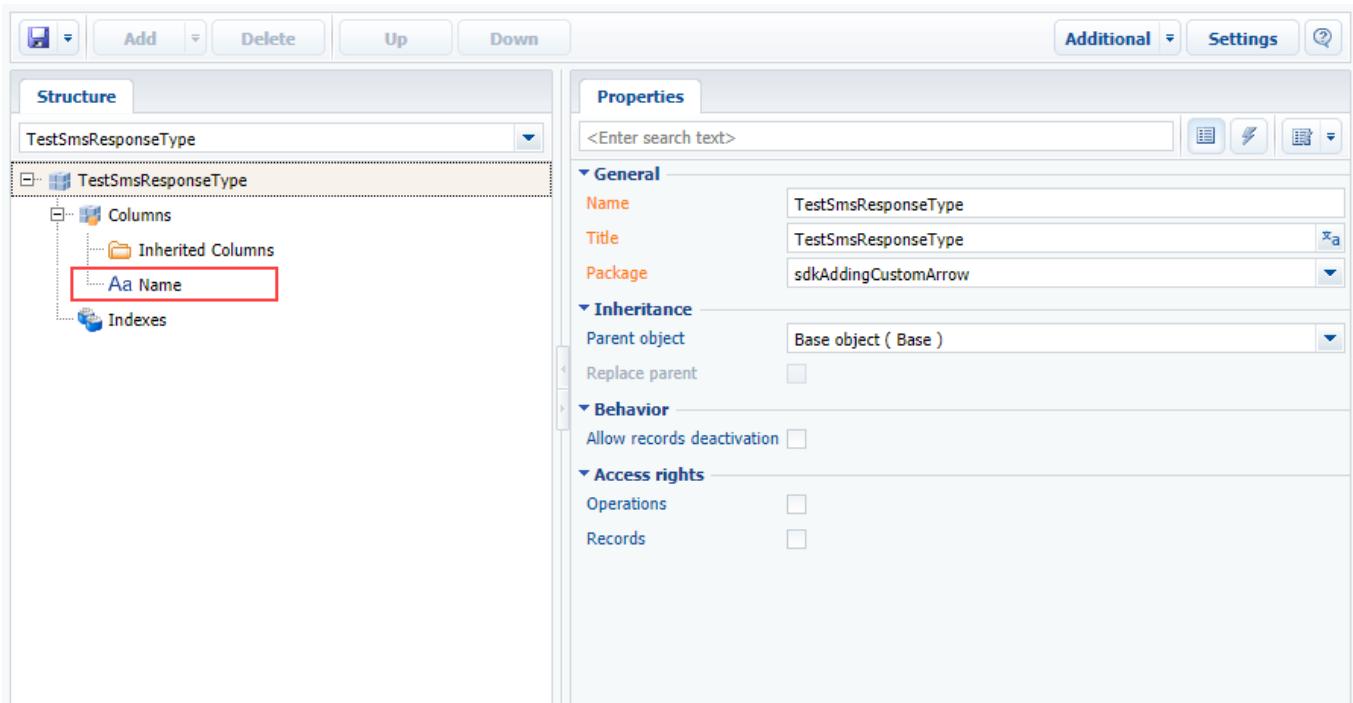
Learn more about adding object schema columns in the “[Creating the entity schema](#)” article.

Fig. 1. Columns and properties of the [TestSmsTarget] object schema

The screenshot shows the Creatio object schema editor interface. On the left, the 'Structure' tab displays the object tree under 'TestSmsTarget'. A red box highlights the 'Columns' section, which contains four items: 'PhoneNumber', 'SmsText', 'Contact', and 'TestSmsResponse'. On the right, the 'Properties' tab shows the following configuration for the [TestSmsTarget] object:

Name	TestSmsTarget
Title	TestSmsTarget
Package	sdkAddingCustomArrow
Parent object	Base object (Base)
Allow records deactivation	<input type="checkbox"/>
Operations	<input type="checkbox"/>
Records	<input type="checkbox"/>

Fig. 2. Columns and properties of the [TestSmsResponseType] object schema



Save and publish the objects.

Create a lookup for the TestSmsResponseType object and populate it with the “Sms delivered”, “Canceled”, “Error while receiving”, etc. values (Fig. 3). The response identifiers (Ids) will be used in the edit page code of the condition flow properties from the bulk SMS.

Fig. 3. The [Test sms response] lookup

Name
Canceled
Sms delivered
Error while receiving

2. Creating a new schema for the flow element

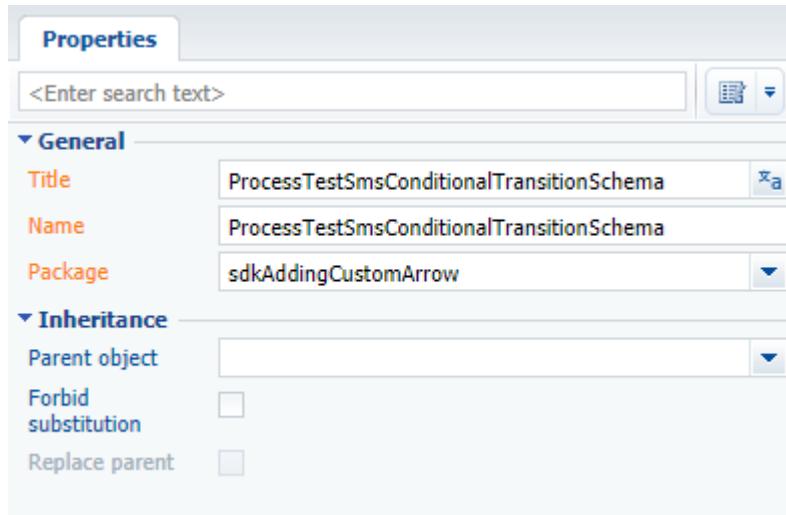
To display the element in the [Campaign designer] user interface, create a new module schema in the development package. The procedure for creating a module schema is covered in the **“Creating a custom client module schema”** article. Set the following properties for the created schema (fig. 4):

- [Title] – “ProcessTestSmsConditionalTransitionSchema”

- [Name] – “ProcessTestSmsConditionalTransitionSchema”

The schema name for the arrow must contain the “Process” prefix

Fig. 4. Properties of the ProcessTestSmsConditionalTransitionSchema module schema



Add the following source code to the [Source code] section of the schema:

```

define("ProcessTestSmsConditionalTransitionSchema", ["CampaignEnums",
    "ProcessTestSmsConditionalTransitionSchemaResources",
    "ProcessCampaignConditionalSequenceFlowSchema"],
    function(CampaignEnums) {
        Ext.define("Terrasoft.manager.ProcessTestSmsConditionalTransitionSchema", {
            extend: "Terrasoft.ProcessCampaignConditionalSequenceFlowSchema",
            alternateClassName:
"Terrasoft.ProcessTestSmsConditionalTransitionSchema",
            managerItemId: "4b5e70b0-a631-458e-ab22-856ddc913444",
            mixins: {
                parametrizedProcessSchemaElement:
"Terrasoft.ParametrizedProcessSchemaElement"
            },
            // The full type name of the connected arrow element.
            typeName: "Terrasoft.Configuration.TestSmsConditionalTransitionElement",
Terrasoft.Configuration",
            // The name of the arrow element for connecting to campaign elements.
connectionUserName: "TestSmsConditionalTransition",
            // The name of the arrow property setup page.
editPageSchemaName: "TestSmsConditionalTransitionPropertiesPage",
            elementType:
CampaignEnums.CampaignSchemaElementTypes.CONDITIONAL_TRANSITION,
            // Bulk sms response collection.
            testSmsResponseId: null,
            // The checkbox that takes into consideration the response condition for
contact transition.
            isResponseBasedStart: false,
            getSerializableProperties: function() {
                var baseSerializableProperties = this.callParent(arguments);
                // Properties for serialization and transfer to the server part when
saving.
                Ext.Array.push(baseSerializableProperties, ["testSmsResponseId",
"isResponseBasedStart"]);
                return baseSerializableProperties;
            }
        });
        return Terrasoft.ProcessTestSmsConditionalTransitionSchema;
    }
);

```

```
});
```

Save the created schema.

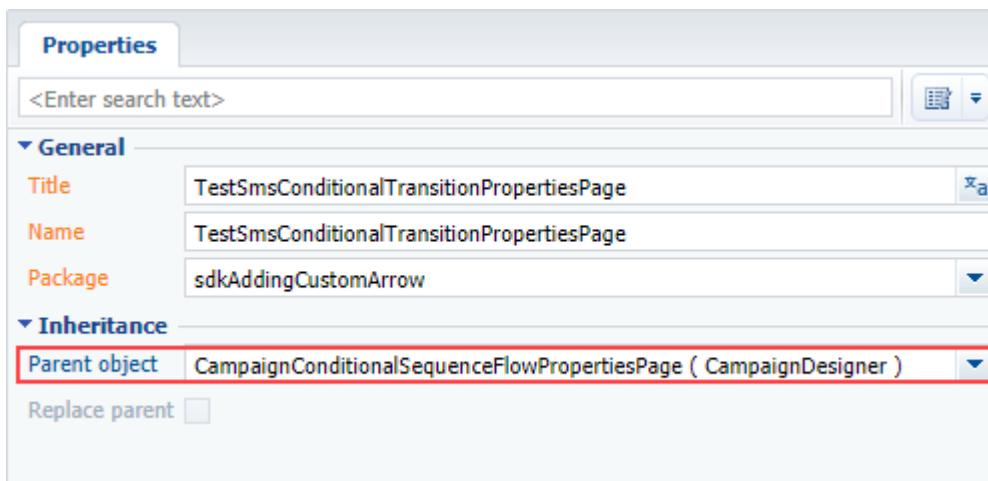
3. Creating the edit page of the flow element properties

To display and modify campaign element properties, create its edit page in the development package. Create the schema replacing *CampaignConditionalSequenceFlowPropertiesPage* (the *CampaignDesigner* package). The procedure for creating a replacing schema is covered in the “**Creating a custom client module schema**” article.

Set the following properties for the created schema (fig. 5):

- [Title] – “TestSmsConditionalTransitionPropertiesPage”
- [Name] – “TestSmsConditionalTransitionPropertiesPage”
- [Parent object] – “CampaignConditionalSequenceFlowPropertiesPage”

Fig. 5. Properties of the edit page schema



Add localizable strings, whose properties are listed in table 2 to the created schema.

Table 2. Primary properties of the localizable strings

[Name]	[Value]
ReactionModeCaption	What is the result of the {o} step?
ReactionModeDefault	Transfer participants regardless of their response
ReactionModeWithCondition	Set up responses for transferring participants
IsTestSmsDelivered	Test SMS delivered
IsErrorWhileReceiving	Error while receiving

Add the following source code to the [Source code] section of the schema:

```
define("TestSmsConditionalTransitionPropertiesPage", ["BusinessRuleModule"],
  function(BusinessRuleModule) {
    return {
      messages: {},
      attributes: {
        "ReactionModeEnum": {
          dataValueType: this.Terrasoft.DataValueType.CUSTOM_OBJECT,
          type: this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
          value: {
            Default: {
              value: "0",
              captionName: "Resources.Strings.ReactionModeDefault"
            },
            WithCondition: {

```

```
        value: "1",
        captionName:
"Resources.Strings.ReactionModeWithCondition"
    }
}
},
"ReactionMode": {
    "dataValueType": this.Terrasoft.DataValueType.LOOKUP,
    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN,
    "isRequired": true
},
"IsTestSmsDelivered": {
    "dataValueType": this.Terrasoft.DataValueType.BOOLEAN,
    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN
},
"IsErrorWhileReceiving": {
    "dataValueType": this.Terrasoft.DataValueType.BOOLEAN,
    "type": this.Terrasoft.ViewModelColumnType.VIRTUAL_COLUMN
}
},
rules:
{
    "ReactionConditionDecision": {
        "BindReactionConditionDecisionRequiredToReactionMode": {
            "ruleType": BusinessRuleModule.enums.RuleType.BINDPARAMETER,
            "property": BusinessRuleModule.enums.Property.REQUIRED,
            "conditions": [
                {
                    "leftExpression": {
                        "type": BusinessRuleModule.enums.ValueType.ATTRIBUTE,
                        "attribute": "ReactionMode"
                    },
                    "comparisonType": this.Terrasoft.ComparisonType.EQUAL,
                    "rightExpression": {
                        "type": BusinessRuleModule.enums.ValueType.CONSTANT,
                        "value": "1"
                    }
                }
            ]
        }
    }
},
methods: {
    // Ensures sms-response correspondence (based on the
    TestSmsResponseType lookup).
    // We assume that Creatio already contains the TestSmsResponseType
    lookup
    // with the TestSmsDelivered and ErrorWhileReceiving records.
    getResponseConfig: function() {
        return {
            "IsTestSmsDelivered": "F2FC75B3-58C3-49A6-B2F2-353262068145",
            "IsErrorWhileReceiving": "37B9F9D5-E897-4B7B-A65E-
3B3799A18D72"
        };
    },
    subscribeEvents: function() {
        this.callParent(arguments);
        // Connecting the handler to the event of changing the
        ReactionMode attribute value
        this.on("change:ReactionMode", this.onReactionModeLookupChanged,
this);
    },
}
```

```
// Event handler-method of changing the ReactionMode attribute.
onReactionModeLookupChanged: function() {
    var reactionModeEnum = this.get("ReactionModeEnum");
    var reactionMode = this.get("ReactionMode");
    var decisionModeEnabled = (reactionMode && reactionMode.value ===
reactionModeEnum.WithCondition.value);
    if (!decisionModeEnabled) {
        this.set("ReactionConditionDecision", null);
    }
},
// Initiates the viewModel properties to display the page when
opening.
initParameters: function(element) {
    this.callParent(arguments);
    var isResponseBasedStart = element.isResponseBasedStart;
    this.initReactionMode(isResponseBasedStart);
    this.initTestSmsResponses(element.testSmsResponseId);
},
// Auxiliary method cutting the line to the specified length and
adding allipsis at the end.
cutString: function(strValue, strLength) {
    var ellipsis = Ext.String.ellipsis(strValue.substring(strLength),
0);
    return strValue.substring(0, strLength) + ellipsis;
},
// Sets the status value to "Sms delivered".
initIsTestSmsDelivered: function(value) {
    if (value === undefined) {
        value = this.get("IsTestSmsDelivered");
    }
    this.set("IsTestSmsDelivered", value);
},
// Sets the status value to "Error while receiving".
initIsErrorWhileReceiving: function(value) {
    if (value === undefined) {
        var isErrorWhileReceiving =
this.get("IsErrorWhileReceiving");
        value = isErrorWhileReceiving;
    }
    this.set("IsErrorWhileReceiving", value);
},
// Initiates the selected responses when opening the page.
initTestSmsResponses: function(responseIdsJson) {
    if (!responseIdsJson) {
        return;
    }
    var responseIds = JSON.parse(responseIdsJson);
    var config = this.getResponseConfig();
    Terrasoft.each(config, function(propValue, propName) {
        if (responseIds.indexOf(propValue) > -1) {
            this.set(propName, true);
        }
    }, this);
},
initReactionMode: function(value) {
```

```
        var isDefault = !value;
        this.setLookupValue(isDefault, "ReactionMode", "WithCondition",
this);
    },

    // Auxiliary method extracting the identifier array from the incoming
JSON parameter.
getIds: function(idsJson) {
    if (idsJson) {
        try {
            var ids = JSON.parse(idsJson);
            if (this.Ext.isArray(ids)) {
                return ids;
            }
        } catch (error) {
            return [];
        }
    }
    return [];
},

onPrepareReactionModeList: function(filter, list) {
    this.prepareList("ReactionModeEnum", list, this);
},

// Saves the response values and the setting of adding the response
condition.
saveValues: function() {
    this.callParent(arguments);
    var element = this.get("ProcessElement");
    var isResponseBasedStart =
this.getIsReactionModeWithConditions();
    element.isResponseBasedStart = isResponseBasedStart;
    element.testSmsResponseId =
this.getTestSmsResponseId(isResponseBasedStart);
},

// Receives the serialized Ids of the selected responses.
getTestSmsResponseId: function(isResponseActive) {
    var responseIds = [];
    if (isResponseActive) {
        var config = this.getResponseConfig();
        Terrasoft.each(config, function(propValue, propName) {
            var attrValue = this.get(propName);
            if (attrValue && propValue) {
                responseIds.push(propValue);
            }
        }, this);
    }
    return JSON.stringify(responseIds);
},

getLookupValue: function(parameterName) {
    var value = this.get(parameterName);
    return value ? value.value : null;
},

getContextHelpCode: function() {
    return "CampaignConditionalSequenceFlow";
},

getIsReactionModeWithConditions: function() {
```

```
        return this.isLookupValueEqual("ReactionMode", "1", this);
    },

getSourceElement: function() {
    var flowElement = this.get("ProcessElement");
    if (flowElement) {
        return flowElement.findSourceElement();
    }
    return null;
},
// Adds the name of the element that the arrow is generated from to
the text.
getQuestionCaption: function() {
    var caption = this.get("Resources.Strings.ReactionModeCaption");
    caption = this.Ext.String.format(caption,
this.getSourceElement().getCaption());
    return caption;
}
},
diff: /**SCHEMA_DIFF*/[
// Container.
{
    "operation": "insert",
    "name": "ReactionContainer",
    "propertyName": "items",
    "parentName": "ContentContainer",
    "className": "Terrasoft.GridLayoutEdit",
    "values": {
        "layout": {
            "column": 0,
            "row": 2,
            "colSpan": 24
        },
        "itemType": this.Terrasoft.ViewItemType.GRID_LAYOUT,
        "items": []
    }
},
// Title.
{
    "operation": "insert",
    "name": "ReactionModeLabel",
    "parentName": "ReactionContainer",
    "propertyName": "items",
    "values": {
        "layout": {
            "column": 0,
            "row": 0,
            "colSpan": 24
        },
        "itemType": this.Terrasoft.ViewItemType.LABEL,
        "caption": {
            "bindTo": "getQuestionCaption"
        },
        "classes": {
            "labelClass": ["t-title-label-proc"]
        }
    }
}
```

```
        }
    },
    // List.
    {
        "operation": "insert",
        "name": "ReactionMode",
        "parentName": "ReactionContainer",
        "propertyName": "items",
        "values":
        {
            "contentType": this.Terrasoft.ContentType.ENUM,
            "controlConfig":
            {
                "prepareList":
                {
                    "bindTo": "onPrepareReactionModeList"
                }
            },
            "isRequired": true,
            "layout":
            {
                "column": 0,
                "row": 1,
                "colSpan": 24
            },
            "labelConfig":
            {
                "visible": false
            },
            "wrapClass": ["no-caption-control"]
        }
    },
    // List element.
    {
        "operation": "insert",
        "parentName": "ReactionContainer",
        "propertyName": "items",
        "name": "IsTestSmsDelivered",
        "values":
        {
            "wrapClass": ["t-checkbox-control"],
            "visible":
            {
                "bindTo": "ReactionMode",
                "bindConfig":
                {
                    converter: "getIsReactionModeWithConditions"
                }
            },
            "caption":
            {
                "bindTo": "Resources.Strings.IsTestSmsDelivered"
            },
            "layout":
            {
                "column": 0,
                "row": 2,
                "colSpan": 22
            }
        }
    },
    // List element.
```

```

    {
        "operation": "insert",
        "parentName": "ReactionContainer",
        "propertyName": "items",
        "name": "IsErrorWhileReceiving",
        "values":
        {
            "wrapClass": ["t-checkbox-control"],
            "visible":
            {
                "bindTo": "ReactionMode",
                "bindConfig":
                {
                    converter: "getIsReactionModeWithConditions"
                }
            },
            "caption":
            {
                "bindTo": "Resources.Strings.IsErrorWhileReceiving"
            },
            "layout":
            {
                "column": 0,
                "row": 3,
                "colSpan": 22
            }
        }
    }
] /**SCHEMA_DIFF*/
);
}
);
}

```

Save the created schema.

4. Creating the server part of the flow element from the [Bulk SMS] element

To implement saving the base and user campaign element properties, create a class interacting with the server part of the application. The class should inherit *CampaignSchemaElement* and override the *ApplyMetaDataValue()* and *WriteMetaData()* methods.

Create the source code schema with the following properties:

- [Title] – “TestSmsConditionalTransitionElement”
- [Name] – “TestSmsConditionalTransitionElement”

Creating the source code schema is covered in the “[Creating the \[Source code\] schema](#)” article.

Add the following source code to the [Source code] section of the schema:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.Globalization;
    using System.Linq;
    using Newtonsoft.Json;
    using Terrasoft.Common;
    using Terrasoft.Core;
    using Terrasoft.Core.Campaign;
    using Terrasoft.Core.DB;
    using Terrasoft.Core.Process;
```

```
[DesignModeProperty(Name = "TestSmsResponseId",
    UsageType = DesignModeUsageType.NotVisible, MetaPropertyName =
TestSmsResponseIdPropertyName)]
[DesignModeProperty(Name = "IsResponseBasedStart",
    UsageType = DesignModeUsageType.Advanced, MetaPropertyName =
IsResponseBasedStartPropertyName)]
public class TestSmsConditionalTransitionElement : ConditionalSequenceFlowElement
{
    private const string TestSmsResponseIdPropertyName = "TestSmsResponseId";
    private const string IsResponseBasedStartPropertyName =
"IsResponseBasedStart";

    public TestSmsConditionalTransitionElement() {}

    public
TestSmsConditionalTransitionElement(TestSmsConditionalTransitionElement source)
    : this(source, null, null) {}

    public
TestSmsConditionalTransitionElement(TestSmsConditionalTransitionElement source,
        Dictionary<Guid, Guid> dictToRebind, Core.Campaign.CampaignSchema
parentSchema)
    : base(source, dictToRebind, parentSchema) {
        IsResponseBasedStart = source.IsResponseBasedStart;
        _testSmsResponseIdJson =
JsonConvert.SerializeObject(source.TestSmsResponseId);
    }

    private string _testSmsResponseIdJson;

    private IEnumerable<Guid> Responses {
        get {
            return TestSmsResponseId;
        }
    }

    [MetaTypeProperty("{DC597899-B831-458A-A58E-FB43B1E266AC}")]
    public IEnumerable<Guid> TestSmsResponseId {
        get {
            return !_string.IsNullOrWhiteSpace(_testSmsResponseIdJson)
                ? JsonConvert.DeserializeObject<IEnumerable<Guid>>
(_testSmsResponseIdJson)
                : Enumerable.Empty<Guid>();
        }
    }

    [MetaTypeProperty("{3FFA4EA0-62CC-49A8-91FF-4096AEC561F6}",
    IsExtraProperty = true, IsUserProperty = true)]
    public virtual bool IsResponseBasedStart {
        get;
        set;
    }

    protected override void ApplyMetaDataValue(DataReader reader) {
        base.ApplyMetaDataValue(reader);
        switch (reader.CurrentName) {
            case TestSmsResponseIdPropertyName:
                _testSmsResponseIdJson = reader.GetValue<string>();
                break;
            case IsResponseBasedStartPropertyName:
                IsResponseBasedStart = reader.GetBoolValue();
        }
    }
}
```

```

        break;
    default:
        break;
    }
}

public override void WriteMetaData(DataWriter writer) {
    base.WriteMetaData(writer);
    writer.WriteValue(IsResponseBasedStartPropertyName, IsResponseBasedStart,
false);
    writer.WriteValue(TestSmsResponseIdPropertyName, _testSmsResponseIdJson,
null);
}

public override object Clone() {
    return new TestSmsConditionalTransitionElement(this);
}

public override object Copy(Dictionary<Guid, Guid> dictToRebind,
Core.Campaign.CampaignSchema parentSchema) {
    return new TestSmsConditionalTransitionElement(this, dictToRebind,
parentSchema);
}

// Overrides the factory method for creating the executed element
// Returns the element with the TestSmsConditionalTransitionFlowElement type
public override ProcessFlowElement CreateProcessFlowElement(UserConnection
userConnection) {
    var sourceElement = SourceRef as TestSmsElement;
    var executableElement = new TestSmsConditionalTransitionFlowElement {
        UserConnection = userConnection,
        TestSmsResponses = TestSmsResponseId,
        PhoneNumber = sourceElement.PhoneNumber,
        SmsText = sourceElement.SmsText
    };
    InitializeCampaignProcessFlowElement(executableElement);
    InitializeCampaignTransitionFlowElement(executableElement);
    InitializeConditionalTransitionFlowElement(executableElement);
    return executableElement;
}
}

```

Save and publish the created schema.

5. Creating the executed element for the flow from the [Bulk SMS] element

To add a functionality that would take into consideration the response type as per the sent bulk sms, create the executed element. It is a class, the inheritor of the *ConditionalTransitionFlowElement* class.

To create the executed element, add the source code schema with the following properties in the development package:

- [Title] – “TestSmsConditionalTransitionElement”
 - [Name] – “TestSmsConditionalTransitionElement”

Add the following source code to the [Source code] section of the schema:

```
namespace Terrasoft.Configuration
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
```

```

using Terrasoft.Common;
using Terrasoft.Core.DB;

public class TestSmsConditionalTransitionFlowElement :
ConditionalTransitionFlowElement
{
    public string SmsText { get; set; }

    public string PhoneNumber { get; set; }

    public IEnumerable<Guid> TestSmsResponses { get; set; }

    private void ExtendWithResponses() {
        TransitionQuery.CheckArgumentNull("TransitionQuery");
        if (TestSmsResponses.Any()) {
            Query responseSelect = GetSelectByParticipantResponses();
            TransitionQuery.And("ContactId").In(responseSelect);
        }
    }

    private Query GetSelectByParticipantResponses() {
        var responseSelect = new Select(UserConnection)
            .Column("ContactId")
            .From("TestSmsTarget")
            .Where("SmsText").IsEqual(Column.Parameter(SmsText))
            .And("PhoneNumber").IsEqual(Column.Parameter(PhoneNumber))
            .And("TestSmsResponseId")
            .In(Column.Parameters(TestSmsResponses)) as Select;
        responseSelect.SpecifyNoLockHints(true);
        return responseSelect;
    }

    protected override void CreateQuery() {
        base.CreateQuery();
        ExtendWithResponses();
    }
}
}

```

Save and publish the created schema.

6. Creating the CampaignConnectorManager replacing module for adding the flow operation logic

To add specific logic of the flow operation upon changing the source (i.e. the element that the outgoing arrow is generated from), create a new module schema replacing the *CampaignConnectorManager* module in the development package. The procedure for creating a module schema is covered in the “**Creating a custom client module schema**” article. Set the following properties for the created schema:

- [Title] – “TestSmsCampaignConnectorManager”
- [Name] – “TestSmsCampaignConnectorManager”

Add the following source code to the [Source code] section of the schema:

```

define("TestSmsCampaignConnectorManager", [], function() {

Ext.define("Terrasoft.TestSmsCampaignConnectorManager", {
    // Specify replacing of the CampaignConnectorManager module
    override: "Terrasoft.CampaignConnectorManager",

    // Add mapping of the name of arrow source campaign element - arrow type
    (full name)
}

```

```
initMappingCollection: function() {
    this.callParent(arguments);

this.connectorTypesMappingCollection.addIfNotExists("TestSmsElementSchema",
    "Terrasoft.ProcessTestSmsConditionalTransitionSchema");
},

// Virtual method for reload
// Arrow processing logic before its substitute by an arrow with a new type.
additionalBeforeChange: function(prevTransition, sourceItem, targetItem) {
    // additional logic here
},

// Virtual method for reload
// Populating specific fields of the created arrow based on the previous
arrow.
fillAdditionalProperties: function(prevElement, newElement) {
    if (newElement.getTypeInfo().typeName ===
"ProcessTestSmsConditionalTransitionSchema") {
        // Copy the configured responses if the previous arrow is of the same
type
        newElement.testSmsResponseId = prevElement.testSmsResponseId ?
prevElement.testSmsResponseId : null;
        // Copy the configuration of response setup
        newElement.isResponseBasedStart = prevElement.isResponseBasedStart
            ? prevElement.isResponseBasedStart
            : false;
    }
}
);
});

Save the created schema.
```

7. Connecting the **CampaignConnectorManager** replacing module

To connect the module created on the previous step, create a replacing client module and specify *BootstrapModulesV2* from the *NUI package as the parent object*. The procedure of creating a replacing client module is covered in the “**Creating a custom client module schema**“ article.

Add the following source code to the [Source code] section of the schema:

```
// Set the previously created TestSmsCampaignConnectorManager module as a dependency
define("BootstrapModulesV2", ["TestSmsCampaignConnectorManager"], function() {});
```

Save the created schema.

During an actual implementation of this case, we recommend creating a separate [Bulk SMS] object schema. The [TestSmsElement] and [TestSmsConditionalTransitionElement] objects will contain the [Id] of this object and not the *SmsText*, *PhoneNumber*...fields. The *TestSmsCampaignProcessElement* executed element in the *Execute()* method must contain the logic of adding contacts to the bulk sms audience. A separate mechanism (or several mechanisms) must perform sending of the bulk sms and afterwards record the participants' responses. Based on these responses, the arrow will transfer the campaign audience to the following campaign step.

Web-to-Case

Contents

- **Introduction**
- **Creating Web-to-Case landing pages**

Web-to-Case

Beginner

Easy

Medium

Advanced

Introduction

Web-to-Case functionality implements the ability to create cases in the Creatio by filling the required form fields embedded in a third-party site - landing.

The *ProductCore* package depends on the *WebForms* package, that contains Web-to-Case functionality. This means that landings can be used in all products. Pre-configured base functionality is implemented in the service enterprise, customer center, marketing products and all bundles that these products are part of.

More information about landings can be found in the [\[Landings\] section](#) articles of the corresponding products (such as Marketing Creatio).

Web-to-Case configuration can be done in the system interface. To implement generated JavaScript to a third-party site, you need the basic Web development skills.

The Web-to-Case base functionality allows to configure the following features without programming (using minor improvements on a third-party site):

- The form interface and styles.
- List of the additionally passed fields.
- List of default values for the fields that are not displayed in the form.
- The list of domains from which the case registration for each landing will be possible.
- The address to which the user will be redirected after submitting the form.
- JavaScript event handlers of successful/unsuccessful case registration.
- Additional landings, that can be configured in different way. That makes it possible to distinguish cases created from different sites.

You can modify the project to set up a preliminary handler of case registration through the Web-to-Case with the data validation, correction, creation of related entities and etc. The automatic creation of contact for the registered case is configured in the Creatio base configuration in the handler of case registration through the Web form.

The logic of the automatic filling of case fields.

In the process of case registration through the Web form, the following fields are recommended for filling: [Name], [Email], [Phone], [Case subject]. The [Case subject] value will be passed to the new case.

The Creatio will identify the contact by [Name], [Email] and [Phone] fields. The search is performed in a following way:

1. If contact fields matches the [Name], [Email] and [Phone] fields from the filled form, they will be added to the created case.
2. If contact fields matches only the [Name] and [Email] fields from the filled form, they will be added to the created case.
3. If contact fields matches only the [Email] field from the filled form, it will be added to the created case.
4. Otherwise, a new contact is created and the [Name], [Email] and [Phone] fields will be filled in. The created contact is added to the registered case.

If more than one contact are found, then the first contact will be used as contact of the case. Also the case registration date (*RegisteredOn* column) will be automatically filled with the current date and time.

Recommendations for the execution of project solutions

If you need to customize the Web-to-Case, use its base functionality as an example.

To execute the project solution:

1. Create a page schema that is inherited from the *CaseGeneratedWebFormPageV2*. The page should not be a replacement page.
2. Add a record of the new type of landing to the *LandingType* table and localization to the *SysLandingTypeLcz* table.
3. Register the typed page created in the first step (the value of the type is new).
4. If you need preliminary processing of the form data before saving the record in the database, you need to

create a class that implements the *IGeneratedWebFormPreProcessHandler* interface. This class is a preliminary handler for case registration. Implement the *Execute()* method. This method is the entry point to the handler. Additional actions are implemented in this method. You can take the *WebFormCasePreProcessHandler* schema as an example.

5. If you need to perform actions after saving the record in the database, you need to create a class that implements the *IGeneratedWebFormPreProcessHandler* interface. This class is a preliminary handler for case registration. Implement the *Execute()* method and perform necessary actions.
6. If you created the registration handlers of the case, register them in the *WebFormProcessHandlers* table. Use an existing record as an example of registration.
7. Edit the script template that forms the configuration JavaScript object of the landing, and place it in the *ScriptTemplate* localized string of the created page. Specify the similar script for all localizations used. You can find an example of the script in the *CaseGeneratedWebFormPageV2* schema.
8. Bind all created data to the package.

Creating Web-to-Case landing pages

Beginner

Easy

Medium

Advanced

Introduction

Web-to-Case functionality implements the ability to create cases in the Creatio by filling out a web form fields embedded in a landing page on a third-party website.

Web-to-Case landing record can be configured in the system interface in the [Landing pages and web forms] section. To add the JavaScript code (generated by Creatio for each landing record) to a third-party site, you need the basic Web development skills.

More information about landings can be found in the [\[Landing pages and web forms\] section](#) articles of the corresponding products (such as Marketing Creatio). More information about the Web-to-Case functionality can be found in the "**Web-to-Case**" article.

To create a Web-to-Case landing page:

1. Create new landing record in the Creatio.
2. Create new landing page that will contain the code that binds landing form (on the website) and the landing record (in Creatio).
3. Add the landing page to the website.

Steps to create Web-to-Case landing

1. Create new landing record in the Creatio

To create a new landing record, execute the [Add] action in the [Landing pages and web forms] section. Fill in the following fields on the opened page (Fig. 1):

- [Name] – landing page name in Creatio.
- [Website domains] – your landing page URL.
- [Status] – landing status.
- [Redirection URL] – the URL that is opened after the landing page form is completed.

Fig. 1 Landing edit page

The screenshot shows the 'MyLanding' configuration page. On the left, a sidebar menu includes 'Queues', 'Queues settings', 'Landing pages and web forms' (which is selected and highlighted in orange), 'Feed', and 'Dashboards'. The main area has tabs for 'LANDING SETUP', 'ATTACHMENTS AND NOTES', and 'DEFAULT'. Under 'LANDING SETUP', there's a step for setting up a redirection URL with 'Redirection URL' set to 'http://bpmonline.com'. Another step involves copying code and mapping fields, with a script source provided.

When creating a case, you can receive only four fields ("Subject", "Email", "Name" and "Phone") from the landing page. Therefore, you must set the default values for the new landing record(Fig. 2).

Fig. 2 Values by default

This screenshot shows the 'MyLanding' configuration page with the 'DEFAULT VALUES' tab selected. It displays a table for defining default values for fields like 'Account' (set to 'Axiom') and 'Category' (set to 'Incident'). A vertical sidebar on the right contains icons for user profile, gear, question mark, phone, email, and notifications.

Field	Value
Account	Axiom
Category	Incident

Save the page to apply the changes.

2. Create a landing page

To create landing page, you need to create a standard HTML page containing a Web form in any text editor using HTML markup.

To register the data sent via the web-form, add four fields to the form (using `<input>` element) that define the case:

- Case subject
- Contact email
- Contact name
- Contact phone

Specify the `name` and `id` attributes for each field.

To send a form data to Creatio when creating a new [Case] object, you need to add a JavaScript script to the HTML page. Copy the script source code from the [STEP 2. Copy the code and configure and map the fields] field of the landing edit page (Fig. 1).

The script must be copied from the already saved landing.

The script contains the *config* configuration object that has following properties:

- *fields* – contains the object with "Subject", "Email", "Name" and "Phone" values that must match the *id* attribute selectors of the corresponding web form fields.
- *landingId* – contains the landing Id in the database.
- *serviceUrl* – contains URL of the service to which the form data will be sent.
- *redirectUrl* – contains redirection URL specified in the [Redirection URL] field of the landing.
- *onSuccess* – contains a function that handles the successful creation of a case. Optional property.
- *onError* – contains a function that handles the error of the case creation. Optional property.

The *config* configuration object is passed as an argument of the *createObject()* function that must be executed when the form is submitted.

To call the *createObject()* function when sending a form, add the *onSubmit = "createObject(); return false"* attribute to the *<form>* tag of the HTML page of the Landing page (see STEP 3, Fig. 1).

An example of the complete landing page source code for the case registration:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <!--STEP 2-->
    <!--This part needs to be copied from the STEP 2 field of the lending edit page-->
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
    <script src="https://webtracking-v01.creatio.com/JS/track-cookies.js"></script>
    <script src="https://webtracking-v01.creatio.com/JS/create-object.js"></script>
    <script>
        /**
         * Replace the "css-selector" placeholders in the code below with the element
         * selectors on your landing page.
         * You can use #id or any other CSS selector that will define the input field
         * explicitly.
         * Example: "Email": "#MyEmailField".
         * If you don't have a field from the list below placed on your landing,
         leave the placeholder or remove the line.
        */
        var config = {
            fields: {
                "Subject": "#subject-field", // Case subject
                "Email": "#email-field", // Visitor's email
                "Name": "#name-field", // Visitor's name code
                "Phone": "#phone-field", // Visitor's phone number
            },
            landingId: "8ab71187-0428-4372-b81c-fd05b141a2e7",
            serviceUrl:
                "http://localhost/creationservice710/0/ServiceModel/GeneratedObjectWebFormService.svc/
                SaveWebFormObjectData",
                redirectUrl: "http://creatio.com",
                onSuccess: function(response) {
                    window.alert(response.resultMessage);
                },
                onError: function(response) {
                    window.alert(response.resultMessage);
                }
            };
        /**
         * The function below creates a object from the submitted data.
        */
    
```

```
* Bind this function call to the "onSubmit" event of the form or any other
elements events.
* Example: <form class="mainForm" name="landingForm"
onSubmit="createObject(); return false">
/*
function createObject() {
    landing.createObjectFromLanding(config)
}
</script>
<!--STEP 2-->
</head>
<body>
<h1>Landing web-page</h1>
<div>
    <h2>Case form</h2>
    <form class="mainForm" name="landingForm" onSubmit="createObject(); return
false">
        Subject:<br>
        <input type="text" name="subject" id="subject-field"><br>
        Email:<br>
        <input type="text" name="Email" id="email-field"><br>
        Name:<br>
        <input type="text" name="Name" id="name-field"><br>
        Phone:<br>
        <input type="text" name="Phone" id="phone-field"><br><br>
        <input type="submit" value="Submit">
        </font>
    </form>
</div>
</body>
</html>
```

3. Add the page to the website.

A case from the landing page will be added to the Creatio only if the page is hosted on the site whose name is listed in the [Website domains] field of the landing page record in Creatio. If you open the page in the browser locally, then an empty message will be displayed when the case is created.

Fig. 3 Empty message

The screenshot shows a web browser window with the address bar displaying "file:///C:/creatio/Landing/index.html". A modal alert dialog is open, containing the text "This page says:" followed by "null" and an "OK" button. Below the browser window, there is a form with the following fields:

Subject:	<input type="text" value="Test Case"/>
Email:	<input type="text" value="some@mail.com"/>
Name:	<input type="text" value="Test name"/>
Phone:	<input type="text" value="123456789"/>

At the bottom left of the form area is a "Submit" button.

The output of an empty message is configured in the `onError()` method of the configuration object.

If you place the page on the local server of the computer that serves as the [reserved domain name localhost](#) (as specified in the landing setting , Fig. 1), then the script that adds the address from the web page of the landing will work correctly (Fig. 4)

Fig. 4 The correct adding of data

localhost says:
Data successfully saved

OK

Subject:
Test Case

Email:
some@mail.com

Name:
Test name

Phone:
123456789

Submit

As a result, a case with specified parameters will be automatically created.

Fig. 5 Automatically created case

Case #SR00000008: Test Case

What can I do for you? Creatio

Service Contacts Accounts Cases Activities Services Service agreements Configuration items Problems Changes

Resolution time

Priority: Medium

Contact: User Test

Account: Axiom

SLA: 4 — Axiom

Category: Incident

Service: Configuration item

Assignees group

NEXT STEPS (0)

New In progress Waiting for re... Resolved Closed

PROCESSING CLOSURE AND FEEDBACK CASE INFORMATION ATTACHMENTS

History

John Best to: test@gmail.com 5/5/2017 at 10:58 AM via Email

Hello, User Test!

According to your request, we have registered the Incident #SR00000008 "Test Case".

Description: Test Case.

Created on: 5/5/2017 7:58:14 AM;

Read more

Draft

Web-To-Object. Integration via landings and web-forms

Beginner

Easy

Medium

Advanced

Introduction

Web-to-Object is a mechanism for implementing simple one-way integrations with Creatio. It enables you to create records in Creatio sections (leads, cases, orders, etc.) simply by sending the necessary data to the Web-to-Object service.

The most common cases of using the Web-to-Object service are the following:

- Integrating Creatio with custom landings and web forms. The service call is performed from a landing (a customized custom page with a web form), after the visitor submits the completed web form.
- Integrating with external systems to create Creatio objects.

Using Web-to-Object will enable you to configure the registration of virtually any objects in Creatio (e.g., a lead, an order or a case).

The [\[Landings and web-forms\]](#) section is used to work with landing in Creatio. This section is available in all Creatio products, however it might not be enabled in workplaces of certain products (e.g., Sales Creatio). Each record in the [Landing and web-forms] section corresponds to a landing page. The record edit page features a [Landing setup] tab.

Learn more about the process of creating landing pages based on the Web-to-Case mechanism in the “[Creating Web-to-Case landing pages](#)” article.

Implementing the Web-to-Object service

The main functionality of the Web-To-Object mechanism is contained in the *WebForm* package and is common to all products. Depending on the product, this functionality may be extended by the Web-to-Lead (the *WebLeadForm* package), Web-to-Order (the *WebOrderForm* package), and **Web-to-Case** (the *WebCaseForm* package) mechanisms.

To process data received from a web-form of a landing, the *WebForm* package implements the *GeneratedObjectWebFormService* configuration service (the *Terrasoft.Configuration.GeneratedWebService* class). The data of the landing's web-form is accepted as the argument of the *public string SaveWebFormObjectData(FormData formData)* method. They are then passed to the *public void HandleForm(FormData formData)* method of the *Terrasoft.Configuration.WebFormHandler* class, in which the corresponding system object is created.

External API of the Web-to-Object service

To use the service, send a POST request to:

```
[Creatio application  
path]/0/ServiceModel/GeneratedObjectWebFormService.svc/SaveWebFormObjectData
```

For example:

```
http://mycreatio.com/0/ServiceModel/GeneratedObjectWebFormService.svc/SaveWebFormObjectData
```

The content type of the request is *application/json*. In addition to the required cookies, the JSON object containing the data of the web-form must be added to the content of the request. JSON object example:

```
{  
    "formData": {  
        "formId": "d66ebbf6-f588-4130-9f0b-0ba3414dafb8",  
        "formFieldsData": [  
            { "name": "Name", "value": "John Smith" },  
            { "name": "Email", "value": "j.smith@creatio.com" },  
            { "name": "Zip", "value": "00000" },  
            { "name": "MobilePhone", "value": "0123456789" },  
            { "name": "Company", "value": "Creatio" },  
            { "name": "Industry", "value": "" },  
            { "name": "FullJobTitle", "value": "Sales Manager" },  
            { "name": "UseEmail", "value": "" },  
            { "name": "City", "value": "Boston" },  
            { "name": "State", "value": "Massachusetts" }  
        ]  
    }  
}
```

```
        { "name": "Country", "value": "USA" },
        { "name": "Commentary", "value": "" },
        { "name": "BpmHref", "value": "http://localhost/Landing/" },
        { "name": "BpmSessionId", "value": "0ca32d6d-5d60-9444-ec34-5591514b27a3" }
    ]
}
}
```

Using the Web-to-Object service

Integrating with custom landings and web-forms

Integrating Creatio with custom landings and web-forms is covered in the following articles:

- [The \[Landing and web-forms\] section](#)
- [Creating Web-to-Case landing pages](#)

Integrating with external systems

To integrate with external systems:

1. Create a new record in the [\[Landing and web-forms\]](#) section
2. Get the address to the service (*serviceUrl* property) and the identifier (the *landingId* property) from the configuration object of the created record.
3. Implement the process of sending a POST-request to the Web-to-Object service (at the received address) in the external system. Add the necessary data to the request in form of a JSON object. Set the value of the received identifier to the *formId* property of the JSON object.

Setting up web forms for a custom object

Beginner

Easy

Medium

Advanced

Introduction

Create a custom Creatio object via a landing page web form on a third party website. Learn more about landing pages in the “[\[Landing pages and web forms\] section](#)” article.

The general procedure of creating a custom object via a web form is as follows:

1. Register a new landing page type.
2. Add an edit page for the web form.
3. Map the new landing page type to the created edit page.
4. Update the scripts for the web form record page.
5. Create and configure a landing page in the **[Landing pages and web forms]** section.
6. Deploy and set up a landing page with a web form.

Tracking website events only works for leads and does not work for custom objects.

Case description

Create a custom **[Contact]** object using a landing page web form.

Source code

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

Case implementation algorithm

1. Register a new landing page type

To do this:

1. Open the System Designer by clicking . Go to the [System setup] block → click [Lookups].
2. Select the [Landing types] lookup.
3. Add a new record.

In the created record, specify (Fig.1):

- [Name] – “Contact”
- [Object] – “Contact.”

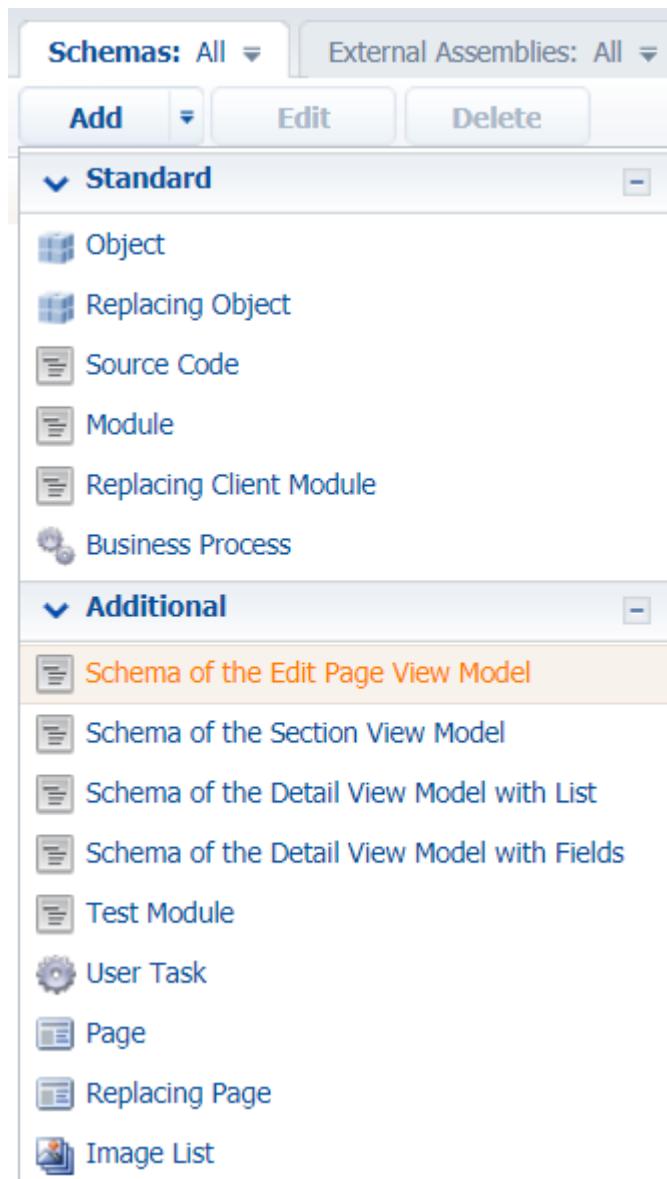
Fig. 1. Setting landing page parameters

Name	Description	Object
Contact		Contact
Case		Case
Event participant		Event participant
Lead		Lead

2. Add an edit page for the web form

Run the [Add] → [Schema of the Edit Page View Model] menu command on the [Schemas] tab in the [Configuration] section of the custom package (Fig. 2). The procedure for creating a view model schema of the edit page is covered in the “**Creating a custom client module schema**” article.

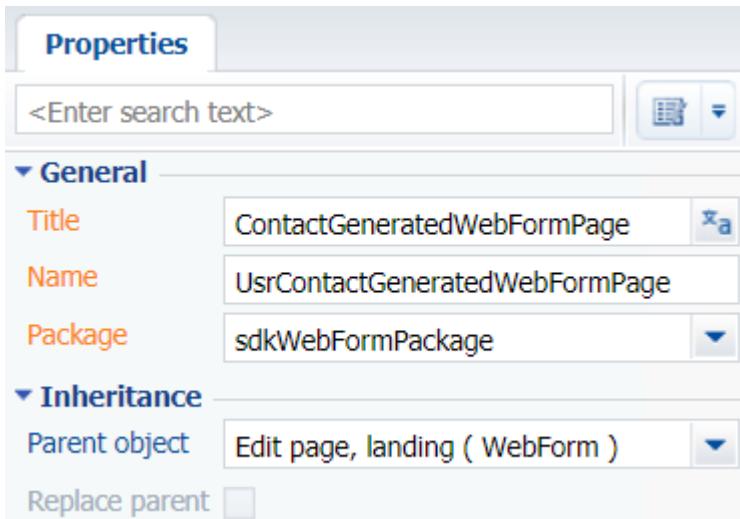
Fig. 2. – Adding a view model schema of the edit page



Specify the following parameters for the created schema of the edit page view model (Fig. 3):

- **[Title]** – "ContactGeneratedWebFormPage"
- **[Name]** – "UsrContactGeneratedWebFormPage"
- **[Parent object]** – "Edit, landing."

Fig. 3. Setting up the mini-page view model schema



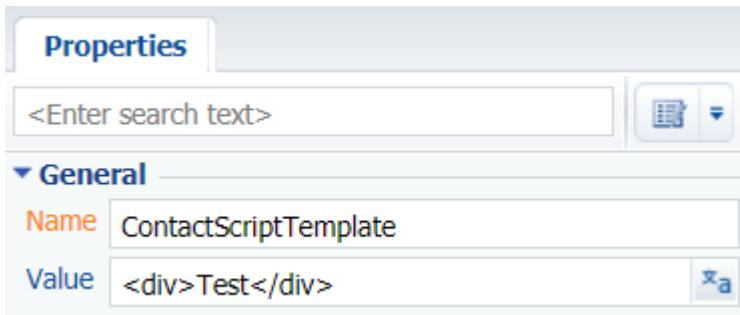
The complete source code of the module is available below:

```
// UsrContactGeneratedWebFormPage - unique schema name.
define("UsrContactGeneratedWebFormPage", ["UsrContactGeneratedWebFormPageResources"],
    function() {
        return {
            details: /**SCHEMA_DETAILS*/{}/**SCHEMA_DETAILS*/,
            methods: {
                /**
                 * @inheritDoc
                 */
                getScriptTemplateFromResources: function() {
                    var scriptTemplate;
                    if (this.getIsFeatureEnabled("OutboundCampaign")) {
                        // ContactScriptTemplate - localizable string name.
                        scriptTemplate =
this.get("Resources.Strings.ContactScriptTemplate");
                    } else {
                        scriptTemplate =
this.get("Resources.Strings.ScriptTemplate");
                    }
                    return scriptTemplate;
                },
                diff: /**SCHEMA_DIFF*/[]/**SCHEMA_DIFF*/
            };
        });
    });
});
```

Save the schema after making the changes.

Add the *ContactScriptTemplate* localizable string. Specify *<div>Test</div>* as the string value (Fig. 4). Learn more about working with localizable strings in the "**Source code designer**" article.

Fig. 4. Setting up a localizable string



Save the schema after making the changes.

3. Map the new landing page type to the created edit page

To do this, add the record to the **[dbo.SysModuleEdit]** DB table. Execute the following SQL script to add the record:

```
-- Parameters of new landing page
DECLARE @editPageName nvarchar(250) = N'UsrContactGeneratedWebFormPage'; -- declare
the name of the created schema
DECLARE @landingTypeName NVARCHAR(250) = N'Contact'; -- the type name of the landing
DECLARE @actionCaption NVARCHAR(250) = N'Contact form'; -- declare the type name for
the landing in the section when creating a new record

-- Set system parameters based on new landing page
DECLARE @generatedWebFormEntityUniqueId uniqueidentifier = '41AE7D8D-BEC3-41DF-A6F0-
2AB0D08B3967';
DECLARE @cardSchemaUniqueId uniqueidentifier = (select top 1 UID from SysSchema where Name
= @editPageName);
DECLARE @pageCaption nvarchar(250) = (select top 1 Caption from SysSchema where Name
= @editPageName);
DECLARE @sysModuleEntityId uniqueidentifier = (select top 1 Id from SysModuleEntity
where SysEntitySchemaUID = @generatedWebFormEntityUniqueId);
DECLARE @landingTypeId uniqueidentifier = (SELECT TOP 1 Id FROM LandingType WHERE
Name = @landingTypeName);

-- Adding new Landing page variant to application interface
INSERT INTO SysModuleEdit
(Id, SysModuleEntityId, TypeColumnName, UseModuleDetails, CardSchemaUID,
ActionKindCaption, ActionKindName, PageCaption)
VALUES
(NEWID(), @sysModuleEntityId, @landingTypeId, 1, @cardSchemaUniqueId, @actionCaption,
@editPageName, @pageCaption)
```

41AE7D8D-BEC3-41DF-A6F0-2AB0D08B3967 – non-editable identifier of the *GeneratedWebForm* entity schema in the **[dbo.SysSchema]** DB table. The ID is relevant for any case featuring adding a landing page for a custom entity.

Clear the browser cache after running the script. As a result, you will be able to add the new **[Contact form]** landing type in the **[Landing pages and web forms]** section. However, the script that must be added to the source code of the landing page will not be immediately available on the landing record page (Fig. 6).

Fig. 5. The record list of the [Landing pages and web forms] section

The screenshot shows the 'Landing pages and web forms' section of the Creatio interface. On the left, a sidebar menu is visible with options like Marketing, Contacts, Campaigns, Email, Landing pages and web forms (which is selected and highlighted in red), Events, Leads, Accounts, and Dashboards. The main area displays a table of landing pages. One row, 'Contact form', is highlighted with a red box. The table columns include Name, Type, Created on, Leads, and Status. The 'Contact form' entry has a Type of 'Lead', was created on 6/3/2015 at 12:30 PM, has 0 leads, and is in an 'Active' status. Another row, 'Bpm'online marketing - free trial', is also shown.

Fig. 6. Landing edit page

The screenshot shows the 'New record' page for a landing page setup. The left sidebar menu includes Marketing, Contacts, Campaigns, Email, Landing pages and web forms (selected and highlighted in red), Events, Leads, Accounts, Dashboards, and Marketing plans. The main area is titled 'New record' and shows a form with fields: Name*, Website domains*, Description, and Status*. Below the form, there is a 'FREQUENTLY ASKED QUESTIONS' section with links to various setup guides. The right side of the screen shows tabs for 'LANDING SETUP', 'ATTACHMENTS AND NOTES', 'DEFAULT VALUES', 'HISTORY', and 'FEED'. Under the 'LANDING SETUP' tab, three steps are outlined: 1. Set up a redirection URL (optional), 2. Copy the code and configure and map the fields, and 3. Insert the customized code into the landing page source code. Step 3 includes a code snippet: `onSubmit="createObject(); return false"`.

4. Update the scripts for the web form record page

The value of the variable contains an escaped HTML string with `<script>` tags and other information for setting up web form field clusters – the columns for the created entity. This value must be localizable. To do so, execute the following SQL script:

```
-- Landing edit page schema name
DECLARE @editPageName nvarchar(250) = N'UsrContactGeneratedWebFormPage'; -- declare the name of the created schema

-- region Scripts' structure
DECLARE @scriptPrefix nvarchar(max) = N'<div style="font-family: "Courier New"; monospace; font-size: 10pt;">&lt;script src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"&gt;&lt;/script&gt;
```

```

<br>&lt;script src='https://webtracking-v01.bpmonline.com/JS/track-cookies.js'>&lt;/script&gt;
<br>&lt;script src="#apiUrl##">&lt;/script&gt;<br>/*
<br>*&nbsp;Replace&nbsp;the&nbsp;'&quot;<span style="color: #0c0cec;">css-
selector</span>&quot;&nbsp;placeholders&nbsp;in&nbsp;the&nbsp;code&nbsp;below&nbsp;wi-
th&nbsp;element&nbsp;selectors&nbsp;on&nbsp;your&nbsp;landing&nbsp;page.
<br>*&nbsp;You&nbsp;can&nbsp;use&nbsp;#id&nbsp;or&nbsp;any&nbsp;other&nbsp;CSS&nbsp;s-
elector&nbsp;that&nbsp;will&nbsp;define&nbsp;the&nbsp;input&nbsp;field&nbsp;explici-
tly.<br>*&nbsp;Example:&nbsp;&quot;Email"&quot;:&nbsp;&quot;#MyEmailField"&quot;.
<br>*&nbsp;If&nbsp;you&nbsp;don&quot;t&nbsp;have&nbsp;a&nbsp;field&nbsp;from&nbsp;the
&nbsp;list&nbsp;below&nbsp;placed&nbsp;on&nbsp;your&nbsp;landing,&nbsp;leave&nbsp;the
&nbsp;placeholder&nbsp;or&nbsp;remove&nbsp;the&nbsp;line.
<br>*/<br>var config = {
    fields: [
        {
            DECLARE @sqriptDelimiter nvarchar(max) =
N'<br>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;';
            DECLARE @sqriptSuffix nvarchar(max) = N'<br>&nbsp;&nbsp;&nbsp;&nbsp;';
            <br>&nbsp;&nbsp;&nbsp;&nbsp;landingId: ##landingId##,
            <br>&nbsp;&nbsp;&nbsp;&nbsp;serviceUrl: ##serviceUrl##,
            <br>&nbsp;&nbsp;&nbsp;&nbsp;redirectUrl: ##redirectUrl##<br>};
        <br>*&nbsp;The&nbsp;function&nbsp;below&nbsp;creates&nbsp;a&nbsp;object&nbsp;from&nbs-
p;the&nbsp;submitted&nbsp;data.
<br>*&nbsp;Bind&nbsp;this&nbsp;function&nbsp;call&nbsp;to&nbsp;the&nbsp;'onSubmi-
t'"&nbsp;event&nbsp;of&nbsp;the&nbsp;form&nbsp;or&nbsp;any&nbsp;other&nbsp;ele-
ments&nbsp;events.
<br>*&nbsp;Example:&nbsp;&lt;form class='mainForm' name='lan-
dingForm' onsubmit='createObject(); return false;'&gt;
<br>*&nbsp;&lt;function createObject() &nbsp;
{&lt;script&gt; landing.createObjectFromLanding(config)&lt;/script&gt;}<br>/*
<br>*&nbsp;The&nbsp;function&nbsp;below&nbsp;inits&nbsp;landing&nbsp;page&nbsp;using&
nbsp;URL&nbsp;parameters.<br>*&nbsp;&lt;function initLanding() &nbsp;
{&lt;script&gt; landing.initLanding(config)&lt;/script&gt;}&lt;/function&gt;
<br>jQuery(document).ready(initLanding)&lt;/div&gt;';
-- endregion

-- region Scripts' variables
DECLARE @sqriptNameColumn nvarchar(max); -- declare column variables to map to the
landing
DECLARE @sqriptEmailColumn nvarchar(max);
DECLARE @scriptResult nvarchar(max);
-- endregion

-- Adding entity columns.
SET @sqriptNameColumn = N'"Name"&quot;:&nbsp;&quot;<span style="color:
#0c0cec;">css-selector</span>&quot;,&nbsp;//&nbsp;Name&nbsp;of&nbsp;a&nbsp;contact';
SET @sqriptEmailColumn = N'"Email"&quot;:&nbsp;&quot;<span style="color:
#0c0cec;">css-selector</span>&quot;,&nbsp;//&nbsp;Email';
-- Concat result scripts.
SET @scriptResult = @sqriptPrefix + @sqriptNameColumn + @sqriptDelimiter +
@sqriptEmailColumn + @sqriptSuffix;

-- Set new localizable scripts value for resource with name like '%ScriptTemplate'
UPDATE SysLocalizableValue
SET [Value] = @scriptResult
WHERE SysSchemaId = (SELECT TOP 1 Id FROM SysSchema WHERE [Name] = @editPageName)
and [Key] like '%ScriptTemplate.Value'

```

In the *Adding entity columns* block, add the names of the entity columns to be filled with the values from the web form.

Replace the double quote ("") and space () characters with the " and HTML character entity references.

Add the `(@scriptVariableNameColumn)` variable and [concatenate](#) it to `scriptResult` for adding a field.

If the values of other fields (except for `Name` and `Email`) are required after completing the setup, re-run the script from this paragraph after registering all of the required columns, including the existing ones, in the *Adding entity columns* block. When the script is re-run, the settings created earlier are updated.

After the script execution completes, open the schema created in the configuration and re-save it to re-save the resources as well. As a result, when selecting **[Contact form]** in the **[Landing pages and web forms]** section, Creatio will display the landing edit page (Fig. 6) combined with the script to copy to the landing page source code.

Fig. 7. Landing edit page

The script contains the `config` configuration object, which has the following properties defined:

- `fields` – contains the `Name` and `Email` properties. Their values must match the selectors of the `id` attributes of the corresponding field of the web form.
- `landingId` – the landing page ID in the database.
- `serviceUrl` – the URL of the service to which the web form data will be sent.
- `onSuccess` – the handler function to process a successful contact creation. Optional property.
- `onError` – the handler function to process a contact creation error. Optional property.

The configuration to be formed is displayed below.

```
var config = {
    fields: {
        "Name": "css-selector", // Contact name
        "Email": "css-selector", // Email name
    },
    landingId: "b73790ab-acb1-4806-baea-4342a1f3b2a8",
    serviceUrl:
"http://localhost:85/0/ServiceModel/GeneratedObjectWebFormService.svc/SaveWebFormObj
ctData",
    redirectUrl: ""
};
```

5. Create and configure a landing page in the [Landing pages and web forms] section

To do this, go the **[Landing pages and web forms]** section and click **[Contact form]**. Learn more about adding **[Landing pages and web forms]** records in the “[Adding new records in the \[Landing pages and web forms\] section](#)” article.

In the created record, specify (Fig. 8):

- **[Name]** – “Contact”
- **[Website domains]** – "<http://localhost:85/Landing/LandingPage.aspx>"
- **[Status]** – “Active”.

Fig. 8. Landing edit page

Save the page to apply the changes.

6. Deploy and set up a landing page with a web form

Create a landing page with a web form using HTML markup in any text editor. Learn more about creating a landing page and adding the landing script in the "[How to connect your website landing page to Creatio](#)" article.

To register contact data sent via the web form in Creatio, add the following fields (`<input>` HTML element) to the source code of the landing page:

- Contact name.
- Contact email.

Specify the `name` and `id` attributes for each field.

To create a new **[Contact]** object when sending web form data to Creatio, add the JavaScript code to the landing page. Copy the source code from the **[STEP 2. Copy the code and configure and map the fields]** field on the landing record page (fig 8.).

The `config` object is passed as an argument of the `createObject()` function, which is executed upon submitting the web form.

Ensure that the `createObject()` function is called upon submitting the web form. To do this, add the `onSubmit="createObject(); return false"` attribute to the `<form>` element. You can retrieve the required attribute value from the **[STEP 3. Insert the customized code into the landing page source code. Set up a function to create the object on form submit]** field on the landing edit page (Fig. 8).

The complete source code of the landing page is available below:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"><!--WAP 2--> <!--Copy this part from the STEP 2 field of
the landing edit page--><script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js"></script>
<script src="https://webtracking-v01.bpmonline.com/JS/track-cookies.js"></script>
<script src="https://webtracking-v01.bpmonline.com/JS/create-object.js"></script>
<script>
```

```
/***
 * Replace the "css-selector" placeholders in the code below with the element
selectors on your landing page.
 * You can use #id or any other CSS selector that will define the input field
explicitly.
 * Example: "Email": "#MyEmailField".
 * If you don't have a field from the list below placed on your landing, leave the
placeholder or remove the line.
 */
var config = {
    fields: {
        "Name": "#name-field", // Name of a contact
        "Email": "#email-field", // Email
    },
    landingId: "b73790ab-acb1-4806-baea-4342a1f3b2a8",
    serviceUrl:
"http://localhost:85/0/ServiceModel/GeneratedObjectWebFormService.svc/SaveWebFormObje
ctData",
    redirectUrl: "",
    onSuccess: function(response) {
        window.alert(response.resultMessage);
    },
    onError: function(response) {
        window.alert(response.resultMessage);
    }
};
/***
 * The function below creates a object from the submitted data.
 * Bind this function call to the "onSubmit" event of the form or any other
elements events.
 * Example: <form class="mainForm" name="landingForm" onSubmit="createObject();
return false">
*/
function createObject() {
    landing.createObjectFromLanding(config)
}
/***
 * The function below inits landing page using URL parameters.
*/
function initLanding() {
    landing.initLanding(config)
}
jQuery(document).ready(initLanding)
</script><!--#AWF 2--></head>
<body>
<h1>Landing web-page</h1>
<div>
    <h2>Contact form</h2>
    <form method="POST" class="mainForm" name="landingForm" onSubmit="createObject();
return false">Name:<br><input type="text" name="Name" id="name-field"><br> Email:<br>
<input type="text" name="Email" id="email-field"><br><br><br><input type="submit"
value="Submit">
    </font>
    </form>
</div>
</body>
</html>
```

Open the landing page Specify the values for the created contact (Fig. 9):

- [Name] – "New User"

- **[Email]** – "new_user@creatio.com".

Fig. 9. Landing page

Landing web-page

Contact form

Name:

New User

Email:

new_user@creatio.com

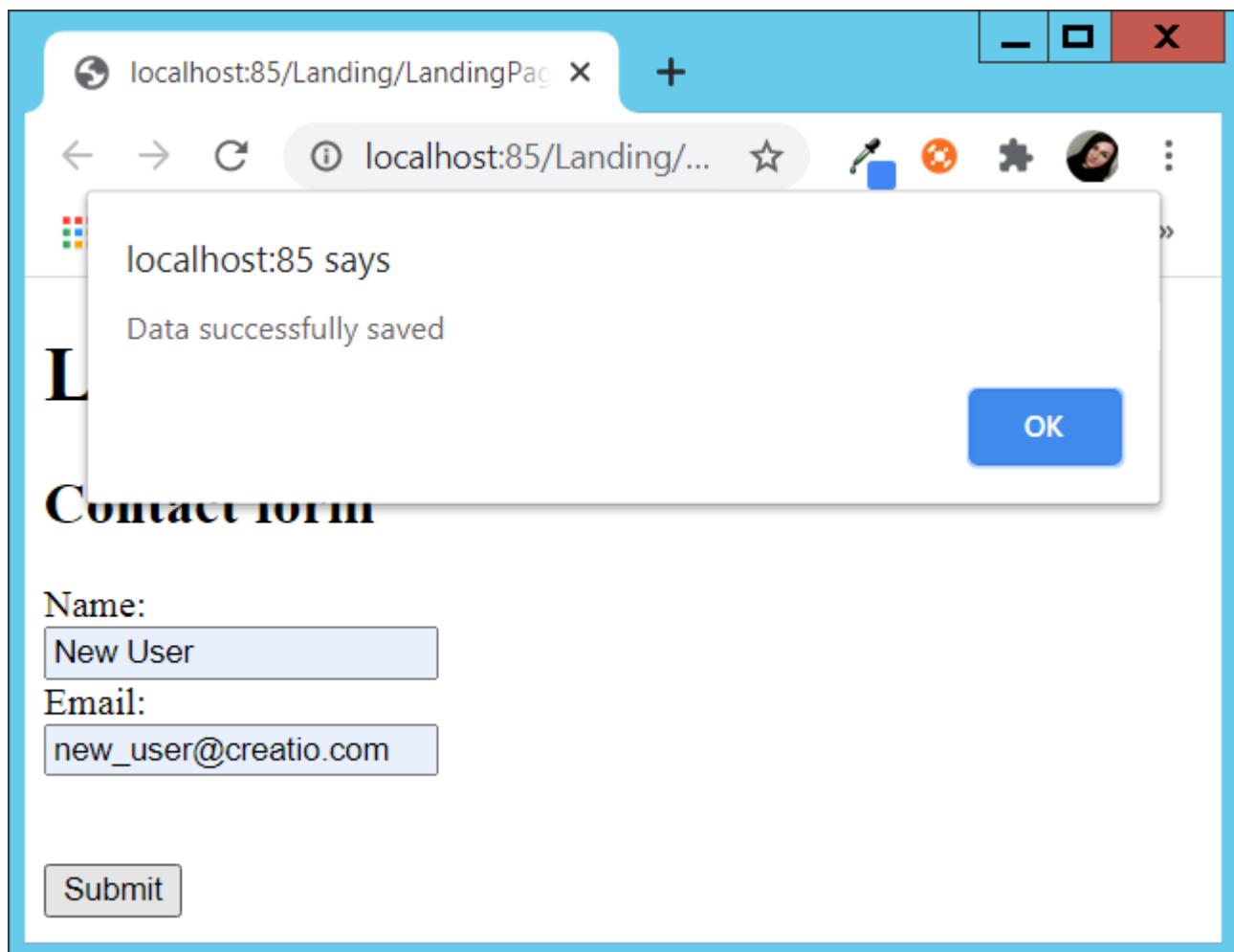
Submit

Click **[Submit]** to create the contact.

The contact from the landing page will only be added if the domain of the landing page is specified in the **[Website domains]** field on the landing record page.

If you place the page on the local server available via the ["localhost" reserved domain name](#) (as specified in the landing page setup, Fig. 8), then the script for creating a contact from the landing page will work correctly (Fig. 10).

Fig. 10. A message about successful data addition



As a result, Creatio will create a contact with the specified parameters (Fig. 11).

Fig. 11. Automatically created contact

The screenshot shows the Creatio application interface. On the left, a sidebar menu lists "Marketing", "Contacts", "Campaigns", "Email", "Landing pages and web forms", "Events", "Leads", and "Accounts". The "Contacts" item is selected. The main area is titled "Contacts" and shows a list of contacts. One contact, "Andrew Baker (sample)", is listed with details: Job title - Specialist, Email - a.baker@ac.com, Business phone - +1 617 440 2031, and Mobile phone - +1 617 221 5187. Below this, another contact, "Email Supervisor", is listed. A new contact, "New User", is shown in a red-bordered box, indicating it is the focus. Its details are: Account - Accom (sample), Email - new_user@creatio.com. To the right of the contact list, there is a vertical toolbar with icons for user profile, settings, help, phone, email, messaging, notifications, and a search bar. The top right corner displays the Creatio logo and version information: "Creatio 7.16.2.1600 VIEW".

Prediction

Contents

- How to implement custom prediction model

How to implement custom prediction model

Beginner

Easy

Medium

Advanced

Introduction

Prediction service of the lookup field uses methods of statistic analysis for learning on the base of historical data and prediction of values for new records.

For more information about these functions please refer to the “[Machine learning service](#)” article.

Case description

Implement automatic prediction for the [AccountCategory] column by the values of the [Country], [EmployeesNumber] and [Industry] field while saving the account record. The following conditions should be met:

- Model learning should be created on the base of account records for last 90 days.
- Model Retraining should be performed every 30 days.
- Permissible value of prediction accuracy for the model – 0,6.

To complete this case you need to check the correctness of the value of the [Creatio cloud services API key] (*CloudServicesAPIKey* code) system setting and the URL of the predictive service in the [Service endpoint Url] field of the [ML problem types] lookup.

You can use the page UI-tools (fields and filters), as well as the [Predict data] business process element to perform the described case. You can find more cases on implementing prediction using the default Creatio tools in the [“Predictive analysis”](#) article.

Case implementation algorithm

1. Model learning

To train the model:

1. Add a record to the [ML Model] lookup. Values of the record fields are given in the Table 1.

Table 1. Values of the record fields of the MLModel lookup

Field	Value
Name	Predict account category
ML problem type	Lookup prediction
Target schema for prediction	Account
Quality metric low limit	0,6
Model retrain frequency (days)	30
Training set metadata	{ "inputs": [{ "name": "CountryId", "type": "Lookup", "isRequired": true }, { "name": "EmployeesNumberId", "type": "Lookup", "isRequired": true },] }

```
{  
    "name": "IndustryId",  
    "type": "Lookup",  
    "isRequired": true  
}  
],  
"output": {  
    "name": "AccountCategoryId",  
    "type": "Lookup",  
    "displayName": "AccountCategory"  
}  
}  
}  
  
Training set query  


```
new Select(userConnection)
.Column("a", "Id").As("Id")
.Column("a", "CountryId")
.Column("a", "EmployeesNumberId")
.Column("a", "IndustryId")
.Column("a", "AccountCategoryId")
.Column("c", "Name").As("AccountCategory")
.From("Account").As("a")
.InnerJoin("AccountCategory").As("c").On("c",
"Id").IsEqual("a", "AccountCategoryId")
.Where("a",
"CreatedOn").IsGreater(Column.Parameter(DateTime.Now.AddDays(-
90)))
```


```

You can find examples of queries in the "**Creating data queries for the machine learning model**" article.

Predictions enabled
(checkbox)

Enable

2. Perform the [Execute model training job] action on the [ML Model] lookup field.

Wait until the values of the [Model processing status] field will be changed in following sequence: *DataTransfer*, *QueuedToTrain*, *Training*, *Done*. The process may take several hours to finish (it depends on the amount of passed data and general workload of the predictive service).

2. Performing the prediction

To start the predictions:

1. Create a business process in the user package. Select the saving of the [Contact] object as a start signal for the process. Check if the required fields are populated (Fig. 1).

Fig. 1. Start signal properties.



Which type of signal is received?

Object signal

Object*

Account

Which event should trigger the signal?

Record added

The added record must meet filter conditions

Actions ▾

- No. of employees
is filled in
- Industry is filled in
- AND
- Country is filled in

2. Add the *MLModelId* lookup parameter that refers to the [ML Model] entity. Select the record with the [Predict account category] model as a value.

3. Add the *RecordId* lookup parameter that refers to the [Account] entity. Select a reference for the *RecordId* parameter of the [Signal] element as a value.

4. Add a [Script task] element on the business process diagram and add the following code there:

```
var userConnection = Get<UserConnection>("UserConnection");
// Getting the Id of the Account record.
Guid entityId = Get<Guid>("RecordId");
// Getting the id of the model.
var modelId = Get<Guid>("MLModelId");
var connectionArg = new ConstructorArgument("userConnection", userConnection);
// Object for calling prediction.
var predictor = ClassFactory.Get<MLEntityPredictor>(connectionArg);
// Call of the forecasting service. The Data is saved in MLPrediction and in case of
high probability of forecasting the data is saved in the required field of the
Account.
predictor.PredictEntityValueAndSaveResult(modelId, entityId);
return true;
```

After saving and compiling the process, the prediction will be performed for new accounts. The prediction will be displayed on the account edit page.

This implementation of the prediction slows down the saving an account record because call of the prediction service

is executed in 2 seconds. This can reduce the performance of the mass operations with data saving, like import from Excel.